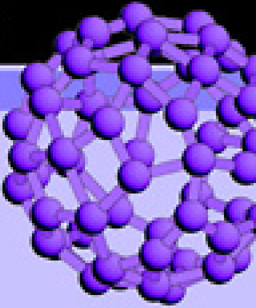
An Introduction to Computational Physics

SECOND EDITION

TAO PANG



CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521825696

An Introduction to Computational Physics

Numerical simulation is now an integrated part of science and technology. Now in its second edition, this comprehensive textbook provides an introduction to the basic methods of computational physics, as well as an overview of recent progress in several areas of scientific computing. The author presents many step-by-step examples, including program listings in Java^{TM}, of practical numerical methods from modern physics and areas in which computational physics has made significant progress in the last decade.

The first half of the book deals with basic computational tools and routines, covering approximation and optimization of a function, differential equations, spectral analysis, and matrix operations. Important concepts are illustrated by relevant examples at each stage. The author also discusses more advanced topics, such as molecular dynamics, modeling continuous systems, Monte Carlo methods, the genetic algorithm and programming, and numerical renormalization.

This new edition has been thoroughly revised and includes many more examples and exercises. It can be used as a textbook for either undergraduate or first-year graduate courses on computational physics or scientific computation. It will also be a useful reference for anyone involved in computational research.

Tao Pang is Professor of Physics at the University of Nevada, Las Vegas. Following his higher education at Fudan University, one of the most prestigious institutions in China, he obtained his Ph.D. in condensed matter theory from the University of Minnesota in 1989. He then spent two years as a Miller Research Fellow at the University of California, Berkeley, before joining the physics faculty at the University of Nevada, Las Vegas in the fall of 1991. He has been Professor of Physics at UNLV since 2002. His main areas of research include condensed matter theory and computational physics.

An Introduction to Computational Physics

Second Edition

Tao Pang
University of Nevada, Las Vegas

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521825696

First published in print format 2006

ISBN-13 978-0-511-14046-4 eBook (NetLibrary)
ISBN-10 0-511-14046-0 eBook (NetLibrary)

ISBN-13 978-0-521-82569-6 hardback
ISBN-10 0-521-82569-5 hardback

ISBN-13 978-0-521-53276-1
ISBN-10 0-521-53276-0

To Yunhua, for enduring love

Contents

Preface to first edition

The beauty of Nature is in its detail. If we are to understand different layers of scientific phenomena, tedious computations are inevitable. In the last half-century, computational approaches to many problems in science and engineering have clearly evolved into a new branch of science, *computational science*. With the increasing computing power of modern computers and the availability of new numerical techniques, scientists in different disciplines have started to unfold the mysteries of the so-called *grand challenges*, which are identified as scientific problems that will remain significant for years to come and may require teraflop computing power. These problems include, but are not limited to, global environmental modeling, virus vaccine design, and new electronic materials simulation.

Computational physics, in my view, is the foundation of computational science. It deals with basic computational problems in physics, which are closely related to the equations and computational problems in other scientific and engineering fields. For example, numerical schemes for Newton's equation can be implemented in the study of the dynamics of large molecules in chemistry and biology; algorithms for solving the Schrödinger equation are necessary in the study of electronic structures in materials science; the techniques used to solve the diffusion equation can be applied to air pollution control problems; and numerical simulations of hydrodynamic equations are needed in weather prediction and oceanic dynamics.

Important as computational physics is, it has not yet become a standard course in the curricula of many universities. But clearly its importance will increase with the further development of computational science. Almost every college or university now has some networked workstations available to students. Probably many of them will have some closely linked parallel or distributed computing systems in the near future. Students from many disciplines within science and engineering now demand the basic knowledge of scientific computing, which will certainly be important in their future careers. This book is written to fulfill this need.

Some of the materials in this book come from my lecture notes for a computational physics course I have been teaching at the University of Nevada, Las Vegas. I usually have a combination of graduate and undergraduate students from physics, engineering, and other majors. All of them have some access to the workstations or supercomputers on campus. The purpose of my lectures is to provide the students with some basic materials and necessary guidance so they can work out the assigned problems and selected projects on the computers available to them and in a programming language of their choice.

This book is made up of two parts. The first part (Chapter 1 through Chapter 6) deals with the basics of computational physics. Enough detail is provided so that a well-prepared upper division undergraduate student in science or engineering will have no difficulty in following the material. The second part of the book (Chapter 7 through Chapter 12) introduces some currently used simulation techniques and some of the newest developments in the field. The choice of subjects in the second part is based on my judgment of the importance of the subjects in the future. This part is specifically written for students or beginning researchers who want to know the new directions in computational physics or plan to enter the research areas of scientific computing. Many references are given there to help in further studies.

In order to make the course easy to digest and also to show some practical aspects of the materials introduced in the text, I have selected quite a few exercises. The exercises have different levels of difficulty and can be grouped into three categories. Those in the first category are simple, short problems; a student with little preparation can still work them out with some effort at filling in the gaps they have in both physics and numerical analysis. The exercises in the second category are more involved and aimed at well-prepared students. Those in the third category are mostly selected from current research topics, which will certainly benefit those students who are going to do research in computational science.

Programs for the examples discussed in the text are all written in standard Fortran 77, with a few exceptions that are available on almost all Fortran compilers. Some more advanced programming languages for data parallel or distributed computing are also discussed in Chapter 12. I have tried to keep all programs in the book structured and transparent, and I hope that anyone with knowledge of any programming language will be able to understand the content without extra effort. As a convention, all statements are written in upper case and all comments are given in lower case. From my experience, this is the best way of presenting a clear and concise Fortran program. Many sample programs in the text are explained in sufficient detail with commentary statements. I find that the most efficient approach to learning computational physics is to study well-prepared programs. Related programs used in the book can be accessed via the World Wide Web at the URL `http://www.physics.unlv.edu/~pang/cp.html`. Corresponding programs in C and Fortran 90 and other related materials will also be available at this site in the future.

This book can be used as a textbook for a computational physics course. If it is a one-semester course, my recommendation is to select materials from Chapters 1 through 7 and Chapter 11. Some sections, such as 4.6 through 4.8, 5.6, and 7.8, are good for graduate students or beginning researchers but may pose some challenges to most undergraduate students.

Tao Pang
Las Vegas, Nevada

Preface

Since the publication of the first edition of the book, I have received numerous comments and suggestions on the book from all over the world and from a far wider range of readers than anticipated. This is a firm testament of what I claimed in the Preface to the first edition that computational physics is truly the foundation of computational science.

The Internet, which connects all computerized parts of the world, has made it possible to communicate with students who are striving to learn modern science in distant places that I have never even heard of. The main drive for having a second edition of the book is to provide a new generation of science and engineering students with an up-to-date presentation to the subject.

In the last decade, we have witnessed steady progress in computational studies of scientific problems. Many complex issues are now analyzed and solved on computers. New paradigms of global-scale computing have emerged, such as the Grid and web computing. Computers are faster and come with more functions and capacity. There has never been a better time to study computational physics.

For this new edition, I have revised each chapter in the book thoroughly, incorporating many suggestions made by the readers of the first edition. There are more examples given with more sample programs and figures to make the explanation of the material easier to follow. More exercises are given to help students digest the material. Each sample program has been completely rewritten to reflect what I have learned in the last few years of teaching the subject. A lot of new material has been added to this edition mainly in the areas in which computational physics has made significant progress and a difference in the last decade, including one chapter on genetic algorithm and programming. Some material in the first edition has been removed mainly because there are more detailed books on those subjects available or they appear to be out of date. The website for this new edition is at `http://www.physics.unlv.edu/~pang/cp2.html`.

References are cited for the sole purpose of providing more information for further study on the relevant subjects. Therefore they may not be the most authoritative or defining work. Most of them are given because of my familiarity with, or my easy access to, the cited materials. I have also tried to limit the number of references so the reader will not find them overwhelming. When I have had to choose, I have always picked the ones that I think will benefit the readers most.

Java is adopted as the instructional programming language in the book. The source codes are made available at the website. Java, an object-oriented and interpreted language, is the newest programming language that has made a major impact in the last few years. The strength of Java is in its ability to work with web browsers, its comprehensive API (application programming interface), and its built-in security and network support. Both the source code and bytecode can run on any computer that has Java with exactly the same result. There are many advantages in Java, and its speed in scientific programming has steadily increased over the last few years. At the moment, a carefully written Java program, combined with static analysis, just-in-time compiling, and instruction-level optimization, can deliver nearly the same raw speed as C or Fortran. More scientists, especially those who are still in colleges or graduate schools, are expected to use Java as their primary programming language. This is why Java is used as the instructional language in this edition. Currently, many new applications in science and engineering are being developed in Java worldwide to facilitate collaboration and to reduce programming time. This book will do its part in teaching students how to build their own programs appropriate for scientific computing. We do not know what will be the dominant programming language for scientific computing in the future, but we do know that scientific computing will continue playing a major role in fundamental research, knowledge development, and emerging technology.

Acknowledgments

Most of the material presented in this book has been strongly influenced by my research work in the last 20 years, and I am extremely grateful to the University of Minnesota, the Miller Institute for Basic Research in Science at the University of California, Berkeley, the National Science Foundation, the Department of Energy, and the W. M. Keck Foundation for their generous support of my research work.

Numerous colleagues from all over the world have made contributions to this edition while using the first edition of the book. My deepest gratitude goes to those who have communicated with me over the years regarding the topics covered in the book, especially those inspired young scholars who have constantly reminded me that the effort of writing this book is worthwhile, and the students who have taken the course from me.

Chapter 1
Introduction

Computing has become a necessary means of scientific study. Even in ancient times, the quantification of gained knowledge played an essential role in the further development of mankind. In this chapter, we will discuss the role of computation in advancing scientific knowledge and outline the current status of computational science. We will only provide a quick tour of the subject here. A more detailed discussion on the development of computational science and computers can be found in Moreau (1984) and Nash (1990). Progress in parallel computing and global computing is elucidated in Koniges (2000), Foster and Kesselman (2003), and Abbas (2004).

1.1 Computation and science

Modern societies are not the only ones to rely on computation. Ancient societies also had to deal with quantifying their knowledge and events. It is interesting to see how the ancient societies developed their knowledge of numbers and calculations with different means and tools. There is evidence that carved bones and marked rocks were among the early tools used for recording numbers and values and for performing simple estimates more than 20 000 years ago.

The most commonly used number system today is the *decimal system*, which was in existence in India at least 1500 years ago. It has a radix (base) of 10. A number is represented by a string of figures, with each from the ten available figures (0–9) occupying a different decimal level. The way a number is represented in the decimal system is not unique. All other number systems have similar structures, even though their radices are quite different, for example, the *binary system* used on all digital computers has a radix of 2. During almost the same era in which the Indians were using the decimal system, another number system using dots (each worth one) and bars (each worth five) on a base of 20 was invented by the Mayans. A symbol that looks like a closed eye was used for zero. It is still under debate whether the Mayans used a base of 18 instead of 20 after the first level of the hierarchy in their number formation. They applied these dots and bars to record multiplication tables. With the availability of those tables, the

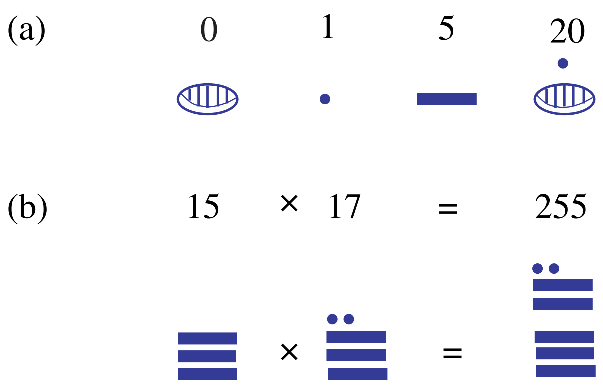


Fig. 1.1 The Mayan number system: (a) examples of using dots and bars to represent numbers; (b) an example of recording multiplication.

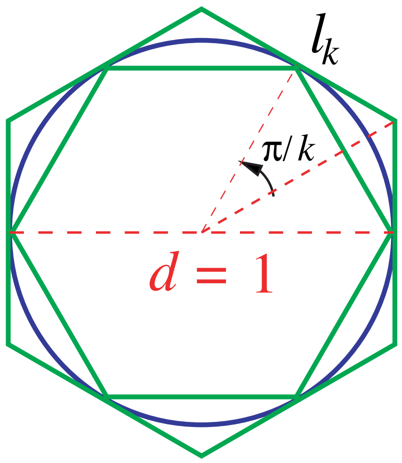


Fig. 1.2 A circle inscribed and circumscribed by two hexagons. The inside polygon sets the lower bound while the outside polygon sets the upper bound of the circumference.

Mayans studied and calculated the period of lunar eclipses to a great accuracy. An example of *Mayan number system* is shown in Fig. 1.1.

One of the most fascinating numbers ever calculated in human history is π, the ratio of the circumference to the diameter of the circle. One of the methods of evaluating π was introduced by Chinese mathematician Liu Hui, who published his result in a book in the third century. The circle was approached and bounded by two sets of regular polygons, one from outside and another from inside of the circle, as shown in Fig. 1.2. By evaluating the side lengths of two 192-sided regular polygons, Liu found that $3.1410 < \pi < 3.1427$, and later he improved his result with a 3072-sided inscribed polygon to obtain $\pi \simeq 3.1416$. Two hundred years later, Chinese mathematician and astronomer Zu Chongzhi and his son Zu Gengzhi carried this type of calculation much further by evaluating the side lengths of two 24 576-sided regular polygons. They concluded that $3.141\,592\,6 < \pi < 3.141\,592\,7$, and pointed out that a good approximation was given by

$\pi \simeq 355/113 = 3.141\,592\,9\ldots$. This is extremely impressive considering the limited mathematics and computing tools that existed then. Furthermore, no one in the next 1000 years did a better job of evaluating π than the Zus.

The Zus could have done an even better job if they had had any additional help in either mathematical knowledge or computing tools. Let us quickly demonstrate this statement by considering a set of evaluations on polygons with a much smaller number of sides. In general, if the side length of a regular k-sided polygon is denoted as l_k and the corresponding diameter is taken to be the unit of length, then the approximation of π is given by

$$\pi_k = kl_k. \tag{1.1}$$

The exact value of π is the limit of π_k as $k \to \infty$. The value of π_k obtained from the calculations of the k-sided polygon can be formally written as

$$\pi_k = \pi_\infty + \frac{c_1}{k} + \frac{c_2}{k^2} + \frac{c_3}{k^3} + \cdots, \tag{1.2}$$

where $\pi_\infty = \pi$ and c_i, for $i = 1, 2, \ldots, \infty$, are the coefficients to be determined. The expansion in Eq. (1.2) is truncated in practice in order to obtain an approximation of π. Then the task left is to solve the equation set

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \tag{1.3}$$

for $i = 1, 2, \ldots, n$, if the expansion in Eq. (1.2) is truncated at the $(n-1)$th order of $1/k$ with $a_{ij} = 1/k_i^{j-1}$, $x_1 = \pi_\infty$, $x_j = c_{j-1}$ for $j > 1$, and $b_i = \pi_{k_i}$. The approximation of π is then given by the approximate π_∞ obtained by solving the equation set. For example, if $\pi_8 = 3.061\,467$, $\pi_{16} = 3.121\,445$, $\pi_{32} = 3.136\,548$, and $\pi_{64} = 3.140\,331$ are given from the regular polygons inscribing the circle, we can truncate the expansion at the third order of $1/k$ and then solve the equation set (see Exercise 1.1) to obtain π_∞, c_1, c_2, and c_3 from the given π_k. The approximation of $\pi \simeq \pi_\infty$ is 3.141 583, which has five digits of accuracy, in comparison with the exact value $\pi = 3.141\,592\,65\ldots$. The values of π_k for $k = 8, 16, 32, 64$ and the extrapolation π_∞ are all plotted in Fig. 1.3. The evaluation can be further improved if we use more π_k or ones with higher values of k. For example, we obtain $\pi \simeq 3.141\,592\,62$ if $k = 32, 64, 128, 256$ are used. Note that we are getting the same accuracy here as the evaluation of the Zus with polygons of 24 576 sides.

In a modern society, we need to deal with a lot more computations daily. Almost every event in science or technology requires quantification of the data involved. For example, before a jet aircraft can actually be manufactured, extensive computer simulations in different flight conditions must be performed to check whether there is a design flaw. This is not only necessary economically, but may help avoid loss of lives. A related use of computers is in the reconstruction of an unexpectred flight accident. This is extremely important in preventing the same accident from happening again. A more common example is found in the cars

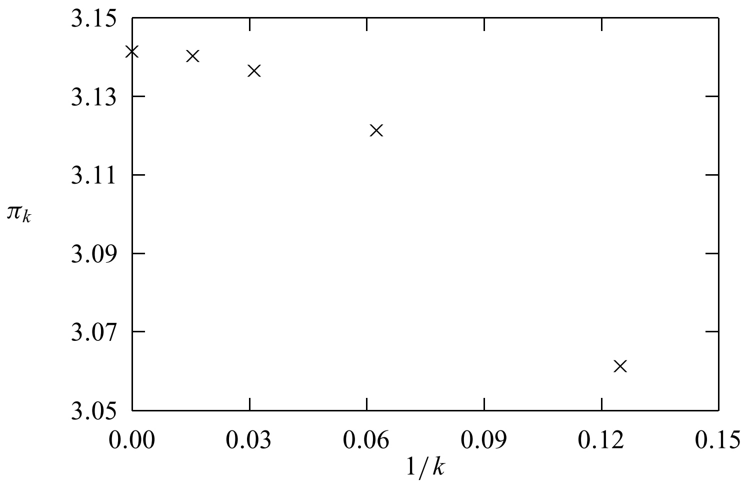


Fig. 1.3 The values of π_k, with $k = 8, 16, 32,$ and 64, plotted together with the extrapolated π_∞.

that we drive, which each have a computer that takes care of the brakes, steering control, and other critical components. Almost any electronic device that we use today is probably powered by a computer, for example, a digital thermometer, a DVD (digital video disc) player, a pacemaker, a digital clock, or a microwave oven. The list can go on and on. It is fair to say that sophisticated computations delivered by computers every moment have become part of our lives, permanently.

1.2 The emergence of modern computers

The advantage of having a reliable, robust calculating device was realized a long time ago. The early *abacus*, which was used for counting, was in existence with the Babylonians 4000 years ago. The Chinese abacus, which appeared at least 3000 years ago, was perhaps the first comprehensive calculating device that was actually used in performing addition, subtraction, multiplication, and division and was employed for several thousand years. A traditional Chinese abacus is made of a rectangular wooden frame and a bar going through the upper middle of the frame horizontally. See Fig. 1.4. There are thirteen evenly spaced vertical rods, each representing one decimal level. More rods were added to later versions. On each rod, there are seven beads that can be slid up and down with five of them held below the middle bar and two above. Zero on each rod is represented by the beads below the middle bar at the very bottom and the beads above at the very top. The numbers one to four are repsented by sliding one–four beads below the middle bar up and five is given be sliding one bead above down. The numbers six to nine are represented by one bead above the middle bar slid down and one–four beads below slid up. The first and last beads on each rod are never used or are only used cosmetically during a calculation. The Japanese abacus, which was modeled on the Chinese abacus, in fact has twenty-one rods, with only five beads

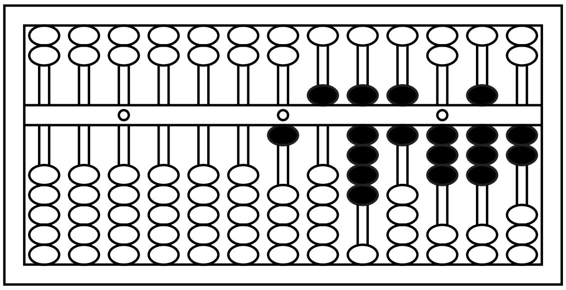


Fig. 1.4 A sketch of a Chinese abacus with the number 15 963.82 shown.

on each rod, one above and four below the middle bar. Dots are marked on the middle bar for the decimal point and for every four orders (ten thousands) of digits. The abacus had to be replaced by the slide rule or numerical tables when a calcualtion went beyond the four basic operations even though later versions of the Chinese abacus could also be used to evaluate square roots and cubic roots.

The *slide rule*, which is considered to be the next major advance in calculating devices, was introduced by the Englishmen Edmund Gunter and Reverend William Oughtred in the mid-seventeenth century based on the logarithmic table published by Scottish mathematician John Napier in a book in the early seventeenth century. Over the next several hundred years, the slide rule was improved and used worldwide to deliver the impressive computations needed, especially during the Industrial Revolution. At about the same time as the introduction of the slide rule, Frenchman Blaise Pascal invented the *mechanical calculating machine* with gears of different sizes. The mechanical calculating machine was enhanced and applied extensively in heavy-duty computing tasks before digital computers came into existence.

The concept of an all-purpose, automatic, and programmable computing machine was introduced by British mathematician and astronomer Charles Babbage in the early nineteenth century. After building part of a mechanical calculating machine that he called a *difference engine*, Babbage proposed constructing a computing machine, called an *analytical engine*, which could be programmed to perform any type of computation. Unfortunately, the technology at the time was not advanced enough to provide Babbage with the necessary machinery to realize his dream. In the late nineteenth century, Spanish engineer Leonardo Torres y Quevedo showed that it might be possible to construct the machine conceived earlier by Babbage using the electromechanical technology that had just been developed. However, he could not actually build the whole machine either, due to lack of funds. American engineer and inventor Herman Hollerith built the very first electromechanical *counting machine*, which was commisioned by the US federal government for sorting the population in the 1890 American census. Hollerith used the profit obtained from selling this machine to set up a company, the Tabulating Machine Company, the predecessor of IBM (International

Business Machines Corporation). These developments continued in the early twentieth century. In the 1930s, scientists and engineers at IBM built the first difference tabulator, while researchers at Bell Laboratories built the first relay calculator. These were among the very first electromechanical calculators built during that time.

The real beginning of the computer era came with the advent of electronic digital computers. John Vincent Atanasoff, a theoretical physicist at the Iowa State University at Ames, invented the electronic digital computer between 1937 and 1939. The history regarding Atanasoff's accomplishment is described in Mackintosh (1987), Burks and Burks (1988), and Mollenhoff (1988). Atanasoff introduced vacuum tubes (instead of the electromechanical devices used earlier by other people) as basic elements, a separated memory unit, and a scheme to keep the memory updated in his computer. With the assistance of Clifford E. Berry, a graduate assistant, Atanasoff built the very first electronic computer in 1939. Most computer history books have cited ENIAC (Electronic Numerical Integrator and Computer), built by John W. Mauchly and J. Presper Eckert with their colleagues at the Moore School of the University of Pennsylvania in 1945, as the first electronic computer. ENIAC, with a total mass of more than 30 tons, consisited of 18 000 vacuum tubes, 15 000 relays, and several hundred thousand resistors, capacitors, and inductors. It could complete about 5000 additions or 400 multiplications in one second. Some very impressive scientific computations were performed on ENIAC, including the study of nuclear fission with the liquid drop model by Metropolis and Frankel (1947). In the early 1950s, scientists at Los Alamos built another electronic digital computer, called MANIAC I (Mathematical Analyzer, Numerator, Integrator, and Computer), which was very similar to ENIAC. Many important numerical studies, including Monte Carlo simulation of classical liquids (Metropolis *et al.*, 1953), were completed on MANIAC I.

All these research-intensive activities accomplished in the 1950s showed that computation was no longer just a supporting tool for scientific research but rather an actual means of probing scientific problems and predicting new scientific phenomena. A new branch of science, *computational science*, was born. Since then, the field of scientific computing has developed and grown rapidly.

The computational power of new computers has been increasing exponentially. To be specific, the computing power of a single computer unit has doubled almost every 2 years in the last 50 years. This growth followed the observation of Gordon Moore, co-founder of Intel, that information stored on a given amount of silicon surface had doubled and would continue to do so in about every 2 years since the introduction of the silicon technology (nicknamed Moore's law). Computers with transistors replaced those with vacuum tubes in the late 1950s and early 1960s, and computers with very-large-scale integrated circuits were built in the 1970s. Microprocessors and vector processors were built in the mid-1970s to set the stage for personal computing and supercomputing. In the 1980s, microprocessor-based personal computers and workstations appeared. Now they have penetrated all aspects of our lives, as well as all scientific disciplines, because of their affordability and low maintenance cost. With technological breakthroughs in the RISC (Reduced Instruction Set Computer) architecture, cache memory, and multiple instruction units, the capacity of each microprocessor is now larger than that of a supercomputer 10 years ago. In the last few years, these fast microprocessors have been combined to form parallel or distributed computers, which can easily deliver a computing power of a few tens of gigaflops (10^9 floating-point operations per second). New computing paradigms such as the Grid were introduced to utilize computing resources on a global scale via the Internet (Foster and Kesselman, 2003; Abbas, 2004).

Teraflop (10^{12} floating-point operations per second) computers are now emerging. For example, Q, a newly installed computer at the Los Alamos National Laboratory, has a capacity of 30 teraflops. With the availability of teraflop computers, scientists can start unfolding the mysteries of the grand challenges, such as the dynamics of the global environment; the mechanism of DNA (deoxyribonucleic acid) sequencing; computer design of drugs to cope with deadly viruses; and computer simulation of future electronic materials, structures, and devices. Even though there are certain problems that computers cannot solve, as pointed out by Harel (2000), and hardware and software failures can be fatal, the human minds behind computers are nevertheless unlimited. Computers will never replace human beings in this regard and the quest for a better understanding of Nature will go on no matter how difficult the journey is. Computers will certainly help to make that journey more colorful and pleasant.

1.3 Computer algorithms and languages

Before we can use a computer to solve a specific problem, we must instruct the computer to follow certain procedures and to carry out the desired computational task. The process involves two steps. First, we need to transform the problem, typically in the form of an equation, into a set of logical steps that a computer can follow; second, we need to inform the computer to complete these logical steps.

Computer algorithms

The complete set of the logical steps for a specific computational problem is called a *computer* or *numerical algorithm*. Some popular numerical algorithms can be traced back over a 100 years. For example, Carl Friedrich Gauss (1866) published an article on the FFT (fast Fourier transform) algorithm (Goldstine, 1977, pp. 249–53). Of course, Gauss could not have envisioned having his algorithm realized on a computer.

Let us use a very simple and familiar example in physics to illustrate how a typical numerical algorithm is constructed. Assume that a particle of mass m is confined to move along the x axis under a force $f(x)$. If we describe its motion with Newton's equation, we have

$$f = ma = m\frac{dv}{dt}, \tag{1.4}$$

where a and v are the acceleration and velocity of the particle, respectively, and t is the time. If we divide the time into small, equal intervals $\tau = t_{i+1} - t_i$, we know from elementary physics that the velocity at time t_i is approximately given by the average velocity in the time interval $[t_i, t_{i+1}]$,

$$v_i \simeq \frac{x_{i+1} - x_i}{t_{i+1} - t_i} = \frac{x_{i+1} - x_i}{\tau}; \tag{1.5}$$

the corresponding acceleration is approximately given by the average acceleration in the same time interval,

$$a_i \simeq \frac{v_{i+1} - v_i}{t_{i+1} - t_i} = \frac{v_{i+1} - v_i}{\tau}, \tag{1.6}$$

as long as τ is small enough. The simplest algorithm for finding the position and velocity of the particle at time t_{i+1} from the corresponding quantities at time t_i is obtained after combining Eqs. (1.4), (1.5), and (1.6), and we have

$$x_{i+1} = x_i + \tau v_i, \tag{1.7}$$

$$v_{i+1} = v_i + \frac{\tau}{m} f_i, \tag{1.8}$$

where $f_i = f(x_i)$. If the initial position and velocity of the particle are given and the corresponding quantities at some later time are sought (the initial-value problem), we can obtain them recursively from the algorithm given in Eqs. (1.7) and (1.8). This algorithm is commonly known as the Euler method for the initial-value problem. This simple example illustrates how most algorithms are constructed. First, physical equations are transformed into discrete forms, namely, difference equations. Then the desired physical quantities or solutions of the equations at different variable points are given in a recursive manner with the quantities at a later point expressed in terms of the quantities from earlier points. In the above example, the position and velocity of the particle at t_{i+1} are given by the position and velocity at t_i, provided that the force at any position is explicitly given by a function of the position. Note that the above way of constructing an algorithm is not limited to one-dimensional or single-particle problems. In fact, we can immediately generalize this algorithm to two-dimensional and three-dimensional problems, or to the problems involving more than one particle, such as the motion of a projectile or a system of three charged particles. The generalized version of the above algorithm is

$$\mathbf{R}_{i+1} = \mathbf{R}_i + \tau \mathbf{V}_i, \tag{1.9}$$

$$\mathbf{V}_{i+1} = \mathbf{V}_i + \tau \mathbf{A}_i, \tag{1.10}$$

where $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n)$ is the position vector of all the n particles in the system; $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$ and $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$, with $\mathbf{a}_j = \mathbf{f}_j/m_j$ for $j = 1, 2, \ldots, n$, are the corresponding velocity and acceleration vectors, respectively.

From a theoretical point of view, the *Turing machine* is an abstract representation of a *universal computer* and also a device to autopsy any algorithm. The concept was introduced by Alan Turing (1936–7) with a description of the universal computer that consists of a read and write head and a tape with an infinite number of units of binaries (0 or 1). The machine is in a specified state for a given moment of operation and follows instructions prescribed by a finite table. A computer algorithm is a set of logical steps that can be achieved by the Turing machine. Logical steps that cannot be achieved by the Turing machine belong to the class of problems that are not solvable by computers. Some such unsolvable problems are discussed by Harel (2000).

The logical steps in an algorithm can be sequential, parallel, or iterative (implicit). How to utilize the properties of a given problem in constructing a fast and accurate algorithm is a very important issue in computational science. It is hoped that the examples discussed in this book will help students learn how to establish efficient and accurate algorithms as well as how to write clean and structured computer programs for most problems encountered in physics and related fields.

Computer languages

Computer programs are the means through which we communicate with computers. The very first computer program was written by Ada Byron, the Countess of Lovelace, and was intended for the analytical engine proposed by Babbage in the mid-1840s. To honor her achievement, an object-oriented programming language (Ada), initially developed by the US military, is named after her. A *computer program* or *code* is a collection of statements, typically written in a well-defined computer *programming language*. Programming languages can be divided into two major categories: low-level languages designed to work with the given hardware, and high-level languages that are not related to any specific hardware.

Simple machine languages and assembly languages were the only ones available before the development of high-level languages. A machine language is typically in binary form and is designed to work with the unique hardware of a computer. For example, a statement, such as adding or multiplying two integers, is represented by one or several binary strings that the computer can recognize and follow. This is very efficient from computer's point of view, but extremely labor-intensive from that of a programmer. To remember all the binary strings for all the statements is a nontrivial task and to debug a program in binaries is a formidable task. Soon after the invention of the digital computer, assembly languages were introduced to increase the efficiency of programming and debugging. They are more advanced than machine languages because they have adopted symbolic addresses. But they are still related to a certain architecture and wiring of the system. A translating device called an assembler is needed to convert an assembly code into a native machine code before a computer can recognize the instructions. Machine languages and assembly languages do not have portability; a program written for one kind of computers could never be used on others.

The solution to such a problem is clearly desirable. We need high-level languages that are not associated with the unique hardware of a computer and that can work on all computers. Ideal programming languages would be those that are very concise but also close to the logic of human languages. Many high-level programming languages are now available, and the choice of using a specific programming language on a given computer is more or less a matter of personal taste. Most high-level languages function similarly. However, for a researcher who is working at the cutting edge of scientific computing, the speed and capacity of a computing system, including the efficiency of the language involved, become critical.

A modern computer program conveys the tasks of an algorithm for a computational problem to a computer. The program cannot be executed by the computer before it is translated into the native machine code. A translator, a program called a *compiler*, is used to translate (or compile) the program to produce an executable file in binaries. Most compilers also have an option to produce an objective file first and then link it with other objective files and library routines to produce a combined executable file. The compiler is able to detect most errors introduced during programming, that is, the process of writing a program in a high-level language. After running the executable program, the computer will output the result as instructed.

The newest programming language that has made a major impact in the last few years is Java, an object-oriented, interpreted language. The strength of Java lies in its ability to work with web browsers, its comprehensive GUI (graphical user interface), and its built-in security and network support. Java is a truly universal language because it is fully platform-independent: "write once, run everywhere" is the motto that Sun Microsystems uses to qualify all the features in Java. Both the source code and the compiled code can run on any computer that has Java installed with exactly the same result. The Java compiler converts the source code (`file.java`) into a bytecode (`file.class`), which contains instructions in fixed-length byte strings and can be interpreted/executed on any computer under the Java interpreter, called JVM (Java Virtual Machine).

There are many advantages in Java, and its speed in scientific programming has been steadily increased over the last few years. At the moment, a carefully written Java program, combined with static analysis, just-in-time compiling, and instruction-level optimization, can deliver nearly the same raw speed as the incumbent C or Fortran (Boisvert *et al.*, 2001).

Let us use the algorithm that we highlighted earlier for a particle moving along the x axis to show how an algorithm is translated into a program in Java. For simplicity, the force is taken to be an elastic force $f(x) = -kx$, where k is the elastic constant. We will also use $m = k = 1$ for convenience. The following Java program is an implementation of the algorithm given in Eqs. (1.7) and (1.8); each statement in the program is almost self-explanatory.

```java
// An example of studying the motion of a particle in
// one dimension under an elastic force.

import java.lang.*;
public class Motion {
  static final int n = 100000, j = 500;
  public static void main(String argv[]) {
    double x[] = new double[n+1];
    double v[] = new double[n+1];

 // Assign time step and initial position and velocity
    double dt = 2*Math.PI/n;
    x[0] = 0;
    v[0] = 1;

 // Calculate other position and velocity recursively
    for (int i=0; i<n; ++i) {
      x[i+1] = x[i]+v[i]*dt;
      v[i+1] = v[i]-x[i]*dt;
    }

 // Output the result in every j time steps
    double t = 0;
    double jdt = j*dt;
    for (int i=0; i<=n; i+=j) {
      System.out.println(t +" " + x[i] + " " + v[i]);
      t += jdt;
    }
  }
}
```

The above program contains some key elements of a typical Java program. The first line imports the Java language package that contains the major features and mathematical functions in the language. The program starts with a public class declaration with a main method under this class. Arrays are treated as objects. For a good discussion on the Java programming language, see van der Linden (2004), and for its relevance in scientific computing, see Davies (1999).

The file name of a program in Java must be the same as that of the only public class in the code. In the above example, the file name is therefore `Motion.java`. After the program is compiled with the command `javac Motion.java`, a bytecode is created under the file name `Motion.class`. The bytecode can then be interpreted/executed with the command `java Motion`. Some of the newest compilers create an executable file, native machine code in a binary form to speed

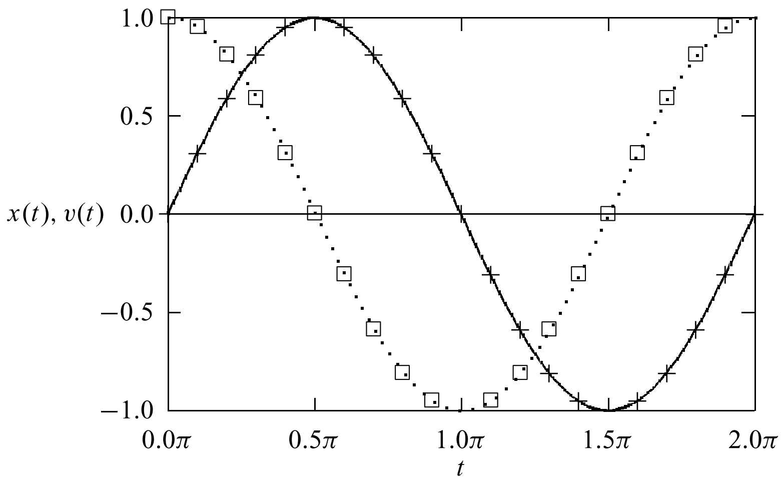


Fig. 1.5 The time-dependent position (+) and velocity (□) of the particle generated from the program `Motion.java` and the corresponding analytical results (solid and dotted lines, respectively).

up the computation. The executable file is then machine-dependent. Figure 1.5 is a plot of the output from the above program together with the analytical result. The numerical result generated from the program agrees well with the analytical result. Because the algorithm we have used here is a very simple one, we have to use a very small time step in order to obtain the result with a reasonable accuracy. In Chapter 4, we will introduce and discuss more efficient algorithms for solving differential equations. With these more efficient algorithms, we can usually reach the same accuracy with a very small number of mesh points in one period of the motion, for example, 100 points instead of 100 000.

There are other high-level programming languages that are used in scientific computing. The longest-running candidate is Fortran (*For*mula *tran*slation), which was introduced in 1957 as one of the earliest high-level languages and is still one of the primary languages in computational science. Of course, the Fortran language has evolved from its very early version, known as Fortran 66, to Fortran 77, which has been the most popular language for scientific computing in the last 30 years. For a modern discussion on the Fortran language and its applications, see Edgar (1992). The newest version of Fortran, known as Fortran 90, has absorbed many important features for parallel computing. Fortran 90 has many extensions over the standard Fortran 77. Most of these extensions are established based on the extensions already adopted by computer manufacturers to enhance their computer performance. Efficient compilers with a full implementation of Fortran 90 are available for all major computer systems. A complete discussion on Fortran 90 can be found in Brainerd, Goldberg, and Adams (1996). Two new variants of Fortran 90 have now been introduced, Fortran 95 and Fortran 2000 (Metcalf, Reid, and Cohen, 2004), which are still to be ratified. In the last 15 years, there have been some other new developments in parallel and distributed computing with new protocols and environments under various software packages, which we will leave to the readers to discover and explore.

The other popular programming language for scientific computing is the C programming language. Most system programmers and software developers prefer to use C in developing system and application software because of its high flexibility (Kernighan and Ritchie, 1988). For example, the Unix operating system (Kernighan and Pike, 1984) now used on almost all workstations and supercomputers was initially written in C.

In the last 50 years of computer history, many programming languages have appeared and then disappeared for one reason or another. Several languages have made significant impact on how computing tasks are achieved today. Examples include Cobol, Algol, Pascal, and Ada. Another object-oriented language is C++, which is based on C and contains valuable extensions in several important aspects (Stroustrup, 2000). At the moment, C++ has perhaps been the most popular language for game developers.

Today, Fortran is still by far the dominant programming language in scientific computing for two very important reasons: Many application packages are available in Fortran, and the structure of Fortran is extremely powerful in dealing with equations. However, the potential of Java and especially its ability to work with the Internet through applets and servlets has positioned it ahead of any other language. For example, Java is clearly the front runner for the newest senario in high-performance computing of constructing global networks of computers through the concept of the Grid (Foster and Kesselman, 2003; Abbas, 2004). More scientisits, especially those emerging from colleges or graduate schools, are expected to use Java as their first, primary programming language. So the choice of using Java as the instructional langage here has been made with much thought. Readers who are familiar with any other high-level programming language should have no difficulty in understanding the logical structures and contents of the example programs in the book. The reason is very simple. The logic in all high-level languages is similar to the logic of our own languages. All the simple programs written in high-level languages should be self-explanatory, as long as enough commentary lines are provided.

There have been some very exciting new developments in Java. The new version of Java, Java 2 or JDK (Java Development Kit) 1.2, has introduced `BigInteger` and `BigDecimal` classes that allow us to perform computations with integers and floating-point numbers to any accuracy. This is an important feature and has opened the door to better scientific programming styles and more durable codes. JDK 1.4 has also implemented `strictfp` in order to relax the restricition in regular floating-point data. Many existing classes have also been improved to provide better performance or to eliminate the instability of some earlier classes. Many vendors are developing new compilers for Java to implement just-in-time compiling, static analysis, and instruction-level optimization to improve the running speed of Java. Some of these new compilers are very successful in comparison with the traditional Fortran and C compilers on key problems in scientific computing. Many new applications in science are being developed in

Java worldwide. It is the purpose of this book to provide students with building blocks for developing practical skills in scientific computing, which has become a critical pillar in fundamental research, knowledge development, and emerging technology.

Exercises

1.1 The value of π can be estimated from the calculations of the side lengths of regular polygons inscribing a circle. In general,

$$\pi_k = \pi_\infty + \frac{c_1}{k} + \frac{c_2}{k^2} + \frac{c_3}{k^3} + \cdots ,$$

where π_k is the ratio of the perimeter to the diameter of a regular k-sided polygon. Determine the approximate value of $\pi \simeq \pi_\infty$ from $\pi_8 = 3.061\,467$, $\pi_{16} = 3.121\,445$, $\pi_{32} = 3.136\,548$, and $\pi_{64} = 3.140\,331$ of the inscribing polygons. Which c_i is most significant and why? What happens if we use the values from the polygons circumscribing the circle, for example, $\pi_8 = 3.313\,708$, $\pi_{16} = 3.182\,598$, $\pi_{32} = 3.151\,725$, and $\pi_{64} = 3.144\,118$?

1.2 Show that the Euler method for Newton's equation in Section 1.3 is accurate up to a term on the order of $(t_{i+1} - t_i)^2$. Discuss how to improve its accuracy.

1.3 An efficient program should always avoid unnecessary operations, such as the calculation of any constant or repeated access to arrays, subprograms, library routines, or to other objects inside a loop. The problem becomes worse if the loop is long or inside other loops. Examine the example programs in this chapter and Chapters 2 and 3 and improve the efficiency of these programs if possible.

1.4 Several mathematical constants are used very frequently in science, such as π, e, and the Euler constant $\gamma = \lim_{n\to\infty} \left(\sum_{k=1}^{n} k^{-1} - \ln n\right)$. Find three ways of creating each of π, e, and γ in a code. After considering language specifications, numerical accuracy, and efficiency, which way of creating each of them is most appropriate? If we need to use such a constant many times in a program, should the constant be created once and stored under a variable to be used over and over again, or should it be created/accessed every time it is needed?

1.5 Translate the Java program in Section 1.3 for a particle moving in one dimension into another programming language.

1.6 Modify the program given in Section 1.3 to study a particle, under a uniform gravitational field vertically and a resistive force $\mathbf{f}_\mathrm{r} = -\kappa v \mathbf{v}$, where v ($\mathbf{v}$) is the speed (velocity) of the particle and κ is a positive parameter. Analyze the height dependence of the speed of a raindrop with different m/κ, where m is the mass of the raindrop, taken to be a constant for simplicity. Plot the terminal speed of the raindrop against m/κ, and compare it with the result of free falling.

1.7 The dynamics of a comet is governed by the gravitational force between the comet and the Sun, $\mathbf{f} = -GMm\mathbf{r}/r^3$, where $G = 6.67 \times 10^{-11}$ N m^2/kg^2 is the gravitational constant, $M = 1.99 \times 10^{30}$ kg is the mass of the Sun, m is the mass of the comet, $\mathbf{r}$ is the position vector of the comet measured from the Sun, and r is the magnitude of $\mathbf{r}$. Write a program to study the motion of Halley's comet that has an aphelion (the farthest point from the Sun) distance of 5.28×10^{12} m and an aphelion velocity of 9.12×10^2 m/s. What are the proper choices of the time and length units for the problem? Discuss the error generated by the program in each period of Halley's comet.

1.8 People have made motorcycle jumps over long distances. We can build a model to study these jumps. The air resistance on a moving object is given by $\mathbf{f}_\mathrm{r} = -cA\rho v\mathbf{v}/2$, where v ($\mathbf{v}$) is the speed (velocity) and A is cross section of the moving object, ρ is the density of the air, and c is a coefficient on the order of 1 for all other uncounted factors. Assuming that the cross section is $A = 0.93$ m^2, the maximum taking-off speed of the motorcycle is 67 m/s, the air density is $\rho = 1.2$ kg/m^3, the combined mass of the motorcycle and the person is 250 kg, and the coefficient c is 1, find the tilting angle of the taking-off ramp that can produce the longest range.

1.9 One way to calculate π is by randomly throwing a dart into the unit square defined by $x \in [0, 1]$ and $y \in [0, 1]$ in the xy plane. The chance of the dart landing inside the unit circle centered at the origin of the coordinates is $\pi/4$, from the comparison of the areas of one quarter of the circle and the unit square. Write a program to calculate π in such a manner. Use the random-number generator provided in the programming language selected or the one given in Chapter 2.

1.10 Object-oriented languages are convenient for creating applications that are icon-driven. Write a Java application that simulates a Chinese abacus. Test your application by performing the four basic math operations (addition, subtraction, multiplication, and division) on your abacus.

Chapter 2
Approximation of a function

This chapter and the next examine the most commonly used methods in computational science. Here we concentrate on some basic aspects associated with numerical approximation of a function, interpolation, least-squares and spline approximations of a curve, and numerical representations of uniform and other distribution functions. We are only going to give an introductory description of these topics here as a preparation for other chapters and many of the issues will be revisited in a greater depth later. Note that some of the material covered here would require much more space if discussed thoroughly. For example, complete coverage of the issues involved in creating good random-number generators could form a separate book. Therefore, we only focus on the basics of the topics here.

2.1 Interpolation

In numerical analysis, the results obtained from computations are always approximations of the desired quantities and in most cases are within some uncertainties. This is similar to experimental observations in physics. Every single physical quantity measured carries some experimental error. We constantly encounter situations in which we need to interpolate a set of discrete data points or to fit them to an adjustable curve. It is extremely important for a physicist to be able to draw conclusions based on the information available and to generalize the knowledge gained in order to predict new phenomena.

Interpolation is needed when we want to infer some local information from a set of incomplete or discrete data. Overall approximation or fitting is needed when we want to know the general or global behavior of the data. For example, if the speed of a baseball is measured and recorded every $1/100$ of a second, we can then estimate the speed of the baseball at any moment by interpolating the recorded data around that time. If we want to know the overall trajectory, then we need to fit the data to a curve. In this section, we will discuss some very basic interpolation schemes and illustrate how to use them in physics.

Linear interpolation

Consider a discrete data set given from a discrete function $f_i = f(x_i)$ with $i = 0, 1, \ldots, n$. The simplest way to obtain the approximation of $f(x)$ for $x \in [x_i, x_{i+1}]$ is to construct a straight line between x_i and x_{i+1}. Then $f(x)$ is given by

$$f(x) = f_i + \frac{x - x_i}{x_{i+1} - x_i}(f_{i+1} - f_i) + \Delta f(x), \tag{2.1}$$

which of course is not accurate enough in most cases but serves as a good start in understanding other interpolation schemes. In fact, any value of $f(x)$ in the region $[x_i, x_{i+1}]$ is equal to the sum of the linear interpolation in the above equation and a quadratic contribution that has a unique curvature and is equal to zero at x_i and x_{i+1}. This means that the error $\Delta f(x)$ in the linear interpolation is given by

$$\Delta f(x) = \frac{\gamma}{2}(x - x_i)(x - x_{i+1}), \tag{2.2}$$

with γ being a parameter determined by the specific form of $f(x)$. If we draw a quadratic curve passing through $f(x_i)$, $f(a)$, and $f(x_{i+1})$, we can show that the quadrature

$$\gamma = f''(a), \tag{2.3}$$

with $a \in [x_i, x_{i+1}]$, as long as $f(x)$ is a smooth function in the region $[x_i, x_{i+1}]$; namely, the kth-order derivative $f^{(k)}(x)$ exists for any k. This is the result of the Taylor expansion of $f(x)$ around $x = a$ with the derivatives $f^{(k)}(a) = 0$ for $k > 2$. The maximum error in the linear interpolation of Eq. (2.1) is then bounded by

$$|\Delta f(x)| \le \frac{\gamma_1}{8}(x_{i+1} - x_i)^2, \tag{2.4}$$

where $\gamma_1 = \max[|f''(x)|]$ with $x \in [x_i, x_{i+1}]$. The upper bound of the error in Eq. (2.4) is obtained from Eq. (2.2) with γ replaced by γ_1 and x solved from $d\Delta f(x)/dx = 0$. The accuracy of the linear interpolation can be improved by reducing the interval $h_i = x_{i+1} - x_i$. However, this is not always practical.

Let us take $f(x) = \sin x$ as an illustrative example here. Assuming that $x_i = \pi/4$ and $x_{i+1} = \pi/2$, we have $f_i = 0.707$ and $f_{i+1} = 1.000$. If we use the linear interpolation scheme to find the approximate value of $f(x)$ at $x = 3\pi/8$, we have the interpolated value $f(3\pi/8) \simeq 0.854$ from Eq. (2.1). We know, of course, that $f(3\pi/8) = \sin(3\pi/8) = 0.924$. The actual difference is $|\Delta f(x)| = 0.070$, which is smaller than the maximum error estimated with Eq. (2.4), 0.077.

The above example is a very simple one, showing how most interpolation schemes work. A continuous curve (a straight line in the above example) is constructed from the given discrete set of data and then the interpolated value is read off from the curve. The more points there are, the higher the order of the curve can be. For example, we can construct a quadratic curve from three data points and a cubic curve from four data points. One way to achieve higher-order interpolation is through the Lagrange interpolation scheme, which is a generalization of the linear interpolation that we have just discussed.

The Lagrange interpolation

Let us first make an observation about the linear interpolation discussed in the preceding subsection. The interpolated function actually passes through the two points used for the interpolation. Now if we use three points for the interpolation, we can always construct a quadratic function that passes through all the three points. The error is now given by a term on the order of h^3, where h is the larger interval between any two nearest points, because an x^3 term could be added to modify the curve to pass through the function point if it were actually known. In order to obtain the generalized interpolation formula passing through $n + 1$ data points, we rewrite the linear interpolation of Eq. (2.1) in a symmetric form with

$$\begin{aligned} f(x) &= \frac{x - x_{i+1}}{x_i - x_{i+1}} f_i + \frac{x - x_i}{x_{i+1} - x_i} f_{i+1} + \Delta f(x) \\ &= \sum_{j=i}^{i+1} f_j p_{1j}(x) + \Delta f(x), \end{aligned} \tag{2.5}$$

where

$$p_{1j}(x) = \frac{x - x_k}{x_j - x_k}, \tag{2.6}$$

with $k \neq j$. Now we can easily generalize the expression to an nth-order curve that passes through all the $n + 1$ data points,

$$f(x) = \sum_{j=0}^{n} f_j p_{nj}(x) + \Delta f(x), \tag{2.7}$$

where $p_{nj}(x)$ is given by

$$p_{nj}(x) = \prod_{k \neq j}^{n} \frac{x - x_k}{x_j - x_k}. \tag{2.8}$$

In other words, $\Delta f(x_j) = 0$ at all the data points. Following a similar argument to that for linear interpolation in terms of the Taylor expansion, we can show that the error in the nth-order Lagrange interpolation is given by

$$\Delta f(x) = \frac{\gamma}{(n+1)!}(x - x_0)(x - x_1) \cdots (x - x_n), \tag{2.9}$$

where

$$\gamma = f^{(n+1)}(a), \tag{2.10}$$

with $a \in [x_0, x_n]$. Note that $f(a)$ is a point passed through by the $(n + 1)$th-order curve that also passes through all the $f(x_i)$ with $i = 0, 1, \ldots, n$. Therefore, the maximum error is bounded by

$$|\Delta f(x)| \leq \frac{\gamma_n}{4(n+1)} h^{n+1}, \tag{2.11}$$

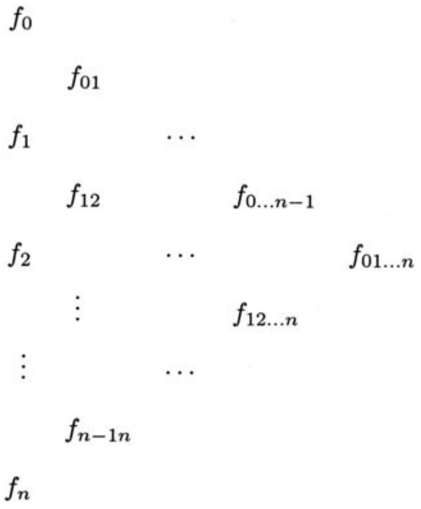


Fig. 2.1 The hierarchy in the Aitken scheme for $n+1$ data points.

where $\gamma_n = \max[|f^{(n+1)}(x)|]$ with $x \in [x_0, x_n]$ and h is the largest $h_i = x_{i+1} - x_i$. The above upper bound can be obtained by replacing γ with γ_n in Eq. (2.9) and then maximizing the pairs $(x - x_0)(x - x_n)$, $(x - x_1)(x - x_{n-1})$, $\ldots$, and $(x - x_{(n-1)/2})(x - x_{(n+1)/2})$ individually for an even $n + 1$. For an odd $n + 1$, we can choose the maximum value nh for the $x - x_i$ that is not paired. Equation (2.7) can be rewritten into a power series

$$f(x) = \sum_{k=0}^{n} a_k x^k + \Delta f(x), \tag{2.12}$$

with a_k given by expanding $p_{nj}(x)$ in Eq. (2.7). Note that the generalized form reduces to the linear case if $n = 1$.

The Aitken method

One way to achieve the Lagrange interpolation efficiently is by performing a sequence of linear interpolations. This scheme was first developed by Aitken (1932). We can first work out n linear interpolations with each constructed from a neighboring pair of the $n + 1$ data points. Then we can use these n interpolated data points to achieve another level of $n - 1$ linear interpolations with the next neighboring points of x_i. We repeat this process until we obtain the final result after n levels of consecutive linear interpolations. We can summarize the scheme in the following equation:

$$f_{i\ldots j} = \frac{x - x_j}{x_i - x_j} f_{i\ldots j-1} + \frac{x - x_i}{x_j - x_i} f_{i+1\ldots j}, \tag{2.13}$$

with $f_i = f(x_i)$ to start. If we want to obtain $f(x)$ from a given set f_i for $i = 0, 1, \ldots, n$, we can carry out n levels of consecutive linear interpolations as shown in Fig. 2.1, in which every column is constructed from the previous column by

Table 2.1. *Result of the example with the Aitken method*

x_i	f_i	f_{ij}	f_{ijk}	f_{ijkl}	f_{ijklm}
0.0	1.000 000				
		0.889 246			
0.5	0.938 470		0.808 792		
		0.799 852		0.807 272	
1.0	0.765 198		0.806 260		0.807 473
		0.815 872		0.807 717	
1.5	0.511 828		0.811 725		
		0.857 352			
2.0	0.223 891				

linear interpolations of the adjacent values. For example,

$$f_{012} = \frac{x - x_2}{x_0 - x_2} f_{01} + \frac{x - x_0}{x_2 - x_0} f_{12} \tag{2.14}$$

and

$$f_{01234} = \frac{x - x_4}{x_0 - x_4} f_{0123} + \frac{x - x_0}{x_4 - x_0} f_{1234}. \tag{2.15}$$

It can be shown that the consecutive linear interpolations outlined in Fig. 2.1 recover the standard Lagrange interpolation. The Aitken method also provides a way of estimating the error of the Lagrange interpolation. If we use the five-point case, that is, $n + 1 = 5$, as an illustrative example, the error in the Lagrange interpolation scheme is roughly given by

$$\Delta f(x) \approx \frac{|f_{01234} - f_{0123}| + |f_{01234} - f_{1234}|}{2}, \tag{2.16}$$

where the differences are taken from the last two columns of the hierarchy.

Let us consider the evaluation of $f(0.9)$, from the given set $f(0.0) = 1.000\,000$, $f(0.5) = 0.938\,470$, $f(1.0) = 0.765\,198$, $f(1.5) = 0.511\,828$, and $f(2.0) = 0.223\,891$, as an actual numerical example. These are the values of the zeroth-order Bessel function of the first kind, $J_0(x)$. All the consecutive linear interpolations of the data with the Aitken method are shown in Table 2.1.

The error estimated from the differences of the last two columns of the data in the table is

$$\Delta f(x) \approx \frac{|0.807\,473 - 0.807\,273| + |0.807\,473 - 0.807\,717|}{2} \simeq 2 \times 10^{-4}.$$

The exact result of $f(0.9)$ is 0.807 524. The error in the interpolated value is $|0.807\,473 - 0.807\,524| \simeq 5 \times 10^{-5}$, which is a little smaller than the estimated error from the differences of the last two columns in Table 2.1. The following program is an implementation of the Aitken method for the Lagrange interpolation, using the given example of the Bessel function as a test.

```
// An example of extracting an approximate function
// value via the Lagrange interpolation scheme.

import java.lang.*;
public class Lagrange {
  public static void main(String argv[]) {
    double xi[] = {0, 0.5, 1, 1.5, 2};
    double fi[] = {1, 0.938470, 0.765198, 0.511828,
      0.223891};
    double x = 0.9;
    double f = aitken(x, xi, fi);
    System.out.println("Interpolated value: " + f);
  }

// Method to carry out the Aitken recursions.

  public static double aitken(double x, double xi[],
    double fi[]) {
    int n = xi.length-1;
    double ft[] = (double[]) fi.clone();
    for (int i=0; i<n; ++i) {
      for (int j=0; j<n-i; ++j) {
        ft[j] = (x-xi[j])/(xi[i+j+1]-xi[j])*ft[j+1]
              +(x-xi[i+j+1])/(xi[j]-xi[i+j+1])*ft[j];
      }
    }
    return ft[0];
  }
}
```

After running the above program, we obtain the expected result, $f(0.9) \simeq 0.807\,473$. Even though we have a quite accurate result here, the interpolation can be influenced significantly by the rounding error in some cases if the Aitken procedure is carried out directly. This is due to the change in the interpolated value being quite small compared to the actual value of the function during each step of the consecutive linear interpolations. When the number of data points involved becomes large, the rounding error starts to accumulate. Is there a better way to achieve the interpolation?

A better way is to construct an indirect scheme that improves the interpolated value at every step by updating the differences of the interpolated values from the adjacent columns, that is, by improving the corrections of the interpolated values over the preceding column rather than the interpolated values themselves. The effect of the rounding error is then minimized. This procedure is accomplished with the *up-and-down method*, which utilizes the upward and downward corrections

$$\Delta_{ij}^{+} = f_{j\ldots j+i} - f_{j+1\ldots j+i}, \tag{2.17}$$

$$\Delta_{ij}^{-} = f_{j\ldots j+i} - f_{j\ldots j+i-1}, \tag{2.18}$$

Fig. 2.2 The hierarchy for both Δ^+_{ij} and Δ^-_{ij} in the upward and downward correction method for $n + 1$ data points.

$$\begin{array}{ccccc}
\Delta_{00} & & & & \\
 & \Delta_{10} & & & \\
\Delta_{01} & & \cdots & & \\
 & \Delta_{11} & & \Delta_{n-10} & \\
\Delta_{02} & & \cdots & & \Delta_{n0} \\
 & \vdots & & \Delta_{n-11} & \\
\vdots & & \cdots & & \\
 & \Delta_{1n-1} & & & \\
\Delta_{0n} & & & &
\end{array}$$

defined at each step from the differences between two adjacent columns. Here Δ^-_{ij} is the downward (going down along the triangle in Fig. 2.1) correction and Δ^+_{ij} is the upward (going up along the triangle in Fig. 2.1) correction. The index i here is for the level of correction and j is for the element in each level. The hierarchy for both Δ^+_{ij} and Δ^-_{ij} is show in Fig. 2.2. It can be shown from the definitions of Δ^+_{ij} and Δ^-_{ij} that they satisfy the following recursion relations:

$$\Delta^+_{ij} = \frac{x_{i+j} - x}{x_{i+j} - x_j}(\Delta^+_{i-1j} - \Delta^-_{i-1j+1}), \tag{2.19}$$

$$\Delta^-_{ij} = \frac{x_j - x}{x_{i+j} - x_j}(\Delta^+_{i-1j} - \Delta^-_{i-1j+1}), \tag{2.20}$$

with the starting column $\Delta^\pm_{0j} = f_j$. Here x_j is chosen as the data point closest to x. In general, we use the upward correction as the first correction if $x < x_j$. Otherwise, the downward correction is used. Then we alternate the downward and upward corrections in the steps followed until the final result is reached. If the upper (lower) boundary of the triangle in Fig. 2.2 is reached during the process, only downward (upward) corrections can be used afterward.

We can use the numerical values of the Bessel function in Table 2.1 to illustrate the method. Assume that we are still calculating $f(x)$ with $x = 0.9$. It is easy to see that the starting point should be $x_j = 1.0$, because it is closest to $x = 0.9$. So the zeroth-order approximation of the interpolated data is $f(x) \approx f(1.0)$. The first correction to $f(x)$ is then Δ^+_{11}. In the next step, we alternate the direction of the correction and use the downward correction, Δ^-_{21} in this example, to improve $f(x)$ further. We can continue the procedure with another upward correction and another downward correction to reach the final result. We can write a simple program to accomplish what we have just described. It is a good practice to write a method, function, or subroutine, depending on the particular language used, with x, x_i, and f_i, for $i = 0, 1, \ldots, n$, being the input and $f(x)$ being the output. The method can then be used for any interpolation task. Here is an implementation of the up-and-down method in Java.

```
// Method to complete the interpolation via upward and
// downward corrections.
  public static double upwardDownward(double x,
    double xi[], double fi[]) {
    int n = xi.length-1;
    double dp[][] = new double[n+1][];
    double dm[][] = new double[n+1][];
 // Assign the 1st columns of the corrections
    dp[0] = (double[]) fi.clone();
    dm[0] = (double[]) fi.clone();
 // Find the closest point to x
    int j0 = 0, k0 = 0;
    double dx = x-xi[0];
    for (int j=1; j<=n; ++j) {
      double dx1 = x-xi[j];
      if (Math.abs(dx1)<Math.abs(dx)) {
        j0 = j;
        dx = dx1;
      }
    }
    k0 = j0;
 // Evaluate the rest of the corrections recursively
    for (int i=1; i<=n; ++i) {
      dp[i] = new double[n-i+1];
      dm[i] = new double[n-i+1];
      for (int j=0; j<n-i+1; ++j) {
        double d = dp[i-1][j]-dm[i-1][j+1];
        d /= xi[i+j]-xi[j];
        dp[i][j] = d*(xi[i+j]-x);
        dm[i][j] = d*(xi[j]-x);
      }
    }
 // Update the interpolation with the corrections
    double f = fi[j0];
    for (int i=1; i<=n; ++i) {
      if (((dx<0)||(k0==n)) && (j0!=0)) {
        j0--;
        f += dp[i][j0];
        dx = -dx;
      }
      else {
        f += dm[i][j0];
        dx = -dx;
        k0++;
      }
    }
     return f;
  }
```

We can replace the Aitken method in the earlier example program with this method. The numerical result, with the input data from Table 2.1, is exactly the same as the earlier result, $f(0.9) = 0.807\,473$, as expected.

2.2 Least-squares approximation

As we have pointed out, interpolation is mainly used to find the local approximation of a given discrete set of data. In many situations in physics we need to know the global behavior of a set of data in order to understand the trend in a specific measurement or observation. A typical example is a polynomial fit to a set of experimental data with error bars.

The most common approximation scheme is based on the least squares of the differences between the approximation $p_m(x)$ and the data $f(x)$. If $f(x)$ is the data function to be approximated in the region $[a, b]$ and the approximation is an mth-order polynomial

$$p_m(x) = \sum_{k=0}^{m} a_k x^k, \tag{2.21}$$

we can construct a function of a_k for $k = 0, 1, \ldots, m$ as

$$\chi^2[a_k] = \int_a^b [\, p_m(x) - f(x)]^2 \, dx \tag{2.22}$$

for the continuous data function $f(x)$, and

$$\chi^2[a_k] = \sum_{i=0}^{n} [\, p_m(x_i) - f(x_i)]^2 \tag{2.23}$$

for the discrete data function $f(x_i)$ with $i = 0, 1, \ldots, n$. Here we have used a generic variable a_k inside a square bracket for a quantity that is a function of a set of independent variables $a_0, a_1, \ldots, a_m$. This notation will be used throughout this book. Here χ^2 is the conventional notation for the summation of the squares of the deviations.

The least-squares approximation is obtained with $\chi^2[a_k]$ minimized with respect to all the $m + 1$ coefficients through

$$\frac{\partial \chi^2[a_k]}{\partial a_l} = 0, \tag{2.24}$$

for $l = 0, 1, 2, \ldots, m$. The task left is to solve this set of $m + 1$ linear equations to obtain all the a_l. This general problem of solving a linear equation set will be discussed in detail in Chapter 5. Here we consider a special case with $m = 1$, that is, the linear fit. Then we have

$$p_1(x) = a_0 + a_1 x, \tag{2.25}$$

with

$$\chi^2[a_k] = \sum_{i=0}^{n} (a_0 + a_1 x_i - f_i)^2 . \tag{2.26}$$

From Eq. (2.24), we obtain

$$(n + 1)a_0 + c_1 a_1 - c_3 = 0, \tag{2.27}$$

$$c_1 a_0 + c_2 a_1 - c_4 = 0, \tag{2.28}$$

where $c_1 = \sum_{i=0}^{n} x_i$, $c_2 = \sum_{i=0}^{n} x_i^2$, $c_3 = \sum_{i=0}^{n} f_i$, and $c_4 = \sum_{i=0}^{n} x_i f_i$. Solving these two equations together, we obtain

$$a_0 = \frac{c_1 c_4 - c_2 c_3}{c_1^2 - (n+1)c_2}, \tag{2.29}$$

$$a_1 = \frac{c_1 c_3 - (n+1)c_4}{c_1^2 - (n+1)c_2}. \tag{2.30}$$

We will see an example of implementing this linear approximation in the analysis of the data from the Millikan experiment in the next section. Note that this approach becomes very involved when m becomes large.

Here we change the strategy and tackle the problem with orthogonal polynomials. In principle, we can express the polynomial $p_m(x)$ in terms of a set of orthogonal polynomials with

$$p_m(x) = \sum_{k=0}^{m} \alpha_k u_k(x), \tag{2.31}$$

where $u_k(x)$ is a set of real orthogonal polynomials that satisfy

$$\int_a^b u_k(x) w(x) u_l(x)\, dx = \langle u_k | u_l \rangle = \delta_{kl} \mathcal{N}_k, \tag{2.32}$$

with $w(x)$ being the weight whose form depends on the specific set of orthogonal polynomials. Here δ_{kl} is the Kronecker δ function, which is 1 for $k = l$ and 0 for $k \neq l$, and $\mathcal{N}_k$ is a normalization constant. The coefficients α_k can be formally related to a_j by a matrix transformation, and are determined with $\chi^2[\alpha_k]$ minimized. Note that $\chi^2[\alpha_k]$ is the same quantity defined in Eq. (2.22) or Eq. (2.23) with $p_m(x)$ from Eq. (2.31). If we want the polynomials to be orthonormal, we can simply divide $u_k(x)$ by $\sqrt{\mathcal{N}_k}$. We will also use the notation in the above equation for the discrete case with

$$\langle u_k | u_l \rangle = \sum_{i=0}^{n} u_k(x_i) w(x_i) u_l(x_i) = \delta_{kl} \mathcal{N}_k. \tag{2.33}$$

The orthogonal polynomials can be generated with the following recursion:

$$u_{k+1}(x) = (x - g_k) u_k(x) - h_k u_{k-1}(x), \tag{2.34}$$

where the coefficients g_k and h_k are given by

$$g_k = \frac{\langle x u_k | u_k \rangle}{\langle u_k | u_k \rangle}, \tag{2.35}$$

$$h_k = \frac{\langle x u_k | u_{k-1} \rangle}{\langle u_{k-1} | u_{k-1} \rangle}, \tag{2.36}$$

with the starting $u_0(x) = 1$ and $h_0 = 0$. We can take the above polynomials and show that they are orthogonal regardless of whether they are continuous or discrete; they always satisfy $\langle u_k | u_l \rangle = \delta_{kl} \mathcal{N}_k$. For simplicity, we will just consider the case with $w(x) = 1$. The formalism developed here can easily be generalized to the cases with $w(x) \neq 1$. We will have more discussions on orthogonal polynomials in Chapter 6 when we introduce special functions and Gaussian quadratures.

The least-squares approximation is obtained if we find all the coefficients α_k that minimize the function $\chi^2[\alpha_k]$. In other words, we would like to have

$$\frac{\partial \chi^2[\alpha_k]}{\partial \alpha_j} = 0 \tag{2.37}$$

and

$$\frac{\partial^2 \chi^2[\alpha_k]}{\partial \alpha_j^2} > 0, \tag{2.38}$$

for $j = 0, 1, \ldots, m$. The first-order derivative of $\chi^2[\alpha_k]$ can easily be obtained. After exchanging the summation and the integration in $\partial \chi^2[\alpha_k]/\partial \alpha_j = 0$, we have

$$\alpha_j = \frac{\langle u_j|f\rangle}{\langle u_j|u_j\rangle}, \tag{2.39}$$

which ensures a minimum value of $\chi^2[\alpha_k]$, because $\partial^2\chi^2[\alpha_k]/\partial\alpha_j^2 = 2\langle u_j|u_j\rangle$ is always greater than zero. We can always construct a set of discrete orthogonal polynomials numerically in the region $[a, b]$. The following method is a simple example for obtaining a set of orthogonal polynomials $u_k(x_i)$ and the coefficients α_k for a given set of discrete data f_i at data points x_i.

```
// Method to generate the orthogonal polynomials and the
// least-squares fitting coefficients.

  public static double[][] orthogonalPolynomialFit
    (int m, double x[], double f[]) {
    int n = x.length-1;
    double u[][] = new double[m+1][n+2];
    double s[] = new double[n+1];
    double g[] = new double[n+1];
    double h[] = new double[n+1];

 // Check and fix the order of the curve
    if (m>n) {
      m = n;
      System.out.println("The highest power"
        + " is adjusted to: " + n);
    }

 // Set up the zeroth-order polynomial
    for (int i=0; i<=n; ++i) {
      u[0][i] = 1;
      double stmp = u[0][i]*u[0][i];
      s[0] += stmp;
      g[0] += x[i]*stmp;
      u[0][n+1] += u[0][i]*f[i];
    }
    g[0] = g[0]/s[0];
    u[0][n+1] = u[0][n+1]/s[0];

 // Set up the first-order polynomial
    for (int i=0; i<=n; ++i) {
      u[1][i] = x[i]*u[0][i]-g[0]*u[0][i];
      s[1] += u[1][i]*u[1][i];
      g[1] += x[i]*u[1][i]*u[1][i];
      h[1] += x[i]*u[1][i]*u[0][i];
      u[1][n+1] += u[1][i]*f[i];
    }
    g[1] = g[1]/s[1];
    h[1] = h[1]/s[0];
    u[1][n+1] = u[1][n+1]/s[1];

// Obtain the higher-order polynomials recursively
    if (m >= 2) {
      for (int i=1; i<m; ++i) {
        for (int j=0; j<=n; ++j) {
          u[i+1][j] = x[j]*u[i][j]-g[i]*u[i][j]
            -h[i]*u[i-1][j];
          s[i+1] += u[i+1][j]*u[i+1][j];
          g[i+1] += x[j]*u[i+1][j]*u[i+1][j];
          h[i+1] += x[j]*u[i+1][j]*u[i][j];
          u[i+1][n+1] += u[i+1][j]*f[j];
        }
        g[i+1] = g[i+1]/s[i+1];
        h[i+1] = h[i+1]/s[i];
        u[i+1][n+1] = u[i+1][n+1]/s[i+1];
      }
    }
    return u;
  }
```

Note that this method is only for discrete functions, with both the polynomials and least-squares fitting coefficients returned in a combined matrix. This is the typical case encountered in data analysis in science. Even with a continuous variable, it is a common practice to discretize the data first and then perform the curve fitting. However, if we must deal with continuous functions without discretization, we can generate the orthogonal polynomials with a continuous variable and apply an integration quadrature for the evaluation of the integrals involved. We will discuss the methods for generating continuous polynomials in Chapter 6. Integration quadratures will be introduced in the next section with several practical examples. A more elaborate scheme called Gaussian quadratures will be studied in Section 6.9.

2.3 The Millikan experiment

Here we use a set of data from the famous oil drop experiment of Millikan as an example to illustrate how we can actually employ the least-squares approximation discussed above. Millikan (1910) published his famous work on the oil drop experiment in *Science*. Based on the measurements of the charges carried by all the oil drops, Millikan concluded that the charge carried by any object is a multiple (with a sign) of a fundamental charge, the charge of an electron (for negative charges) or the charge of a proton (for positive charges). In the article, Millikan extracted the fundamental charge by taking the average of the measured

Table 2.2. *Data from the Millikan experiment*

k	4	5	6	7	8	9	10	11
q_k	6.558	8.206	9.880	11.50	13.14	14.82	16.40	18.04
k	12	13	14	15	16	17	18	
q_k	19.68	21.32	22.96	24.60	26.24	27.88	29.52	

charges carried by all the oil drops. Here we are going to take the data of Millikan and make a least-squares fit to a straight line. Based on the fit, we can estimate the fundamental charge and the accuracy of the Millikan measurement. Millikan did not use the method that we are discussing here to reach his conclusion, of course.

Each measured data point from the Millikan experiment is assigned an integer. The measured charges q_k (in units of 10^{-19} C) and the corresponding integers k are listed in Table 2.2.

From the average charges of the oil drops, Millikan concluded that the fundamental charge is $e = 1.65 \times 10^{-19}$ C, which is very close to the currently accepted value of the fundamental charge, $e = 1.602\,177\,33(49) \times 10^{-19}$ C. Let us take the Millikan's data obtained in Table 2.2 and apply the least-squares approximation discussed in the preceding section for a discrete function to find the fundamental charge and the order of the error in the measurements. We can take the linear equation

$$q_k = ke + \Delta q_k \tag{2.40}$$

as the approximation of the measured data. For simplicity, Δq_k is taken as a constant Δq to represent the overall error associated with the measurement. We leave the problem with different Δq_k as an exercise for the reader. Note that if we fit the data to a straight line in the xy plane, Δq is the intercept on the y axis. The following program applies the least-squares approximation and calculates the fundamental charge e and the overall error Δq using the data from the Millikan experiment.

```
// An example of applying the least-squares approximation
// to the Millikan data on a straight line q=k*e+dq.

import java.lang.*;
public class Millikan {
  public static void main(String argv[]) {
    double k[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
      15, 16, 17, 18};
    double q[] = {6.558, 8.206, 9.880, 11.50, 13.14,
      14.81, 16.40, 18.04, 19.68, 21.32, 22.96, 24.60,
      26.24, 27.88, 29.52};
    int n = k.length-1;
    int m = 21;
    double u[][] = orthogonalPolynomialFit(m, k, q);
```

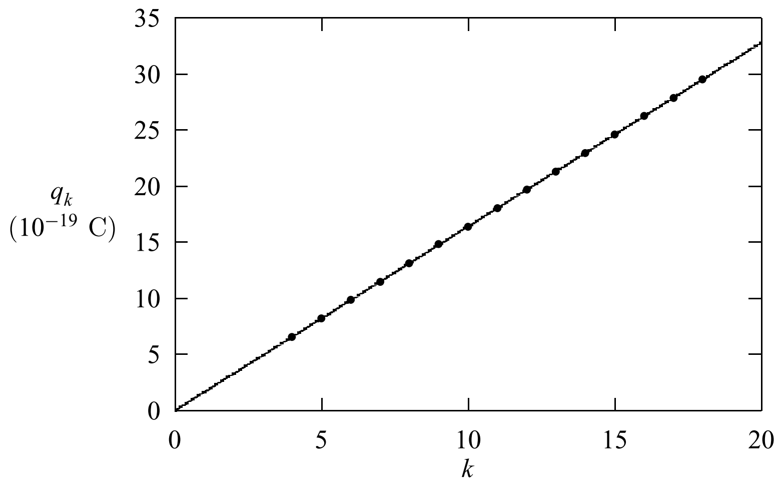


Fig. 2.3 The Millikan measurements of charges carried by the oil drops (dots) and the least-squares approximation of a linear fit (solid line).

```
    double sum = 0;
    for (int i=0; i<=n; ++i) sum += k[i];
    double e = u[1][n+1];
    double dq = u[0][n+1]-u[1][n+1]*sum/(n+1);
    System.out.println("Fundamental charge: " + e);
    System.out.println("Estimated error: " + dq);
  }

  public static double[][] orthogonalPolynomialFit
    (int m, double x[], double f[]) {...}
}
```

Note that we have used ... to represent the body of a method that we introduced earlier to avoid repeating the material. We will do this throughout the book, but the same programs are listed in their entirety on the website for the book. We have also used the fact that $u_0 = 1$, $u_1(x) = x - g_0$, and $g_0 = \sum_{i=0}^{n} x_i/(n+1)$ from Eqs. (2.34) and (2.35) in the above program. So from the linear function $f(x) = a_0 + a_1 x = \alpha_0 + \alpha_1 u_1(x)$, we know that $a_1 = \alpha_1$ and $a_0 = \alpha_0 - g_0 = \alpha_0 - \sum_{i=0}^{n} x_i/(n+1)$ after taking $x = 0$.

After we compile and run the above program, we obtain $e \simeq 1.64 \times 10^{-19}$ C, and the intercept on the y axis gives us a rough estimate of the error bar $|\Delta e| \approx |\Delta q| = 0.03 \times 10^{-19}$ C. The Millikan data and the least-squares approximation of Eq. (2.40) with e and Δq obtained from the above program are plotted in Fig. 2.3. Note that the measured data are very accurately represented by the straight line of the least-squares approximation.

The approach here is very general and we can easily fit the Millikan data to a higher-order curve by changing $m = 1$ to a higher value, for example, $m = 21$, as we actually did in the above program. After running the program, we found that other coefficients are vanishingly small. However, if the data set is nonlinear, other coefficients will become significant.

For the linear fit of the Millikan data, we can take a much simpler approach, like the one given in Eqs. (2.25)–(2.30). The following program is the result of such an approach.

```
// An example of directly fitting the Millikan data to
// p1(x) = a0+a1*x.

import java.lang.*;
public class Millikan2 {
  public static void main(String argv[]) {
    double x[] = {4, 5, 6, 7, 8, 9, 10, 11,
      12, 13, 14, 15, 16, 17, 18};
    double f[] = {6.558, 8.206, 9.880, 11.50,
      13.14, 14.81, 16.40, 18.04, 19.68, 21.32,
      22.96, 24.60, 26.24, 27.88, 29.52};
    int n = x.length-1;
    double c1 = 0, c2 = 0, c3 = 0, c4 = 0;
    for (int i=0; i<=n; ++i) {
      c1 += x[i];
      c2 += x[i]*x[i];
      c3 += f[i];
      c4 += f[i]*x[i];
    }
    double g = c1*c1-c2*(n+1);
    double a0 = (c1*c4-c2*c3)/g;
    double a1 = (c1*c3-c4*(n+1))/g;
    System.out.println("Fundamental charge: " + a1);
    System.out.println("Estimated error: " + a0);
  }
}
```

Of course, we end up with exactly the same result. The point is that sometimes we may need a general approach because we have more than one problem in mind, but other times we may need a direct approach that is simple and fast. We must learn to be able to deal with problems at different levels of complexity accordingly. In certain problems this can make the difference between finding a solution or not.

2.4 Spline approximation

In many instances, we have a set of data that varies rapidly over the range of interest, for example, a typical spectral measurement that contains many peaks and dips. Then there is no such thing as a global polynomial behavior. In such situations, we want to fit the function locally and to connect each piece of the function smoothly. A *spline* is such a tool; it interpolates the data locally through a polynomial and fits the data overall by connecting each segment of the interpolation polynomial by matching the function and its derivatives at the data points.

Assuming that we are trying to create a spline function that approximates a discrete data set $f_i = f(x_i)$ for $i = 0, 1, \ldots, n$, we can use an mth-order polynomial

$$p_i(x) = \sum_{k=0}^{m} c_{ik} x^k \tag{2.41}$$

to approximate $f(x)$ for $x \in [x_i, x_{i+1}]$. Then the coefficients c_{ik} are determined from the smoothness conditions at the nonboundary data points with the lth-order derivative there satisfying

$$p_i^{(l)}(x_{i+1}) = p_{i+1}^{(l)}(x_{i+1}), \tag{2.42}$$

for $l = 0, 1, \ldots, m-1$. These conditions and the values $p_i(x_i) = f_i$ provide $(m+1)(n-1)$ equations from the nonboundary points. So we still need $m+1$ equations in order to solve all the $(m+1)n$ coefficients a_{ik}. Two additional equations $p_0(x_0) = f_0$ and $p_{n-1}(x_n) = f_n$ are obvious and the remaining $m-1$ equations are provided by the choice of some of $p_0^{(l)}(x_0)$ and $p_{n-1}^{(l)}(x_n)$ for $l = m-1, m-2, \ldots$.

The most widely adopted spline function is the cubic spline with $m = 3$. In this case, the number of equations needed from the derivatives of the polynomials at the boundary points is $m - 1 = 2$. One of the choices, called the natural spline, is made by setting the highest-order derivatives to zero at both ends up to the number of equations needed. For the cubic spline, the natural spline is given by the choices of $p_0''(x_0) = 0$ and $p_{n-1}''(x_n) = 0$.

To construct the cubic spline, we can start with the linear interpolation of the second-order derivative in $[x_i, x_{i+1}]$,

$$p_i''(x) = \frac{1}{x_{i+1} - x_i}[(x - x_i)p_{i+1}'' - (x - x_{i+1})p_i''], \tag{2.43}$$

where $p_i'' = p_i''(x_i) = p_{i-1}''(x_i)$ and $p_{i+1}'' = p_{i+1}''(x_{i+1}) = p_i''(x_{i+1})$. If we integrate the above equation twice and use $p_i(x_i) = f_i$ and $p_i(x_{i+1}) = f_{i+1}$, we obtain

$$p_i(x) = \alpha_i(x - x_i)^3 + \beta_i(x - x_{i+1})^3 + \gamma_i(x - x_i) + \eta_i(x - x_{i+1}), \tag{2.44}$$

where

$$\alpha_i = \frac{p_{i+1}''}{6h_i}, \tag{2.45}$$

$$\beta_i = -\frac{p_i''}{6h_i}, \tag{2.46}$$

$$\gamma_i = \frac{f_{i+1}}{h_i} - \frac{h_i p_{i+1}''}{6}, \tag{2.47}$$

$$\eta_i = \frac{h_i p_i''}{6} - \frac{f_i}{h_i}, \tag{2.48}$$

with $h_i = x_{i+1} - x_i$. So if we find all p_i'', we find the spline. Applying the condition

$$p_{i-1}'(x_i) = p_i'(x_i) \tag{2.49}$$

to the polynomial given in Eq. (2.44), we have

$$h_{i-1} p''_{i-1} + 2(h_{i-1} + h_i) p''_i + h_i p''_{i+1} = 6 \left(\frac{g_i}{h_i} - \frac{g_{i-1}}{h_{i-1}} \right), \tag{2.50}$$

where $g_i = f_{i+1} - f_i$. This is a linear equation set with $n - 1$ unknowns p''_i for $i = 1, 2, \ldots, n - 1$. Note that the boundary values are fixed by $p''_0 = p''_n = 0$.

We can write the above equations in a matrix form

$$\begin{pmatrix} d_1 & h_1 & 0 & \cdots & \cdots & 0 \\ h_1 & d_2 & h_2 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & h_{n-3} & d_{n-2} & h_{n-2} \\ 0 & \cdots & \cdots & 0 & h_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} p''_1 \\ p''_2 \\ \vdots \\ p''_{n-2} \\ p''_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}, \tag{2.51}$$

or equivalently,

$$\mathbf{A}\mathbf{p}'' = \mathbf{b}, \tag{2.52}$$

where $d_i = 2(h_{i-1} + h_i)$ and $b_i = 6(g_i/h_i - g_{i-1}/h_{i-1})$. The coefficient matrix $\mathbf{A}$ in this problem is a real, symmetric, and tridiagonal matrix with elements

$$A_{ij} = \begin{cases} d_i & \text{if } i = j, \\ h_i & \text{if } i = j - 1, \\ h_{i-1} & \text{if } i = j + 1, \\ 0 & \text{otherwise}. \end{cases} \tag{2.53}$$

Because of the simplicity of the coefficient matrix, the solution of the equation set becomes quite straightforward. Here we will limit ourselves to the problem with the coefficients given in a tridiagonal matrix and leave the solution of a general linear equation set to Chapter 5.

In general, we can decompose an $m \times m$ square matrix $\mathbf{A}$ into a product of a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$,

$$\mathbf{A} = \mathbf{L}\mathbf{U}, \tag{2.54}$$

with $L_{ij} = 0$ for $i < j$ and $U_{ij} = 0$ for $i > j$. We can choose either $U_{ii} = 1$ or $L_{ii} = 1$. This scheme is called the LU decomposition. The choice of $U_{ii} = 1$ is called the Crout factorization and the alternative choice of $L_{ii} = 1$ is called the Doolittle factorization. Here we work out the Crout factorization and leave the Doolittle factorization as an exercise. For a tridiagonal matrix $\mathbf{A}$ with

$$A_{ij} = \begin{cases} d_i & \text{if } i = j, \\ e_i & \text{if } i = j - 1, \\ c_{i-1} & \text{if } i = j + 1, \\ 0 & \text{otherwise}, \end{cases} \tag{2.55}$$

the matrices **L** and **U** are extremely simple and are given by

$$L_{ij} = \begin{cases} w_i & \text{if } i = j, \\ v_{i-1} & \text{if } i = j+1, \\ 0 & \text{otherwise}, \end{cases} \tag{2.56}$$

and

$$U_{ij} = \begin{cases} 1 & \text{if } i = j, \\ t_i & \text{if } i = j-1, \\ 0 & \text{otherwise}. \end{cases} \tag{2.57}$$

The elements in **L** and **U** can be related easily to d_i, c_i, and e_i if we multiply **L** and **U** and compare the two sides of Eq. (2.54) element by element. Then we have

$$v_i = c_i, \tag{2.58}$$

$$t_i = e_i/w_i, \tag{2.59}$$

$$w_i = d_i - v_{i-1}t_{i-1}, \tag{2.60}$$

with $w_1 = d_1$. The solution of the linear equation $\mathbf{Az} = \mathbf{b}$ can be obtained by forward and backward substitutions because

$$\mathbf{Az} = \mathbf{LUz} = \mathbf{Ly} = \mathbf{b}. \tag{2.61}$$

We can first solve $\mathbf{Ly} = \mathbf{b}$ and then $\mathbf{Uz} = \mathbf{y}$ with

$$y_i = (b_i - v_{i-1}y_{i-1})/w_i, \tag{2.62}$$

$$z_i = y_i - t_i z_{i+1}, \tag{2.63}$$

with $y_1 = b_1/w_1$ and $z_m = y_m$. The following program is an implementation of the cubic spline with the solution of the tridiagonal linear equation set through Crout factorization.

```
// An example of creating cubic-spline approximation of
// a discrete function fi=f(xi).

import java.lang.*;
import java.io.*;
import java.util.*;
public class Spline {
  final static int n = 20;
  final static int m = 100;
  public static void main(String argv[]) throws
    IOException {
    double xi[] = new double[n+1];
    double fi[] = new double[n+1];
    double p2[] = new double[n+1];

 // Read in data points xi and and data fi
    BufferedReader r = new BufferedReader
      (new InputStreamReader
          (new FileInputStream("xy.data")));
      for (int i=0; i<=n; ++i) {
        StringTokenizer
          s = new StringTokenizer(r.readLine());
        xi[i] = Double.parseDouble(s.nextToken());
        fi[i] = Double.parseDouble(s.nextToken());
      }
      p2 = cubicSpline(xi, fi);

   // Find the approximation of the function
      double h = (xi[n]-xi[0])/m;
      double x = xi[0];
      for (int i=1; i<m; ++i) {
        x += h;

     // Find the interval where x resides
        int k = 0;
        double dx = x-xi[0];
        while (dx > 0) {
          ++k;
          dx = x-xi[k];
        }
        --k;

     // Find the value of function f(x)
        dx = xi[k+1]-xi[k];
        double alpha = p2[k+1]/(6*dx);
        double beta = -p2[k]/(6*dx);
        double gamma = fi[k+1]/dx-dx*p2[k+1]/6;
        double eta = dx*p2[k]/6 - fi[k]/dx;
        double f = alpha*(x-xi[k])*(x-xi[k])*(x-xi[k])
          +beta*(x-xi[k+1])*(x-xi[k+1])*(x-xi[k+1])
          +gamma*(x-xi[k])+eta*(x-xi[k+1]);
        System.out.println(x + " " + f);
      }
    }

  // Method to carry out the cubic-spline approximation
  // with the second-order derivatives returned.

    public static double[] cubicSpline(double x[],
      double f[]) {
      int n = x.length-1;
      double p[] = new double [n+1];
      double d[] = new double [n-1];
      double b[] = new double [n-1];
      double c[] = new double [n-1];
      double g[] = new double [n];
      double h[] = new double [n];

   // Assign the intervals and function differences
      for (int i=0; i<n; ++i) {
        h[i] = x[i+1]-x[i];
        g[i] = f[i+1]-f[i];
      }

   // Evaluate the coefficient matrix elements
      for (int i=0; i<n-1; ++i) {
        d[i] = 2*(h[i+1]+h[i]);
        b[i] = 6*(g[i+1]/h[i+1]-g[i]/h[i]);
        c[i] = h[i+1];
      }
   // Obtain the second-order derivatives
      g = tridiagonalLinearEq(d, c, c, b);
      for (int i=1; i<n; ++i) p[i] = g[i-1];
      return p;
    }

  // Method to solve the tridiagonal linear equation set.

    public static double[] tridiagonalLinearEq(double d[],
      double e[], double c[], double b[]) {
      int m = b.length;
      double w[] = new double[m];
      double y[] = new double[m];
      double z[] = new double[m];
      double v[] = new double[m-1];
      double t[] = new double[m-1];

   // Evaluate the elements in the LU decomposition
      w[0] = d[0];
      v[0] = c[0];
      t[0] = e[0]/w[0];
      for (int i=1; i<m-1; ++i) {
        w[i]  = d[i]-v[i-1]*t[i-1];
        v[i]  = c[i];
        t[i]  = e[i]/w[i];
      }
      w[m-1]  = d[m-1]-v[m-2]*t[m-2];

   // Forward substitution to obtain y
      y[0] = b[0]/w[0];
      for (int i=1; i<m; ++i)
        y[i] = (b[i]-v[i-1]*y[i-1])/w[i];

   // Backward substitution to obtain z
      z[m-1] = y[m-1];
      for (int i=m-2; i>=0; --i) {
        z[i] = y[i]-t[i]*z[i+1];
      }
      return z;
    }
  }
```

We have used a set of uniform random numbers from the generator in next section as the data function at uniformly spaced data points in [0, 1]. The output of the above program is plotted together with the original data in Fig. 2.4.

The above approach can be generalized to a higher-order spline. For example, the *quintic spline* is achieved by including forth-order and fifth-order terms. Then the polynomial in the interval $[x_i, x_{i+1}]$ is given by

$$\begin{aligned} p_i(x) = {} & \alpha_i(x-x_i)^5 + \beta_i(x-x_{i+1})^5 + \gamma_i(x-x_i)^3 \\ & + \eta_i(x-x_{i+1})^3 + \delta_i(x-x_i) + \sigma_i(x-x_{i+1}), \end{aligned} \tag{2.64}$$

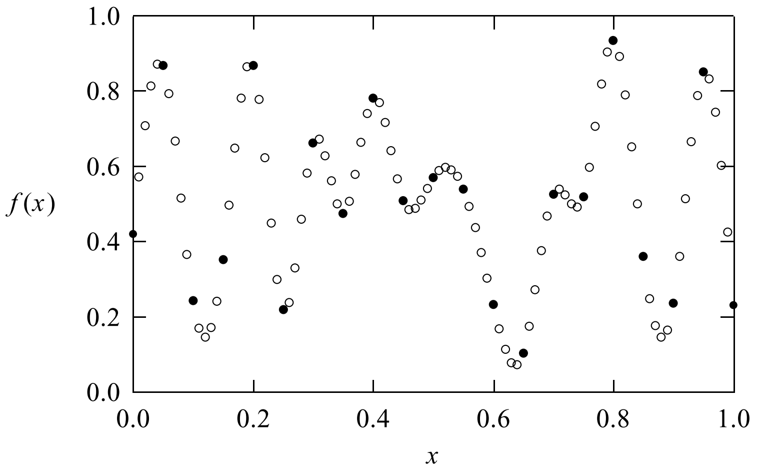


Fig. 2.4 An example of a cubic-spline approximation. The original data are shown as solid circles and the approximations as open circles.

where

$$\alpha_i = \frac{p_{i+1}^{(4)}}{120h_i}, \tag{2.65}$$

$$\beta_i = -\frac{p_i^{(4)}}{120h_i}. \tag{2.66}$$

We must also have $p_i(x_i) = f_i$ and $p_i(x_{i+1}) = f_{i+1}$, which give

$$h_i^5\alpha_i + h_i^3\gamma_i + h_i\delta_i = f_{i+1}, \tag{2.67}$$

$$h_i^5\beta_i + h_i^3\eta_i + h_i\sigma_i = -f_i, \tag{2.68}$$

where $h_i = x_{i+1} - x_i$. In order for the pieces to join smoothly, we also impose that $p_{i-1}^{(l)}(x_i) = p_i^{(l)}(x_i)$ for $l = 1, 2, 3$. From $p'_{i-1}(x_i) = p'_i(x_i)$, we have

$$5h_{i-1}^4\alpha_{i-1} + 3h_{i-1}^2\gamma_{i-1} + \delta_{i-1} + \sigma_{i-1} = 5h_i^4\beta_i + 3h_i^2\eta_i + \delta_i + \sigma_i. \tag{2.69}$$

For $l = 2$, we have

$$10h_{i-1}^3\alpha_{i-1} + 3h_{i-1}\gamma_{i-1} = 10h_i^3\beta_i + 3h_i\eta_i. \tag{2.70}$$

The continuity of the third-order derivative $p_{i-1}^{(3)}(x_i) = p_i^{(3)}(x_i)$ leads to

$$10h_{i-1}^2\alpha_{i-1} + \gamma_{i-1} = 10h_i^2\beta_i + \eta_i. \tag{2.71}$$

Note that we always have $p_i^{(4)} = 120h_{i-1}\alpha_{i-1} = -120h_i\beta_i$. Then from the equations with $l = 2$ and $l = 3$, we obtain

$$\gamma_i = -\frac{h_i^2 + 3h_ih_{i+1} + 2h_{i+1}^2}{36(h_i + h_{i+1})}p_{i+1}^{(4)}, \tag{2.72}$$

$$\eta_i = \frac{2h_{i-1}^2 + 3h_{i-1}h_i + 2h_i^2}{36(h_{i-1} + h_i)}p_i^{(4)}. \tag{2.73}$$

Then we can use Eqs. (2.67) and (2.68) to obtain δ_i and σ_i in terms of $p_i^{(4)}$. Substituting these into Eq. (2.69), we arrive at the linear equation set for $p_i^{(4)}$ with a coefficient matrix in tridiagonal form that can be solved as done for the cubic spline.

2.5 Random-number generators

In practice, many numerical simulations need random-number generators either for setting up initial configurations or for generating new configurations. There is no such thing as *random* in a computer program. A computer will always produce the same result if the input is the same. A random-number generator here really means a pseudo-random-number generator that can generate a long sequence of numbers that can imitate a given distribution. In this section we will discuss some of the basic random-number generators used in computational physics and other computer simulations.

Uniform random-number generators

The most useful random-number generators are those with a uniform distribution in a given region. The three most important criteria for a good uniform random-number generator are the following (Park and Miller, 1988).

First, a good generator should have a long period, which should be close to the range of the integers adopted. For example, if 32-bit integers are used, a good generator should have a period close to $2^{31} - 1 = 2\,147\,483\,647$. The range of the 32-bit integers is $[-2^{31}, 2^{31} - 1]$. Note that one bit is used for the sign of the integers.

Second, a good generator should have the best *randomness*. There should only be a very small correlation among all the numbers generated in sequence. If $\langle A \rangle$ represents the statistical average of the variable A, the k-point correlation function of the numbers generated in the sequence $\langle x_{i_1} x_{i_2} \cdots x_{i_k} \rangle$ for $k = 2, 3, 4, \ldots$ should be very small. One way to illustrate the behavior of the correlation function $\langle x_i x_{i+l} \rangle$ is to plot x_i and x_{i+l} in the xy plane. A good random-number generator will have a very uniform distribution of the points for any $l \neq 0$. A poor generator may show stripes, lattices, or other inhomogeneous distributions.

Finally, a good generator has to be very fast. In practice, we need a lot of random numbers in order to have good statistical results. The speed of the generator can become a very important factor.

The simplest uniform random-number generator is built using the so-called linear congruent scheme. The random numbers are generated in sequence from the linear relation

$$x_{i+1} = (a x_i + b) \bmod c, \tag{2.74}$$

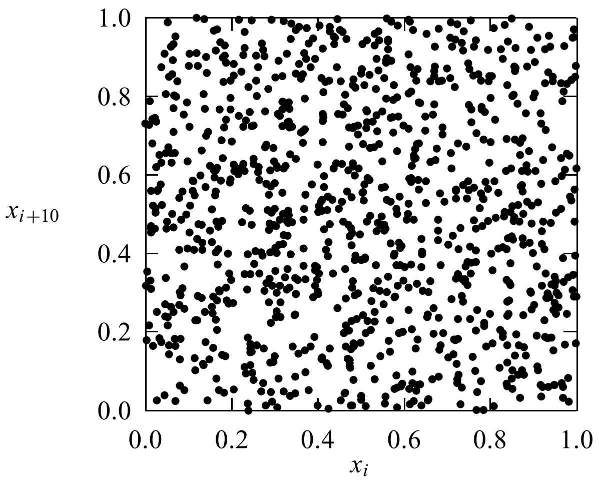


Fig. 2.5 One thousand points of the random-number pairs (x_i, x_{i+10}) normalized to the range of [0, 1].

where a, b, and c are *magic numbers*: their values determine the quality of the generator. One common choice, $a = 7^5 = 16\,807$, $b = 0$, and $c = 2^{31} - 1 = 2\,147\,483\,647$, has been tested and found to be excellent for generating unsigned 32-bit random integers. It has the full period of $2^{31} - 1$ and is very fast. The correlation function $\langle x_{i_1} x_{i_2} \ldots x_{i_k} \rangle$ is very small. In Fig. 2.5, we plot x_i and x_{i+10} (normalized by c) generated using the linear congruent method with the above selection of the magic numbers. Note that the plot is very homogeneous and random. There are no stripes, lattice structures, or any other visible patterns in the plot.

Implementation of this random-number generator on a computer is not always trivial, because of the different numerical range of the integers specified by the computer language or hardware. For example, most 32-bit computers have integers in $[-2^{31}, 2^{31} - 1]$. If a number runs out of this range by accident, the computer will reset it to zero. If the computer could modulate the integers by $2^{31} - 1$ automatically, we could implement a random-number generator with the above magic numbers a, b, and c simply by taking consecutive numbers from $x_{i+1} = 16\,807 x_i$ with any initial choice of $1 < x_0 < 2^{31} - 1$. The range of this generator would be $[0, 2^{31} - 1]$. However, this automatic modulation would cause some serious problems in other situations. For example, when a quantity is out of range due to a bug in the program, the computer would still wrap it back without producing an error message. This is why, in practice, computers do not modulate numbers automatically, so we have to devise a scheme to modulate the numbers generated in sequence. The following method is an implementation of the uniform random-number generator (normalized to the range of [0, 1]) discussed above.

```
// Method to generate a uniform random number in [0,1]
// following x(i+1)=a*x(i) mod c with a=pow(7,5) and
// c=pow(2,31)-1.  Here the seed is a global variable.

  public static double ranf() {
    final int a = 16807, c = 2147483647, q = 127773,
      r = 2836;
    final double cd = c;
    int h = seed/q;
    int l = seed%q;
    int t = a*l-r*h;
    if (t > 0) seed = t;
    else seed = c+t;
    return seed/cd;
  }
```

Note that in the above code, we have used two more magic numbers $q = c/a$ and $r = c \bmod a$. A program in Pascal that implements a method similar to that above is given by Park and Miller (1988). We can easily show that the above method modulates numbers with $c = 2^{31} - 1$ on any computer with integers of 32 bits or more. To use this method, the seed needs to be a global variable that is returned each time that the method is called. To show that the method given here does implement the algorithm correctly, we can set the initial seed to be 1, and then after 10 000 steps, we should have 1 043 618 065 returned as the value of the seed (Park and Miller, 1988).

The above generator has a period of $2^{31} - 1$. If a longer period is desired, we can create similar generators with higher-bit integers. For example, we can create a generator with a period of $2^{63} - 1$ for 64-bit integers with the following method.

```
// Method to generate a uniform random number in [0,1]
// following x(i+1)=a*x(i) mod c with a=pow(7,5) and
// c=pow(2,63)-1.  Here the seed is a global variable.

  public static double ranl() {
    final long a = 16807L, c = 9223372036854775807L,
      q = 548781581296767L, r = 12838L;
    final double cd = c;
    long h = seed/q;
    long l = seed%q;
    long t = a*l-r*h;
    if (t > 0) seed = t;
    else seed = c+t;
    return seed/cd;
  }
```

Note that we have used $a = 7^5$, $c = 2^{63} - 1$, $q = c/a$, and $r = c \bmod a$ in the above method with the seed being a 64-bit (`long`) integer.

In order to start the random-number generator differently every time, we need to have a systematic way of obtaining a different initial seed. Otherwise, we would not be able to obtain fair statistics. Almost every computer language has intrinsic routines to report the current time in an integer form, and we can use this integer to construct an initial seed (Anderson, 1990). For example, most computers can produce $0 \leq t_1 \leq 59$ for the second of the minute, $0 \leq t_2 \leq 59$ for the minute of the hour, $0 \leq t_3 \leq 23$ for the hour of the day, $1 \leq t_4 \leq 31$ for the day of the month, $1 \leq t_5 \leq 12$ for the month of the year, and t_6 for the current year in common era. Then we can choose

$$i_s = t_6 + 70\Big(t_5 + 12\{t_4 + 31[t_3 + 23(t_2 + 59t_1)]\}\Big) \tag{2.75}$$

as the initial seed, which is roughly in the region of $[0, 2^{31} - 1]$. The results should never be the same as long as the jobs are started at least a second apart.

We now demonstrate how to apply the random-number generator and how to initiate the generator with the current time through a simple example. Consider evaluating π by randomly throwing a dart into a unit square defined by $x \in [0, 1]$ and $y \in [0, 1]$. By comparing the areas of the unit square and the quarter of the unit circle we can see that the chance of the dart landing inside a quarter of the unit circle centered at the origin of the coordinates is $\pi/4$. The following program is an implementation of such an evaluation of π in Java.

```java
// An example of evaluating pi by throwing a dart into a
// unit square with 0<x<1 and 0<y<1.

import java.lang.*;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class Dart {
  final static int n = 1000000;
  static int seed;
  public static void main(String argv[]) {

 // Initiate the seed from the current time
    GregorianCalendar t = new GregorianCalendar();
    int t1 = t.get(Calendar.SECOND);
    int t2 = t.get(Calendar.MINUTE);
    int t3 = t.get(Calendar.HOUR_OF_DAY);
    int t4 = t.get(Calendar.DAY_OF_MONTH);
    int t5 = t.get(Calendar.MONTH)+1;
    int t6 = t.get(Calendar.YEAR);
    seed = t6+70*(t5+12*(t4+31*(t3+23*(t2+59*t1))));
    if ((seed%2) == 0) seed = seed-1;

 // Throw the dart into the unit square
    int ic = 0;
    for (int i=0; i<n; ++i) {
      double x = ranf();
      double y = ranf();
      if ((x*x+y*y) < 1) ic++;
    }
    System.out.println("Estimated pi: " + (4.0*ic/n));
  }

  public static double ranf() {...}
}
```

An even initial seed is usually avoided in order to have the full period of the generator realized. Note that in Java the months are represented by the numerals 0–11, so we have added 1 in the above program to have it between 1 and 12. To initiate the 64-bit generator, we can use the method `getTime()` from the `Date` class in Java, which returns the current time in milliseconds in a 64-bit (`long`) integer, measured from the beginning of January 1, 1970.

Uniform random-number generators are very important in scientific computing, and good ones are extremely difficult to find. The generator given here is considered to be one of the best uniform random-number generators.

New computer programming languages such as Java typically come with a comprehensive, intrinsic set of random-number generators that can be initiated automatically with the current time or with a chosen initial seed. We will demonstrate the use of such a generator in Java later in this section.

Other distributions

As soon as we obtain good uniform random-number generators, we can use them to create other types of random-number generators. For example, we can use a uniform random-number generator to create an exponential distribution or a Gaussian distribution.

All the exponential distributions can be cast into their simplest form

$$p(x) = e^{-x} \tag{2.76}$$

after a proper choice of units and coordinates. For example, if a system has energy levels of $E_0, E_1, \ldots, E_n$, the probability for the system to be at the energy level E_i at temperature T is given by

$$p(E_i, T) \propto e^{-(E_i - E_0)/k_B T}, \tag{2.77}$$

where k_B is the Boltzmann constant. If we choose $k_B T$ as the energy unit and E_0 as the zero point, the above equation reduces to Eq. (2.76).

One way to generate the exponential distribution is to relate it to a uniform distribution. For example, if we have a uniform distribution $f(y) = 1$ for $y \in [0, 1]$, we can relate it to an exponential distribution by

$$f(y)\,dy = dy = p(x)\,dx = e^{-x}\,dx, \tag{2.78}$$

which gives

$$y(x) - y(0) = 1 - e^{-x} \tag{2.79}$$

after integration. We can set $y(0) = 0$ and invert the above equation to have

$$x = -\ln(1 - y), \tag{2.80}$$

which relates the exponential distribution of $x \in [0, \infty]$ to the uniform distribution of $y \in [0, 1]$. The following method is an implementation of the exponential random-number generator given in the above equation and is constructed from a uniform random-number generator.

```
// Method to generate an exponential random number from a
// uniform random number in [0,1].

  public static double rane() {
    return -Math.log(1-ranf());
  }

  public static double ranf() {...}
```

The uniform random-number generator obtained earlier is used in this method. Note that when this method is used in a program, the seed still has to be a global variable.

As we have pointed out, another useful distribution in physics is the Gaussian distribution

$$g(x) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-x^2/2\sigma^2}, \tag{2.81}$$

where σ is the variance of the distribution, which we can take as 1 for the moment. The distribution with $\sigma \neq 1$ can be obtained via the rescaling of x by σ. We can use a uniform distribution $f(\phi) = 1$ for $\phi \in [0, 2\pi]$ and an exponential distribution $p(t) = e^{-t}$ for $t \in [0, \infty]$ to obtain two Gaussian distributions $g(x)$ and $g(y)$. We can relate the product of a uniform distribution and an exponential distribution to a product of two Gaussian distributions by

$$\frac{1}{2\pi} f(\phi)\, d\phi\, p(t)\, dt = g(x)\, dx\, g(y)\, dy, \tag{2.82}$$

which gives

$$e^{-t} dt\, d\phi = e^{-(x^2+y^2)/2} dx\, dy. \tag{2.83}$$

The above equation can be viewed as the coordinate transform from the polar system (ρ, ϕ) with $\rho = \sqrt{2t}$ into the rectangular system (x, y), that is,

$$x = \sqrt{2t} \cos\phi, \tag{2.84}$$

$$y = \sqrt{2t} \sin\phi, \tag{2.85}$$

which are two Gaussian distributions if t is taken from an exponential distribution and ϕ is taken from a uniform distribution in the region $[0, 2\pi]$. With the availability of the exponential random-number generator and uniform random-number generator, we can construct two Gaussian random numbers immediately from Eqs. (2.74) and (2.75). The exponential random-number generator itself can be obtained from a uniform random-number generator as discussed above. Below we show how to create two Gaussian random numbers from two uniform random numbers.

```
// Method to create two Gaussian random numbers from two
// uniform random numbers in [0,1].

  public static double[] rang() {
    double x[] = new double[2];
    double r1 = - Math.log(1-ranf());
    double r2 = 2*Math.PI*ranf();
    r1 = Math.sqrt(2*r1);
    x[0] = r1*Math.cos(r2);
    x[1] = r1*Math.sin(r2);
    return x;
  }

  public static double ranf() {...}
```

In principle, we can generate any given distribution numerically. For the Gaussian distribution and exponential distribution, we construct the new generators with the integral transformations in order to relate the distributions sought to the distributions known. A general procedure can be devised by dealing with the integral transformation numerically. For example, we can use the Metropolis algorithm, which will be discussed in Chapter 10, to obtain any distribution numerically.

Percolation in two dimensions

Let us use two-dimensional percolation as an example to illustrate how a random-number generator is utilized in computer simulations. When atoms are added to a solid surface, they first occupy the sites with the lowest potential energy to form small two-dimensional clusters. If the probability of occupying each empty site is still high, the clusters grow on the surface and eventually form a single layer, or a percolated two-dimensional network. If the probability of occupying each empty site is low, the clusters grow into island-like three-dimensional clusters. The general problem of the formation of a two-dimensional network can be cast into a simple model with each site carrying a fixed occupancy probability.

Assume that we have a two-dimensional square lattice with $n \times n$ lattice points. Then we can generate $n \times n$ random numbers $x_{ij} \in [0, 1]$ for $i = 0, 1, \ldots, n-1$ and $j = 0, 1, \ldots, n-1$. The random number x_{ij} is further compared with the assigned occupancy probability $p \in [0, 1]$. The site is occupied if $p > x_{ij}$; otherwise the site remains empty. Clusters are formed by the occupied sites. A site in each cluster is, at least, a nearest neighbor of another site in the same cluster. We can gradually change p from 0 to 1. As p increases, the sizes of the clusters increase and some clusters merge into larger clusters. When p reaches a critical probability p_c, there is one cluster of occupied sites, which reaches all the boundaries of the lattice. We call p_c the percolation threshold. The following method is the core of the simulation of two-dimensional percolation, which assigns a false value to an empty site and a true value to an occupied site.

```
// Method to create a 2-dimensional percolation lattice.

import java.util.Random;
  public static boolean[][] lattice(double p, int n) {
    Random r = new Random();
    boolean y[][] = new boolean[n][n];
    for (int i=0; i<n; ++i) {
      for (int j=0; j<n; ++j) {
        if (p > r.nextDouble()) y[i][j] = true;
        else y[i][j] = false;
      }
    }
    return y;
  }
```

Here y_{ij} is a boolean array that contains true values at all the occupied sites and false values at all the empty sites. We can use the above method with a program that has p increased from 0 to 1, and can sort out the sizes of all the clusters formed by the occupied lattice sites. In order to obtain good statistical averages, the procedure should be carried out many times. For more discussions on percolation, see Stauffer and Aharony (1992) and Grimmett (1999). Note that we have used the intrinsic random-number generator from Java: `nextDouble()` is a method in the `Random` class, and it creates a floating-point random number in [0, 1] when it is called. The default initiation of the generator, as used above, is from the current time.

The above example is an extremely simple application of the uniform random-number generator in physics. In Chapters 8, 10, and 11, we will discuss the application of random-number generators in other simulations. Interested readers can find more on different random-number generators in Knuth (1998), Park and Miller (1988), and Anderson (1990).

Exercises

2.1 Show that the error in the nth-order Lagrange interpolation scheme is bounded by

$$|\Delta f(x)| \leq \frac{\gamma_n}{4(n+1)} h^{n+1},$$

where $\gamma_n = \max[|f^{(n+1)}(x)|]$, for $x \in [x_0, x_n]$.

2.2 Write a program that implements the Lagrange interpolation scheme directly. Test it by evaluating $f(0.3)$ and $f(0.5)$ from the data taken from the error function with $f(0.0) = 0$, $f(0.4) = 0.428\,392$, $f(0.8) = 0.742\,101$, $f(1.2) = 0.910\,314$, and $f(1.6) = 0.970\,348$. Examine the accuracy of the interpolation by comparing the results obtained from the interpolation with the exact values $f(0.3) = 0.328\,627$ and $f(0.5) = 0.520\,500$.

2.3 The Newton interpolation is another popular interpolation scheme that adopts the polynomial

$$p_n(x) = \sum_{j=0}^{n} c_j \prod_{i=0}^{j-1} (x - x_i),$$

where $\prod_{i=0}^{-1}(x - x_i) = 1$. Show that this polynomial is equivalent to that of the Lagrange interpolation and the coefficients c_j here are recursively given by

$$c_j = \frac{f_j - \sum_{k=0}^{j-1} c_k \prod_{i=0}^{k-1} (x_j - x_i)}{\prod_{i=0}^{j-1} (x_j - x_i)}.$$

Write a subprogram that creates all c_j with given x_i and f_i.

2.4 Show that the coefficients in the Newton interpolation, defined in Exercise 2.3, can be cast into divided differences recursively as

$$a_{i\ldots j} = \frac{a_{i+1\ldots j} - a_{i\ldots j-1}}{x_j - x_i},$$

where $a_i = f_i$ are the discrete data and $a_{0\ldots i} = c_i$ are the coefficients in the Newton interpolation. Write a subprogram that creates c_i in this way. Apply this subprogram to create another subprogram that evaluates the interpolated value from the nested expression of the polynomial

$$p_n(x) = c_0 + (x - x_0)\{c_1 + (x - x_1)[c_2 + \cdots + (x - x_{n-1})c_n \cdots]\}.$$

Use the values of the Bessel function in Section 2.1 to test the program.

2.5 The Newton interpolation of Exercise 2.3 can be used inversely to find approximate roots of an equation $f(x) = 0$. Show that $x = p_n(0)$ is such an approximate root if the polynomial

$$p_n(z) = \sum_{j=0}^{n} c_j \prod_{i=0}^{j-1} (z - f_i),$$

where $\prod_{i=0}^{-1}(x - f_i) = 1$. Develop a program that estimates the root of $f(x) = e^x \ln x - x^2$ with $x_i = 1, 1.1, \ldots, 2.0$.

2.6 Modify the program in Section 2.3 for the least-squares approximation to fit the data set $f(0.0) = 1.000\,000$, $f(0.2) = 0.912\,005$, $f(0.4) = 0.671\,133$, $f(0.6) = 0.339\,986$, $f(0.8) = 0.002\,508$, $f(0.9) = -0.142\,449$, and $f(1.0) = -0.260\,052$ and the corresponding points of $f(-x) = f(x)$ as the input. The data are taken from the Bessel function table with $f(x) = J_0(3x)$. Compare the results with the well-known approximate formula for the Bessel function,

$$f(x) = 1 - 2.249\,999\,7\,x^2 + 1.265\,620\,8\,x^4 - 0.316\,386\,6\,x^6 \\ + 0.044\,447\,9\,x^8 - 0.003\,944\,4\,x^{10} + 0.000\,210\,0\,x^{12}.$$

2.7 Use the program in Section 2.3 that fits the Millikan data directly to a linear function to analyze the accuracy of the approximation. Assume that the error bars in the experimental measurements are $|\Delta q_k| = 0.05 q_k$. The accuracy of a curve fitting is determined from

$$\chi^2 = \frac{1}{n+1} \sum_{i=0}^{n} \frac{|f(x_i) - p_m(x_i)|^2}{\sigma_i^2},$$

where σ_i is the error bar of $f(x_i)$ and m is the order of the polynomial. If $\chi \ll 1$, the fitting is considered successful.

2.8 For periodic functions, we can approximate the function in one period with the periodic boundary condition imposed. Modify the cubic-spline program in Section 2.4 to have $p_{n-1}(x_n) = p_0(x_0)$, $p'_{n-1}(x_n) = p'_0(x_0)$, and $p''_{n-1}(x_n) = p''_0(x_0)$. Test the program with $f_i \in [0, 1]$ and $x_i \in [0, 1]$ generated randomly and sorted according to $x_{i+1} \geq x_i$ for $n = 20$.

2.9 Modify the cubic-spline program given in Section 2.4 to have the LU decomposition carried out by the Doolittle factorization.

2.10 Use a fifth-order polynomial

$$p_i(x) = \sum_{k=0}^{5} a_{ik} x^k$$

for $x \in [x_i, x_{i+1}]$ to create a quintic spline approximation for a function $f(x)$ given at discrete data points, $f_i = f(x_i)$ for $i = 0, 1, \ldots, n$, with $p_0^{(4)}(x_0) = p_{n-1}^{(4)}(x_n) = p_0^{(3)}(x_0) = p_{n-1}^{(3)}(x_n) = 0$. Find the linear equation set whose solution provides $p_i^{(4)}$ at every data point. Is the numerical problem here significantly different from that of the cubic spline? Modify the cubic-spline program given in Section 2.4 to one for the quintic spline and test your program by applying it to approximate $f_i \in [0, 1]$, generated randomly, at $x_i \in [0, 1]$, also generated randomly and sorted according to $x_{i+1} \geq x_i$ for $n = 40$.

2.11 We can approximate a function $f(x)$ as a linear combination of a set of basis functions $B_{nk}(x)$ through

$$p(x) = \sum_{j=-\infty}^{\infty} A_j B_{nj-n}(x),$$

for a chosen integer $n \geq 0$, where the functions $B_{nk}(x)$ are called B splines of degree n, which can be constructed recursively with

$$B_{nk}(x) = \frac{x - x_k}{x_{k+n} - x_k} B_{n-1k}(x) + \frac{x_{l+n+1} - x}{x_{k+n+1} - x_{k+1}} B_{n-1k+1}(x),$$

starting from

$$B_{0k}(x) = \begin{cases} 1 & \text{for } x_k \le x < x_{k+1}, \\ 0 & \text{elsewhere}. \end{cases}$$

If $x_{k+1} \ge x_k$ and $f_k = p(x_k)$, show that

$$B_{nk}(x) > 0 \quad \text{and} \quad \sum_{j=-\infty}^{\infty} B_{nj}(x) = 1.$$

Develop a computer program to implement B splines of degree 3. Compare the result for a set data against the cubic-spline approximation.

2.12 If we randomly drop a needle of unit length on an infinite plane that is covered by parallel lines one unit apart, the probability of the needle landing in a gap (not crossing any line) is $1 - 2/\pi$. Derive this probability analytically. Write a program that can sample the needle dropping process and calculate the probability without explicitly using the value of π. Compare you numerical result with the analytical result and discuss the possible sources of error.

2.13 Generate 21 pairs of random numbers (x_i, f_i) in [0, 1] and sort them according to $x_{i+1} \ge x_i$. Treat them as a discrete set of function $f(x)$ and fit them to $p_{20}(x)$, where

$$p_m(x) = \sum_{k=0}^{m} \alpha_k u_k(x),$$

with $u_k(x)$ being orthogonal polynomials. Use the method developed in Section 2.2 to obtain the corresponding $u_k(x)$ and α_k. Numerically show that the orthogonal polynomials satisfy $\langle u_k | u_l \rangle = \delta_{kl}$. Compare the least-squares approximation with the cubic-spline approximation of the same data and discuss the cause of difference.

2.14 Develop a scheme that can generate any distribution $w(x) > 0$ in a given region $[a, b]$. Write a program to implement the scheme and test it with $w(x) = 1$, $w(x) = e^{-x^2}$, and $w(x) = x^2 e^{-x^2}$. Vary a and b, and compare the results with those from the uniform random-number generator and the Gaussian random-number generator given in Section 2.5.

2.15 Generate a large set of Gaussian random numbers and sort them into an increasing order. Then count the data points falling into each of the uniformly divided intervals. Use these values to perform a least-squares fit of the generated data to the function $f(x) = ae^{-x^2/2\sigma^2}$, where a and σ are the parameters to be determined. Comment on the quality of the generator based on the fitting result.

2.16 Write a program that can generate clusters of occupied sites in a two-dimensional square lattice with $n \times n$ sites. Determine $p_{\mathrm{c}}(n)$, the threshold probability with at least one cluster reaching all the boundaries. Then determine p_{c} for an infinite lattice from

$$p_{\mathrm{c}}(n) = p_{\mathrm{c}} + \frac{c_1}{n} + \frac{c_2}{n^2} + \frac{c_3}{n^3} + \cdots ,$$

where c_i and p_{c} can be solved from the $p_{\mathrm{c}}(n)$ obtained.

Chapter 3
Numerical calculus

Calculus is at the heart of describing physical phenomena. As soon as we talk about motion, we must invoke differentiation and integration. For example, the velocity and the acceleration of a particle are the first-order and second-order time derivatives of the corresponding position vector, and the distance traveled by a particle is the integral of the corresponding speed over the elapsed time.

In this chapter, we introduce some basic computational methods for dealing with numerical differentiation and integration, and numerical schemes for searching for the roots of an equation and the extremes of a function. We stay at a basic level, using the simple but practical schemes frequently employed in computational physics and scientific computing. Some of the subjects will be reexamined later in other chapters to a greater depth. Some problems introduced here, such as searching for the global minimum or maximum of a multivariable function, are considered unsolved and are still under intensive research. Several interesting examples are given to illustrate how to apply these methods in studying important problems in physics and related fields.

3.1 Numerical differentiation

One basic tool that we will often use in this book is the Taylor expansion of a function $f(x)$ around a point x_0:

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \cdots + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0) + \cdots . \tag{3.1}$$

The above expansion can be generalized to describe a multivariable function $f(x, y, \ldots)$ around the point $(x_0, y_0, \ldots)$:

$$\begin{aligned} f(x, y, \ldots) = {} & f(x_0, y_0, \ldots) + (x - x_0)f_x(x_0, y_0, \ldots) \\ & + (y - y_0)f_y(x_0, y_0, \ldots) + \frac{(x - x_0)^2}{2!}f_{xx}(x_0, y_0, \ldots) \\ & + \frac{(y - y_0)^2}{2!}f_{yy}(x_0, y_0, \ldots) + \frac{2(x - x_0)(y - y_0)}{2!}f_{xy}(x_0, y_0, \ldots) + \cdots , \end{aligned} \tag{3.2}$$

where the subscript indices denote partial derivatives, for example, $f_{xy} = \partial^2 f/\partial x \partial y$.

The first-order derivative of a single-variable function $f(x)$ around a point x_i is defined from the limit

$$f'(x_i) = \lim_{\Delta x \to 0} \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} \tag{3.3}$$

if it exists. Now if we divide the space into discrete points x_i with evenly spaced intervals $x_{i+1} - x_i = h$ and label the function at the lattice points as $f_i = f(x_i)$, we obtain the simplest expression for the first-order derivative

$$f_i' = \frac{f_{i+1} - f_i}{h} + O(h). \tag{3.4}$$

We have used the notation $O(h)$ for a term on the order of h. Similar notation will be used throughout this book. The above formula is referred to as the *two-point formula* for the first-order derivative and can easily be derived by taking the Taylor expansion of f_{i+1} around x_i. The accuracy can be improved if we expand f_{i+1} and f_{i-1} around x_i and take the difference

$$f_{i+1} - f_{i-1} = 2hf_i' + O(h^3). \tag{3.5}$$

After a simple rearrangement, we have

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2), \tag{3.6}$$

which is commonly known as the *three-point formula* for the first-order derivative. The accuracy of the expression increases to a higher order in h if more points are used. For example, a *five-point formula* can be derived by including the expansions of f_{i+2} and f_{i-2} around x_i. If we use the combinations

$$f_{i+1} - f_{i-1} = 2hf_i' + \frac{h^3}{3} f_i^{(3)} + O(h^5) \tag{3.7}$$

and

$$f_{i+2} - f_{i-2} = 4hf_i' + \frac{8h^3}{3} f_i^{(3)} + O(h^5) \tag{3.8}$$

to cancel the $f_i^{(3)}$ terms, we have

$$f_i' = \frac{1}{12h}(f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}) + O(h^4). \tag{3.9}$$

We can, of course, make the accuracy even higher by including more points, but in many cases this is not good practice. For real problems, the derivatives at points close to the boundaries are important and need to be calculated accurately. The errors in the derivatives of the boundary points will accumulate in other points when the scheme is used to integrate an equation. The more points involved in the expressions of the derivatives, the more difficulties we encounter in obtaining accurate derivatives at the boundaries. Another way to increase the accuracy is by decreasing the interval h. This is very practical on vector computers. The algorithms for first-order or second-order derivatives are usually fully vectorized, so a vector processor can calculate many points in just one computer clock cycle.

Approximate expressions for the second-order derivative can be obtained with different combinations of f_j. The *three-point formula* for the second-order derivative is given by the combination

$$f_{i+1} - 2f_i + f_{i-1} = h^2 f_i'' + O(h^4), \tag{3.10}$$

with the Taylor expansions of $f_{i\pm1}$ around x_i. Note that the third-order term with $f_i^{(3)}$ vanishes because of the cancellation in the combination. The above equation gives the second-order derivative as

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2). \tag{3.11}$$

Similarly, we can combine the expansions of $f_{i\pm2}$ and $f_{i\pm1}$ around x_i and f_i to cancel the f_i', $f_i^{(3)}$, $f_i^{(4)}$, and $f_i^{(5)}$ terms; then we have

$$f_i'' = \frac{1}{12h^2}(-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}) + O(h^4) \tag{3.12}$$

as the *five-point formula* for the second-order derivative. The difficulty in dealing with the points around the boundaries still remains. We can use the interpolation formulas that we developed in the Chapter 2 to extrapolate the derivatives to the boundary points. The following program shows an example of calculating the first-order and second-order derivatives with the three-point formulas.

```
// An example of evaluating the derivatives with the
// 3-point formulas for f(x)=sin(x).

import java.lang.*;
public class Deriv {
  static final int n = 100, m = 5;
  public static void main(String argv[]) {
    double[] x = new double[n+1];
    double[] f = new double[n+1];
    double[] f1 = new double[n+1];
    double[] f2 = new double[n+1];

 // Assign constants, data points, and function
    int k = 2;
    double h = Math.PI/(2*n);
    for (int i=0; i<=n; ++i) {
      x[i]  = h*i;
      f[i]  = Math.sin(x[i]);
    }

 // Calculate 1st-order and 2nd-order derivatives
    f1 = firstOrderDerivative(h, f, k);
    f2 = secondOrderDerivative(h, f, k);

 // Output the result in every m data points
    for (int i=0; i<=n; i+=m) {
      double df1 = f1[i]-Math.cos(x[i]);
      double df2 = f2[i]+Math.sin(x[i]);
      System.out.println("x = " + x[i]);
      System.out.println("f'(x) = " + f1[i]);
      System.out.println("Error in f'(x): " + df1);
      System.out.println("f''(x) = " + f2[i]);
      System.out.println("Error in f''(x): " + df2);
      System.out.println();
    }
  }

// Method for the 1st-order derivative with the 3-point
// formula.  Extrapolations are made at the boundaries.

  public static double[] firstOrderDerivative(double h,
    double f[], int k) {
    int n = f.length-1;
    double[] y = new double[n+1];
    double[] xl = new double[k+1];
    double[] fl = new double[k+1];
    double[] fr = new double[k+1];

// Evaluate the derivative at nonboundary points
    for (int i=1; i<n; ++i)
      y[i] = (f[i+1]-f[i-1])/(2*h);

 // Lagrange-extrapolate the boundary points
    for (int i=1; i<=(k+1); ++i) {
      xl[i-1] = h*i;
      fl[i-1] = y[i];
      fr[i-1] = y[n-i];
    }
    y[0] = aitken(0, xl, fl);
    y[n] = aitken(0, xl, fr);
    return y;
  }

// Method for the 2nd-order derivative with the 3-point
// formula.  Extrapolations are made at the boundaries.

  public static double[] secondOrderDerivative(double h,
    double[] f, int k) {
    int n = f.length-1;
    double[] y = new double[n+1];
    double[] xl = new double[k+1];
    double[] fl = new double[k+1];
    double[] fr = new double[k+1];

// Evaluate the derivative at nonboundary points
    for (int i=1; i<n; ++i) {
      y[i] = (f[i+1]-2*f[i]+f[i-1])/(h*h);
    }

 // Lagrange-extrapolate the boundary points
    for (int i=1; i<=(k+1); ++i) {
      xl[i-1] = h*i;
      fl[i-1] = y[i];
      fr[i-1] = y[n-i];
    }
    y[0] = aitken(0, xl, fl);
    y[n] = aitken(0, xl, fr);
    return y;
  }

  public static double aitken(double x, double xi[],
    double fi[]) {...}
}
```

Table 3.1. *Derivatives obtained in the example*

x	f'	$\Delta f'$	f''	$\Delta f''$
0	0.999 959	−0.000 041	0.000 004	0.000 004
$\pi/10$	0.951 017	−0.000 039	−0.309 087	−0.000 070
$\pi/5$	0.808 985	−0.000 032	−0.587 736	0.000 049
$3\pi/10$	0.587 762	−0.000 023	−0.809 013	0.000 004
$2\pi/5$	0.309 003	−0.000 014	−0.951 055	0.000 001
$\pi/2$	−0.000 004	−0.000 004	−0.999 980	0.000 020

We have taken a simple function $f(x) = \sin x$, given at 101 discrete points with evenly spaced intervals in the region $[0, \pi/2]$. The Lagrange interpolation is applied to extrapolate the derivatives at the boundary points. The numerical results are summarized in Table 3.1, together with their errors. Note that the extrapolated data are of the same order of accuracy as other calculated values for f' and f'' at both $x = 0$ and $x = \pi/2$ because the three-point Lagrange interpolation scheme is accurate to a quadratic behavior. The functions $\sin x$ and $\cos x$ are well approximated by a linear or a quadratic curve at those two points.

In practice, we may encounter two problems with the formulas used above. The first problem is that we may not have the data given at uniform data points. One solution to such a problem is to perform an interpolation of the data first and then apply the above formulas to the function at the uniform data points generated from the interpolation. This approach can be tedious and has errors from two sources, the interpolation and the formulas above. The easiest solution to the problem is to adopt formulas that are suitable for nonuniform data points. If we use the Taylor expansion

$$f(x_{i\pm 1}) = f(x_i) + (x_{i\pm 1} - x_i)f'(x_i) + \frac{1}{2!}(x_{i\pm 1} - x_i)^2 f''(x_i) + O(h^3) \qquad (3.13)$$

and a combination of f_{i-1}, f_i, and f_{i+1} to cancel the second-order terms, we obtain

$$f'_i = \frac{h_{i-1}^2 f_{i+1} + \left(h_i^2 - h_{i-1}^2\right) f_i - h_i^2 f_{i-1}}{h_i h_{i-1}(h_i + h_{i-1})} + O(h^2), \qquad (3.14)$$

where $h_i = x_{i+1} - x_i$ and h is the larger of $|h_{i-1}|$ and $|h_i|$. This is the three-point formula for the first-order derivative in the case of nonuniform data points. Note that the accuracy here is the same as for the uniform data points. This is a better choice than interpolating the data first because the formula can be implemented in almost the same manner as in the case of the uniform data points. The following method returns the first-order derivative for such a situation.

```
// Method to calculate the 1st-order derivative with the
// nonuniform 3-point formula.  Extrapolations are made
// at the boundaries.

  public static double[] firstOrderDerivative2
    (double x[], double f[], int k) {
    int n = x.length-1;
    double[] y = new double[n+1];
    double[] xl = new double[k+1];
    double[] fl = new double[k+1];
    double[] xr = new double[k+1];
    double[] fr = new double[k+1];

 // Calculate the derivative at the field points
    double h0 = x[1]-x[0];
    double a0 = h0*h0;
    for (int i=1; i<n; ++i) {
      double h = x[i+1]-x[i];
      double a = h*h;
      double b = a-a0;
      double c = h*h0*(h+h0);
      y[i] = (a0*f[i+1]+b*f[i]-a*f[i-1])/c;
      h0 = h;
      a0 = a;
    }

 // Lagrange-extrapolate the boundary points
    for (int i=1; i<=(k+1); ++i) {
      xl[i-1] = x[i]-x[0];
      fl[i-1] = y[i];
      xr[i-1] = x[n]-x[n-i];
      fr[i-1] = y[n-i];
    }
    y[0] = aitken(0, xl, fl);
    y[n] = aitken(0, xr, fr);
    return y;
  }

  public static double aitken(double x, double xi[],
    double fi[]) {...}
```

The corresponding three-point formula for the second-order derivative can be derived with a combination of f_{i-1}, f_i, and f_{i+1} to have the first-order terms in the Taylor expansion removed, and we have

$$f_i'' = \frac{2[h_{i-1}f_{i+1} - (h_i + h_{i-1})f_i + h_i f_{i-1}]}{h_i h_{i+1}(h_i + h_{i-1})} + O(h). \tag{3.15}$$

Note that the accuracy here is an order lower than in the case of uniform data points because the third-order terms in the Taylor expansion are not canceled completely at the same time. However, the reality is that the third-order terms are partially canceled depending on how close the data points are to being uniform. We need to be careful in applying the three-point formula above. If higher accuracy is needed, we can resort to a corresponding five-point formula instead.

The second problem is that the accuracy cannot be controlled during evaluation with the three-point or five-point formulas. If the function $f(x)$ is available continuously, that is, for any given x in the region of interest, the accuracy in the numerical evaluation of the derivatives can be systematically improved to any desired level through an adaptive scheme. The basic strategy here is to apply the combination of

$$\Delta_1(h) = \frac{f(x+h) - f(x-h)}{2h} \tag{3.16}$$

and $\Delta_1(h/2)$ to remove the next nonzero term in the Taylor expansion and to repeat this process over and over. For example,

$$\Delta_1(h) - 4\Delta_1(h/2) = -3f'(x) + O(h^4). \tag{3.17}$$

Then we can use the difference in the evaluations of $f'(x)$ from $\Delta_1(h)$ and $\Delta_1(h/2)$ to set up a rough criterion for the desired accuracy. The following program is such an implementation.

```
// An example of evaluating 1st-order derivative through
// the adaptive scheme.

import java.lang.*;
public class Deriv3 {
  public static void main(String argv[]) {
    double del = 1e-6;
    int n = 10, m = 10;
    double h = Math.PI/(2*n);

 // Evaluate the derivative and output the result
    int k = 0;
    for (int i=0; i<=n; ++i) {
      double x = h*i;
      double d = (f(x+h)-f(x-h))/(2*h);
      double f1 = firstOrderDerivative3(x, h, d, del, k, m);
      double df1 = f1-Math.cos(x);
      System.out.println("x = " + x);
      System.out.println("f'(x) = " + f1);
      System.out.println("Error in f'(x): " + df1);
    }
  }

// Method to carry out 1st-order derivative through the
// adaptive scheme.

  public static double firstOrderDerivative3(double x,
    double h, double d, double del, int step,
    int maxstep) {
    step++;
    h = h/2;
    double d1 = (f(x+h)-f(x-h))/(2*h);
    if (step >= maxstep) {
      System.out.println ("Not converged after "
        + step + " recursions");
      return d1;
    }
    else {
      if ((h*h*Math.abs(d-d1)) < del) return (4*d1-d)/3;
      else return firstOrderDerivative3(x, h, d1, del,
        step, maxstep);
    }
  }

  public static double f(double x) {
    return Math.sin(x);
  }
}
```

Running the above program we obtain the first-order derivative of the function accurate to the order of the tolerance set in the program.

In fact, we can analytically work out a recursion by combining more $\Delta_1(h/2^k)$ for $k = 0, 1, \ldots$. For example, using the combination

$$\Delta_1(h) - 20\Delta_1(h/2) + 64\Delta_1(h/4) = 45f'(x) + O(h^6), \tag{3.18}$$

we cancel the fourth-order terms in the Taylor expansion. Note that the fifth-order term is also canceled automatically because of the symmetry in $\Delta_1(h/2^k)$. This process can be repeated analytically; this scheme is called the Richardson extrapolation. The derivation and implementation of the Richardson extrapolation are left as an exercise for the reader.

A similar adaptive scheme can be devised for the second-order derivative. For example, if we define the function

$$\Delta_2(h) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}, \tag{3.19}$$

and then use $\Delta_2(h)$ and $\Delta_2(h/2)$ to remove the next nonzero terms in the Taylor expansion, we have

$$\Delta_2(h) - 4\Delta_2(h/2) = -3f''(x) + O(h^4). \tag{3.20}$$

Based on the above equation and further addition of the terms involving $\Delta_2(h/2^k)$, we can come up with exactly the same adaptive scheme and the Richardson extrapolation for the second-order derivative.

3.2 Numerical integration

Let us turn to numerical integration. In general, we want to obtain the numerical value of an integral, defined in the region $[a, b]$,

$$S = \int_a^b f(x)\,dx. \tag{3.21}$$

We can divide the region $[a, b]$ into n slices, evenly spaced with an interval h. If we label the data points as x_i with $i = 0, 1, \ldots, n$, we can write the entire integral as a summation of integrals, with each over an individual slice,

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,dx. \tag{3.22}$$

If we can develop a numerical scheme that evaluates the summation over several slices accurately, we will have solved the problem. Let us first consider each slice separately. The simplest quadrature is obtained if we approximate $f(x)$ in the region $[x_i, x_{i+1}]$ linearly, that is, $f(x) \simeq f_i + (x - x_i)(f_{i+1} - f_i)/h$. After integrating over every slice with this linear function, we have

$$S = \frac{h}{2} \sum_{i=0}^{n-1} (f_i + f_{i+1}) + O(h^2), \tag{3.23}$$

where $O(h^2)$ comes from the error in the linear interpolation of the function. The above quadrature is commonly referred to as the *trapezoid rule*, which has an overall accuracy up to $O(h^2)$.

We can obtain a quadrature with a higher accuracy by working on two slices together. If we apply the Lagrange interpolation to the function $f(x)$ in the region $[x_{i-1}, x_{i+1}]$, we have

$$\begin{aligned} f(x) = {} & \frac{(x - x_i)(x - x_{i+1})}{(x_{i-1} - x_i)(x_{i-1} - x_{i+1})} f_{i-1} + \frac{(x - x_{i-1})(x - x_{i+1})}{(x_i - x_{i-1})(x_i - x_{i+1})} f_i \\ & + \frac{(x - x_{i-1})(x - x_i)}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)} f_{i+1} + O(h^3). \end{aligned} \tag{3.24}$$

If we carry out the integration for every pair of slices together with the integrand given from the above equation, we have

$$S = \frac{h}{3} \sum_{j=0}^{n/2-1} (f_{2j} + 4f_{2j+1} + f_{2j+2}) + O(h^4), \tag{3.25}$$

which is known as the *Simpson rule*. The third-order term vanishes because of cancelation. In order to pair up all the slices, we have to have an even number of slices. What happens if we have an odd number of slices, or an even number of points in $[a, b]$? One solution is to isolate the last slice and we then have

$$\int_{b-h}^{b} f(x)\,dx = \frac{h}{12}(-f_{n-2} + 8f_{n-1} + 5f_n). \tag{3.26}$$

The expression for $f(x)$ in Eq. (3.24), constructed from the last three points of the function, has been used in order to obtain the above result. The following program is an implementation of the Simpson rule for calculating an integral.

```
// An example of evaluating an integral with the Simpson
// rule over f(x)=sin(x).

import java.lang.*;
public class Integral {
  static final int n = 8;
  public static void main(String argv[]) {
    double f[] = new double[n+1];
    double h = Math.PI/(2.0*n);
    for (int i=0; i<=n; ++i) {
      double x = h*i;
      f[i] = Math.sin(x);
    }
    double s = simpson(f, h);
    System.out.println("The integral is: " + s);
  }

// Method to achieve the evenly spaced Simpson rule.

  public static double simpson(double y[], double h) {
    int n = y.length-1;
    double s0 = 0, s1 = 0, s2 = 0;
    for (int i=1; i<n; i+=2) {
      s0 += y[i];
      s1 += y[i-1];
      s2 += y[i+1];
    }
    double s = (s1+4*s0+s2)/3;

 // Add the last slice separately for an even n+1
    if ((n+1)%2 == 0)
      return h*(s+(5*y[n]+8*y[n-1]-y[n-2])/12);
    else
      return h*s;
  }
}
```

We have used $f(x) = \sin x$ as the integrand in the above example program and $[0, \pi/2]$ as the integration region. The output of the above program is 1.000 008, which has six digits of accuracy compared with the exact result 1. Note that we have used only nine mesh points to reach such a high accuracy.

In some cases, we may not have the integrand given at uniform data points. The Simpson rule can easily be generalized to accommodate cases with nonuniform data points. We can rewrite the interpolation in Eq. (3.24) as

$$f(x) = ax^2 + bx + c, \tag{3.27}$$

where

$$a = \frac{h_{i-1} f_{i+1} - (h_{i-1} + h_i) f_i + h_i f_{i-1}}{h_{i-1} h_i (h_{i-1} + h_i)}, \tag{3.28}$$

$$b = \frac{h_{i-1}^2 f_{i+1} + \left(h_i^2 - h_{i-1}^2\right) f_i - h_i^2 f_{i-1}}{h_{i-1} h_i (h_{i-1} + h_i)}, \tag{3.29}$$

$$c = f_i, \tag{3.30}$$

with $h_i = x_{i+1} - x_i$. We have taken $x_i = 0$ because the integral

$$S_i = \int_{x_{i-1}}^{x_{i+1}} f(x)\, dx \tag{3.31}$$

is independent of the choice of the origin of the coordinates. Then we have

$$S_i = \int_{-h_{i-1}}^{h_i} f(x)\, dx = \alpha f_{i+1} + \beta f_i + \gamma f_{i-1}, \tag{3.32}$$

where

$$\alpha = \frac{2h_i^2 + h_i h_{i-1} - h_{i-1}^2}{6h_i}, \tag{3.33}$$

$$\beta = \frac{(h_i + h_{i-1})^3}{6h_i h_{i-1}}, \tag{3.34}$$

$$\gamma = \frac{-h_i^2 + h_i h_{i-1} + 2h_{i-1}^2}{6h_i}. \tag{3.35}$$

The last slice needs to be treated separately if $n + 1$ is even, as with the case of uniform data points. Then we have

$$S_n = \int_0^{h_{n-1}} f(x)\,dx = \alpha f_n + \beta f_{n-1} + \gamma f_{n-2}, \tag{3.36}$$

where

$$\alpha = \frac{h_{n-1}}{6}\left(3 - \frac{h_{n-1}}{h_{n-1} + h_{n-2}}\right), \tag{3.37}$$

$$\beta = \frac{h_{n-1}}{6}\left(3 + \frac{h_{n-1}}{h_{n-2}}\right), \tag{3.38}$$

$$\gamma = -\frac{h_{n-1}}{6}\frac{h_{n-1}^2}{h_{n-2}(h_{n-1} + h_{n-2})}. \tag{3.39}$$

The equations appear quite tedious but implementing them in a program is quite straightforward following the program for the case of uniform data points.

Even though we can make an order-of-magnitude estimate of the error occurring in either the trapezoid rule or the Simpson rule, it is not possible to control it because of the uncertainty involved in the associated coefficient. We can, however, develop an adaptive scheme based on either the trapezoid rule or the Simpson rule to make the error in the evaluation of an integral controllable. Here we demonstrate such a scheme with the Simpson rule and leave the derivation of a corresponding scheme with the trapezoid rule to Exercise 3.9.

If we expand the integrand $f(x)$ in a Taylor series around $x = a$, we have

$$\begin{aligned} S &= \int_a^b f(x)\,dx \\ &= \int_a^b \left[\sum_{k=0}^{\infty} \frac{(x-a)^k}{k!} f^{(k)}(a)\right] dx \\ &= \sum_{k=0}^{\infty} \frac{h^{k+1}}{(k+1)!} f^{(k)}(a), \end{aligned} \tag{3.40}$$

where $h = b - a$. If we apply the Simpson rule with $x_{i-1} = a$, $x_i = c = (a + b)/2$, and $x_{i+1} = b$, we have the zeroth-level approximation

$$\begin{aligned} S_0 &= \frac{h}{6}\left[f(a) + 4f(c) + f(b)\right] \\ &= \frac{h}{6}\left[f(a) + \sum_{k=0}^{\infty} \frac{h^k}{k!}\left(\frac{1}{2^{k-2}} - 1\right) f^{(k)}(a)\right]. \end{aligned} \tag{3.41}$$

We have expanded both $f(c)$ and $f(b)$ in a Taylor series around $x = a$ in the above equation. Now if we take the difference between S and S_0 and keep only the leading term, we have

$$\Delta S_0 = S - S_0 \approx -\frac{h^5}{2880} f^{(4)}(a). \tag{3.42}$$

We can continue to apply the Simpson rule in the regions $[a, c]$ and $[c, b]$. Then we obtain the first-level approximation

$$S_1 = \frac{h}{12}\left[f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)\right], \tag{3.43}$$

where $d = (a + c)/2$ and $e = (c + b)/2$, and

$$\begin{aligned}\Delta S_1 &= S - S_1 \\ &\approx -\frac{(h/2)^5}{2880} f^{(4)}(a) - \frac{(h/2)^5}{2880} f^{(4)}(c) \\ &\approx -\frac{1}{2^4}\frac{h^5}{2880} f^{(4)}(a).\end{aligned} \tag{3.44}$$

We have used that $f^{(4)}(a) \approx f^{(4)}(c)$. The difference between the first-level and zeroth-level approximations is

$$S_1 - S_0 \approx -\frac{15}{16}\frac{h^5}{2880} f^{(4)}(a) \approx \frac{15}{16}\Delta S_0 \approx 15\Delta S_1. \tag{3.45}$$

The above result can be used to set up the criterion for the error control in an adaptive algorithm. Consider, for example, that we want the error $|\Delta S| \leq \delta$. First we can carry out S_0 and S_1. Then we check whether $|S_1 - S_0| \leq 15\delta$, that is, whether $|\Delta S_1| \leq \delta$. If it is true, we return S_1 as the approximation for the integral. Otherwise, we continue the adaptive process to divide the region into two halves, four quarters, and so on, until we reach the desired accuracy with $|S - S_n| \leq \delta$, where S_n is the nth-level approximation of the integral. Let us here take the evaluation of the integral

$$S = \int_0^{\pi} \frac{[1 + a_0(1 - \cos x)^2]\,dx}{(1 + a_0 \sin^2 x)\sqrt{1 + 2a_0(1 - \cos x)}}, \tag{3.46}$$

for any $a_0 \geq 0$, as an example. The following program is an implementation of the adaptive Simpson rule for the integral above.

```
// An example of evaluating an integral with the adaptive
// Simpson rule.

import java.lang.*;
public class Integral2 {
  static final int n = 100;
  public static void main(String argv[]) {
    double del = 1e-6;
    int k = 0;
    double a = 0;
    double b = Math.PI;
    double s = simpson2(a, b, del, k, n);
    System.out.println ("S = " + s);
  }

// Method to integrate over f(x) with the adaptive
// Simpson rule.

  public static double simpson2(double a, double b,
    double del, int step, int maxstep) {
    double h = b-a;
    double c = (b+a)/2;
    double fa = f(a);
    double fc = f(c);
    double fb = f(b);
    double s0 = h*(fa+4*fc+fb)/6;
    double s1 = h*(fa+4*f(a+h/4)+2*fc
      + 4*f(a+3*h/4)+fb)/12;
    step++;
    if (step >= maxstep) {
      System.out.println ("Not converged after "
        + step + " recursions");
      return s1;
    }
    else {
      if (Math.abs(s1-s0) < 15*del) return s1;
      else return simpson2(a, c, del/2, step, maxstep)
        + simpson2(c, b, del/2, step, maxstep);
    }
  }

// Method to provide the integrand f(x).

  public static double f(double x) {
    double a0 = 5;
    double s = Math.sin(x);
    double c = Math.cos(x);
    double f = (1+a0*(1-c)*(1-c))
      /((1+a0*s*s)*Math.sqrt(1+2*a0*(1-c)));
    return f;
  }
}
```

After running the program, we obtain $S \simeq 3.141\,592\,78$. Note that $a_0 = 5$ is assigned in the above program. In fact, we can assign a random value to a_0 and the result will remain the same within the errorbar. This is because the above integral $S \equiv \pi$, independent of the specific value of a_0. The integral showed up while solving a problem in Jackson (1999). It is quite intriguing because the integrand has such a complicated dependence on a_0 and x and yet the result is so simple and cannot be obtained from any table or symbolic system. Try it and see whether an analytical solution can be found easily. One way to do it is to show first that the integral is independent of a_0 and then set $a_0 = 0$.

The adaptive scheme presented above provides a way of evaluating an integral accurately. It is, however, limited to cases with an integrand that is given continuously in the integration region. For problems that involve an integrand given at discrete points, the strength of the adaptive scheme is reduced because the data must be interpolated and the interpolation may destroy the accuracy in the adaptive scheme. Adaptive schemes can also be slow if a significant number of levels are needed. In determining whether to use an adaptive scheme or not, we should consider how critical and practical the accuracy sought in the evaluation is against the speed of computation and possible errors due to other factors involved. Sometimes the adaptive method introduced above is the only viable numerical approach if a definite answer is sought, especially when an analytic result does not exist or is difficult to find.

3.3 Roots of an equation

In physics, we often encounter situations in which we need to find the possible value of x that ensures the equation $f(x) = 0$, where $f(x)$ can either be an explicit or an implicit function of x. If such a value exists, we call it a *root* or *zero* of the equation. In this section, we will discuss only single-variable problems and leave the discussion of multivariable cases to Chapter 5, after we have gained some basic knowledge of matrix operations.

Bisection method

If we know that there is a root $x = x_r$ in the region $[a, b]$ for $f(x) = 0$, we can use the *bisection method* to find it within a required accuracy. The bisection method is the most intuitive method, and the idea is very simple. Because there is a root in the region, $f(a)f(b) < 0$. We can divide the region into two equal parts with $x_0 = (a + b)/2$. Then we have either $f(a)f(x_0) < 0$ or $f(x_0)f(b) < 0$. If $f(a)f(x_0) < 0$, the next trial value is $x_1 = (a + x_0)/2$; otherwise, $x_1 = (x_0 + b)/2$. This procedure is repeated until the improvement on x_k or $|f(x_k)|$ is less than the given tolerance δ.

Let us take $f(x) = e^x \ln x - x^2$ as an example to illustrate how the bisection method works. We know that when $x = 1$, $f(x) = -1$ and when $x = 2$, $f(x) = e^2 \ln 2 - 4 \approx 1$. So there is at least one value $x_r \in [1, 2]$ that would make $f(x_r) = 0$. In the neighborhood of x_r, we have $f(x_r + \delta) > 0$ and $f(x_r - \delta) < 0$.

```
// An example of searching for a root via the bisection
// method for f(x)=exp(x)*ln(x)-x*x=0.

import java.lang.*;
public class Bisect {
  public static void main(String argv[]) {
    double x = 0, del = 1e-6, a = 1, b = 2;
    double dx = b-a;
    int k = 0;
    while (Math.abs(dx) > del) {
      x = (a+b)/2;
      if ((f(a)*f(x)) < 0) {
        b  = x;
        dx = b-a;
      }
      else {
        a = x;
        dx = b-a;
      }
      k++;
    }
    System.out.println("Iteration number: " + k);
    System.out.println("Root obtained: " + x);
    System.out.println("Estimated error: " + dx);
  }

// Method to provide function f(x)=exp(x)*log(x)-x*x.

  public static double f(double x) {
    return Math.exp(x)*Math.log(x)-x*x;
  }
}
```

The above program gives the root $x_r = 1.694\,601 \pm 0.000\,001$ after only 20 iterations. The error comes from the final improvement in the search. This error can be reduced to a lower value by making the tolerance δ smaller.

The Newton method

This method is based on linear approximation of a smooth function around its root. We can formally expand the function $f(x_r) = 0$ in the neighborhood of the root x_r through the Taylor expansion

$$f(x_r) \simeq f(x) + (x_r - x)f'(x) + \cdots = 0, \tag{3.47}$$

where x can be viewed as a trial value for the root of x_r at the kth step and the approximate value of the next step x_{k+1} can be derived from

$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k) \simeq 0, \tag{3.48}$$

that is,

$$x_{k+1} = x_k + \Delta x_k = x_k - f_k/f'_k, \tag{3.49}$$

with $k = 0, 1, \ldots$. Here we have used the notation $f_k = f(x_k)$. The above iterative scheme is known as the *Newton method*. It is also referred to as the *Newton–Raphson method* in the literature. The above equation is equivalent to approximating the root by drawing a tangent to the curve at the point x_k and taking x_{k+1} as the tangent's intercept on the x axis. This step is repeated toward the root, as illustrated in Fig. 3.1. To see how this method works in a program, we again take the function $f(x) = e^x \ln x - x^2$ as an example. The following program is an implementation of the Newton method.

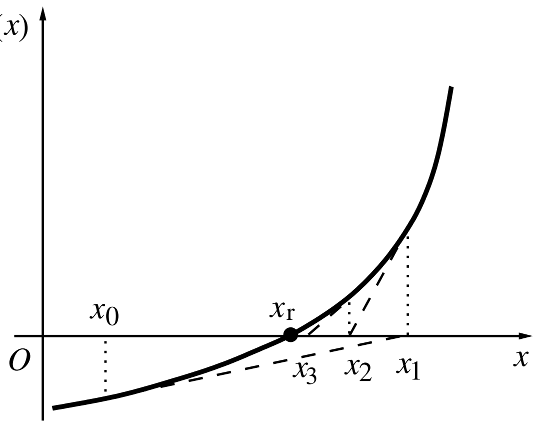


Fig. 3.1 A schematic illustration of the steps in the Newton method for searching for the root of $f(x) = 0$.

```
// An example of searching for a root via the Newton method
// for f(x)=exp(x)*ln(x)-x*x=0.

import java.lang.*;
public class Newton {
  public static void main(String argv[]) {
    double del = 1e-6, a = 1, b = 2;
    double dx = b-a, x=(a+b)/2;
    int k = 0;
    while (Math.abs(dx) > del) {
      dx = f(x)/d(x);
      x -= dx;
      k++;
    }
    System.out.println("Iteration number: " + k);
    System.out.println("Root obtained: " + x);
    System.out.println("Estimated error: " + dx);
  }

  public static double f(double x) {...}

// Method to provide the derivative f'(x).

  public static double d(double x) {
    return Math.exp(x)*(Math.log(x)+1/x)-2*x;
  }
}
```

The above program gives the root $x_r = 1.694\,600\,92$ after only five iterations. It is clear that the Newton method is more efficient, because the error is now much smaller even though we have started the search at exactly the same point, and have gone through a much smaller number of steps. The reason is very simple. Each new step size $|\Delta x_k| = |x_{k+1} - x_k|$ in the Newton method is determined according to Eq. (3.49) by the ratio of the value and the slope of the function $f(x)$ at x_k. For the same function value, a large step is created for the small-slope case, whereas a small step is made for the large-slope case. This ensures an efficient update and also avoids possible overshooting. There is no such mechanism in the bisection method.

Secant method

In many cases, especially when $f(x)$ has an implicit dependence on x, an analytic expression for the first-order derivative needed in the Newton method may not exist or may be very difficult to obtain. We have to find an alternative scheme to achieve a similar algorithm. One way to do this is to replace f'_k in Eq. (3.49) with the two-point formula for the first-order derivative, which gives

$$x_{k+1} = x_k - (x_k - x_{k-1})f_k/(f_k - f_{k-1}). \tag{3.50}$$

This iterative scheme is commonly known as the *secant method*, or the *discrete Newton method*. The disadvantage of the method is that we need two points in order to start the search process. The advantage of the method is that $f(x)$ can now be implicitly given without the need for the first-order derivative. We can still use the function $f(x) = e^x \ln x - x^2$ as an example, in order to make a comparison.

```
// An example of searching for a root via the secant method
// for f(x)=exp(x)*ln(x)-x*x=0.

import java.lang.*;
public class Root {
  public static void main(String argv[]) {
    double del = 1e-6, a = 1, b = 2;
    double dx = (b-a)/10, x = (a+b)/2;
    int n = 6;
    x = secant(n, del, x, dx);
    System.out.println("Root obtained: " + x);
  }

// Method to carry out the secant search.

  public static double secant(int n, double del,
    double x, double dx) {
    int k = 0;
    double x1 = x+dx;
    while ((Math.abs(dx)>del) && (k<n)) {
      double d = f(x1)-f(x);
      double x2 = x1-f(x1)*(x1-x)/d;
      x  = x1;
      x1 = x2;
      dx = x1-x;
      k++;
    }
    if (k==n) System.out.println("Convergence not" +
      " found after " + n + " iterations");
    return x1;
  }

  public static double f(double x) {...}
}
```

The above program gives the root $x_r = 1.694\,601\,0$ after five iterations. As expected, the secant method is more efficient than the bisection method but less efficient than the Newton method, because of the two-point approximation of the first-order derivative. However, if the expression for the first-order derivative cannot easily be obtained, the secant method becomes very useful and powerful.

3.4 Extremes of a function

An associated problem to finding the root of an equation is finding the maxima and/or minima of a function. Examples of such situations in physics occur when considering the equilibrium position of an object, the potential surface of a field, and the optimized structures of molecules and small clusters. Here we consider mainly a function of a single variable, $g = g(x)$, and just touch on the multivariable case of $g = g(x_1, x_2, \ldots, x_l)$ with the steepest-descent method. Other schemes for the multivariable cases are left to later chapters.

Knowing the solution of a nonlinear equation $f(x) = 0$, we can develop numerical schemes to obtain minima or maxima of a function $g(x)$. We know that an extreme of $g(x)$ occurs at the point with

$$f(x) = \frac{dg(x)}{dx} = 0, \tag{3.51}$$

which is a minimum (maximum) if $f'(x) = g''(x)$ is greater (less) than zero. So all the root-search schemes discussed so far can be generalized here to search for the extremes of a single-variable function.

However, at each step of updating the value of x, we need to make a judgment as to whether $g(x_{k+1})$ is increasing (decreasing) if we are searching for a maximum (minimum) of the function. If it is, we accept the update. If it is not, we reverse the update; that is, instead of using $x_{k+1} = x_k + \Delta x_k$, we use $x_{k+1} = x_k - \Delta x_k$. With the Newton method, the increment is $\Delta x_k = -f_k/f'_k$, and with the secant method, the increment is $\Delta x_k = -(x_k - x_{k-1})f_k/(f_k - f_{k-1})$.

Let us illustrate these ideas with a simple example of finding the bond length of the diatomic molecule NaCl from the interaction potential between the two ions (Na^+ and Cl^- in this case). Assuming that the interaction potential is $V(r)$ when the two ions are separated by a distance r, the bond length r_{eq} is the equilibrium distance when $V(r)$ is at its minimum. We can model the interaction potential between Na^+ and Cl^- as

$$V(r) = -\frac{e^2}{4\pi\epsilon_0 r} + V_0 e^{-r/r_0}, \tag{3.52}$$

where e is the charge of a proton, ϵ_0 is the electric permittivity of vacuum, and V_0 and r_0 are parameters of this effective interaction. The first term in Eq. (3.52) comes from the Coulomb interaction between the two ions, but the second term is the result of the electron distribution in the system. We will use $V_0 = 1.09 \times 10^3$ eV, which is taken from the experimental value for solid NaCl (Kittel, 1995), and $r_0 = 0.330$ Å, which is a little larger than the corresponding parameter for solid NaCl ($r_0 = 0.321$ Å), because there is less charge screening in an isolated molecule. At equilibrium, the force between the two ions,

$$f(r) = -\frac{dV(r)}{dr} = -\frac{e^2}{4\pi\epsilon_0 r^2} + \frac{V_0}{r_0} e^{-r/r_0}, \tag{3.53}$$

is zero. Therefore, we search for the root of $f(x) = dg(x)/dx = 0$, with $g(x) = -V(x)$. We will force the algorithm to move toward the minimum of $V(r)$. The following program is an implementation of the algorithm with the secant method to find the bond length of NaCl.

```
// An example of calculating the bond length of NaCl.

import java.lang.*;
public class Bond {
  static final double e2 = 14.4, v0 = 1090, r0 = 0.33;
  public static void main(String argv[]) {
    double del = 1e-6, r = 2, dr = 0.1;
    int n = 20;
    r = secant2(n, del, r, dr);
    System.out.println("The bond length is " + r +
      " angstroms");
  }

// Method to carry out the secant search for the
// maximum of g(x) via the root of f(x)=dg(x)/dx=0.

  public static double secant2(int n, double del,
    double x, double dx) {
    int k = 0;
    double x1 = x+dx, x2 = 0;
    double g0 = g(x);
    double g1 = g(x1);
    if (g1 > g0) x1 = x-dx;
    while ((Math.abs(dx)>del) && (k<n)) {
      double d = f(x1)-f(x);
      dx = -(x1-x)*f(x1)/d;
      x2 = x1+dx;
      double g2 = g(x2);
      if (g2 > g1) x2 = x1-dx;
      x  = x1;
      x1 = x2;
      g1 = g2;
      k++;
    }
    if (k==n) System.out.println("Convergence not" +
      " found after " + n + " iterations");
    return x1;
  }
// Method to provide function g(x)=-e2/x+v0*exp(-x/r0).

  public static double g(double x) {
    return -e2/x+v0*Math.exp(-x/r0);
  }

// Method to provide function f(x)=-dg(x)/dx.

  public static double f(double x) {
    return -e2/(x*x)+v0*Math.exp(-x/r0)/r0;
  }
}
```

The bond length obtained from the above program is $r_{\text{eq}} = 2.36$ Å. We have used $e^2/4\pi\epsilon_0 = 14.4$ Å eV for convenience. The method for searching for the minimum is modified slightly from the secant method used in the preceding section in order to force the search to move toward the minimum of $g(x)$. We will still obtain the same result as with the secant method used in the earlier example for this simple problem, because there is only one minimum of $V(x)$ around the point where we start the search. The other minimum of $V(x)$ is at $x = 0$ which is also a singularity. For a more general $g(x)$, modifications introduced in the above program are necessary.

Another relevant issue is that in many cases we do not have the explicit function $f(x) = g'(x)$ if $g(x)$ is not an explicit function of x. However, we can construct the first-order and second-order derivatives numerically from the two-point or three-point formulas, for example.

In the example program above, the search process is forced to move along the direction of descending the function $g(x)$ when looking for a minimum. In other words, for $x_{k+1} = x_k + \Delta x_k$, the increment Δx_k has the sign opposite to $g'(x_k)$. Based on this observation, an update scheme can be formulated as

$$x_{k+1} = x_k + \Delta x_k = x_k - a g'(x_k), \tag{3.54}$$

with a being a positive, small, and adjustable parameter. This scheme can be generalized to the multivariable case as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k = \mathbf{x}_k - a\boldsymbol{\nabla}g(\mathbf{x}_k)/|\boldsymbol{\nabla}g(\mathbf{x}_k)|, \tag{3.55}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_l)$ and $\boldsymbol{\nabla}g(\mathbf{x}) = (\partial g/\partial x_1, \partial g/\partial x_2, \ldots, \partial g/\partial x_l)$.

Note that step $\Delta\mathbf{x}_k$ here is scaled by $|\boldsymbol{\nabla}g(\mathbf{x}_k)|$ and is forced to move toward the direction of the steepest descent. This is why this method is known as the *steepest-descent method*. The following program is an implementation of such a scheme to search for the minimum of the function $g(x, y) = (x - 1)^2 e^{-y^2} + y(y + 2) e^{-2x^2}$ around $x = 0.1$ and $y = -1$.

```
// An example of searching for a minimum of a multivariable
// function through the steepest-descent method.

import java.lang.*;
public class Minimum {
  public static void main(String argv[]) {
  double del = 1e-6, a = 0.1;
    double x[] = new double[2];
    x[0] = 0.1;
    x[1] =  -1;
    steepestDescent(x, a, del);
    System.out.println("The minimum is at"
      + " x= " + x[0] +", y= " +x[1]);
  }

// Method to carry out the steepest-descent search.

  public static void steepestDescent(double x[],
    double a, double del) {
    int n = x.length;
    double h = 1e-6;
    double g0 = g(x);
    double fi[] = new double[n];
    fi = f(x, h);
    double dg = 0;
    for (int i=0; i<n; ++i) dg += fi[i]*fi[i];
    dg = Math.sqrt(dg);
    double b = a/dg;
    while (dg > del) {
      for (int i=0; i<n; ++i) x[i] -= b*fi[i];
      h /= 2;
      fi = f(x, h);
      dg = 0;
      for (int i=0; i<n; ++i) dg += fi[i]*fi[i];
      dg = Math.sqrt(dg);
      b  = a/dg;
      double g1 = g(x);
      if (g1 > g0) a /= 2;
      else g0 = g1;
    }
  }

// Method to provide the gradient of g(x).

  public static double[] f(double x[], double h) {
    int n = x.length;
    double z[] = new double[n];
    double y[] = (double[]) x.clone();
    double g0 = g(x);
    for (int i=0; i<n; ++i) {
      y[i] += h;
      z[i] = (g(y)-g0)/h;
    }
    return z;
  }

// Method to provide function g(x).

  public static double g(double x[]) {
    return (x[0]-1)*(x[0]-1)*Math.exp(-x[1]*x[1])
      +x[1]*(x[1]+2)*Math.exp(-2*x[0]*x[0]);
  }
}
```

Note that the spacing in the two-point formula for the derivative is reduced by a factor of 2 after each search, so is the step size in case of overshooting. After running the program, we find the minimum is at $x \simeq 0.107\,355$ and $y \simeq -1.223\,376$. This is a very simple but not very efficient scheme. Like many other optimization schemes, it converges to a local minimum near the starting point. The search for a global minimum or maximum of a multivariable function is a nontrivial task, especially when the function contains a significant number of local minima or maxima. Active research is still looking for better and more reliable schemes. In the last few decades, several advanced methods have been introduced for dealing with function optimization, most noticeably, the simulated annealing scheme, the Monte Carlo method, and the genetic algorithm and programming. We will discuss some of these more advanced topics later in the book.

3.5 Classical scattering

Scattering is a very important process in physics. From systems at the microscopic scale, such as protons and neutrons in nuclei, to those at the astronomical scale, such as galaxies and stars, scattering processes play a crucial role in determining their structures and dynamics. In general, a many-body process can be viewed as a sum of many simultaneous two-body scattering events if coherent scattering does not happen.

In this section, we will apply the computational methods that we have developed in this and the preceding chapter to study the classical scattering of two particles, interacting with each other through a pairwise potential. Most scattering processes with realistic interaction potentials cannot be solved analytically. Therefore, numerical solutions of a scattering problem become extremely valuable if we want to understand the physical process of particle–particle interaction. We will assume that the interaction potential between the two particles is spherically symmetric. Thus the total angular momentum and energy of the system are conserved during the scattering.

The two-particle system

The Lagrangian for a general two-body system can be written as

$$\mathcal{L} = \frac{m_1}{2} v_1^2 + \frac{m_2}{2} v_2^2 - V(\mathbf{r}_1, \mathbf{r}_2), \tag{3.56}$$

where m_i, $\mathbf{r}_i$, and $v_i = |d\mathbf{r}_i/dt|$ with $i = 1, 2$ are, respectively, the mass, position vector, and speed of the ith particle, and V is the interaction potential between the two particles, which we take to be spherically symmetric, that is, $V(\mathbf{r}_1, \mathbf{r}_2) = V(r_{21})$, with $r_{21} = |\mathbf{r}_2 - \mathbf{r}_1|$ being the distance between the two particles.

We can always perform a coordinate transformation from $\mathbf{r}_1$ and $\mathbf{r}_2$ to the relative coordinate $\mathbf{r}$ and the center-of-mass coordinate $\mathbf{r}_c$ with

$$\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1, \tag{3.57}$$

$$\mathbf{r}_c = \frac{m_1\mathbf{r}_1 + m_2\mathbf{r}_2}{m_1 + m_2}. \tag{3.58}$$

Then we can express the Lagrangian of the system in terms of the new coordinates and their corresponding speeds as

$$\mathcal{L} = \frac{M}{2}v_c^2 + \frac{m}{2}v^2 - V(r), \tag{3.59}$$

where $r = r_{21}$ and $v = |d\mathbf{r}/dt|$ are the distance and relative speed between the two particles, $M = m_1 + m_2$ is the total mass of the system, $m = m_1m_2/(m_1 + m_2)$ is the reduced mass of the two particles, and $v_c = |d\mathbf{r}_c/dt|$ is the speed of the center of mass. If we study the scattering in the center-of-mass coordinate system with $\mathbf{v}_c = d\mathbf{r}_c/dt = \mathbf{0}$, the process is then represented by the motion of a single particle of mass m in a central potential $V(r)$. In general, a two-particle system with a spherically symmetric interaction can be viewed as a single particle with a reduced mass moving in a central potential that is identical to the interaction potential.

We can reach the same conclusion from Newton's equations

$$m_1\ddot{\mathbf{r}}_1 = \mathbf{f}_1, \tag{3.60}$$

$$m_2\ddot{\mathbf{r}}_2 = \mathbf{f}_2, \tag{3.61}$$

where the accelerations and forces are given by $\ddot{\mathbf{r}}_i = d^2\mathbf{r}_i/dt^2$ and $\mathbf{f}_i = -\nabla_i V(r_{21}) = -dV(r_{21})/d\mathbf{r}_i$. Note that, following the convention in physics, we have used two dots over a variable to denote the second-order time derivative of the variable, and we will also use a dot over a variable to denote the first-order time derivative of the variable. Adding the above two equations and using Newton's third law, $\mathbf{f}_1 = -\mathbf{f}_2$, or dividing the corresponding equation by m_i and then taking the difference, we obtain

$$m\ddot{\mathbf{r}} = \mathbf{f}(\mathbf{r}), \tag{3.62}$$

$$M\ddot{\mathbf{r}}_c = \mathbf{0}, \tag{3.63}$$

where $\mathbf{f}(\mathbf{r}) = -\nabla V(r) = -dV(r)/d\mathbf{r}$. So the motion of a two-particle system with an isotropic interaction is equivalent to the constant-velocity motion of the center of mass plus the relative motion of two particles that is described by an effective particle of mass m in a central potential $V(r)$.

Cross section of scattering

Now we only need to study the scattering process of a particle with a mass m in a central potential $V(r)$. Assume that the particle is coming in from the left with an impact parameter b, the shortest distance between the particle and the potential center if $V(r) \to 0$. A sketch of the process is given in Fig. 3.2.

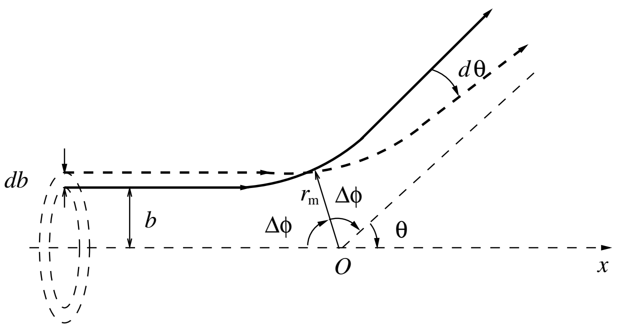


Fig. 3.2 A sketch of the scattering process of a particle in a central potential.

The total cross section of such a scattering process is given by

$$\sigma = \int \sigma(\theta)\, d\Omega, \tag{3.64}$$

where $\sigma(\theta)$ is the differential cross section, or the probability of a particle's being found in the solid angle element $d\Omega = 2\pi \sin\theta\, d\theta$ at the deflection angle θ.

If the particles are coming in with a flux density I (number of particles per unit cross-sectional area per unit time), then the number of particles per unit time within the range db of the impact parameter b is $2\pi I b\, db$. Because all the incoming particles in this area will go out in the solid angle element $d\Omega$ with the probability $\sigma(\theta)$, we have

$$2\pi I b\, db = I\sigma(\theta)\, d\Omega, \tag{3.65}$$

which gives the differential cross section as

$$\sigma(\theta) = \frac{b}{\sin\theta}\left|\frac{db}{d\theta}\right|. \tag{3.66}$$

The reason for taking the absolute value of $db/d\theta$ in the above equation is that $db/d\theta$ can be positive or negative depending on the form of the potential and the impact parameter. However, $\sigma(\theta)$ has to be positive because it is a probability. We can relate this center-of-mass cross section to the cross section measured in the laboratory through an inverse coordinate transformation of Eq. (3.57) and Eq. (3.58), which relates $\mathbf{r}$ and $\mathbf{r}_c$ back to $\mathbf{r}_1$ and $\mathbf{r}_2$. We will not discuss this transformation here; interested readers can find it in any standard advanced mechanics textbook.

Numerical evaluation of the cross section

Because the interaction between two particles is described by a spherically symmetric potential, the angular momentum and the total energy of the system are conserved during the scattering. Formally, we have

$$l = mbv_0 = mr^2\dot{\phi} \tag{3.67}$$

and

$$E = \frac{m}{2}v_0^2 = \frac{m}{2}(\dot{r}^2 + r^2\dot{\phi}^2) + V(r) \tag{3.68}$$

which are respectively the total momentum and total energy and which are constant. Here r is the radial coordinate, ϕ is the polar angle, and v_0 is the initial impact velocity. Combining Eq. (3.67) and Eq. (3.68) with

$$\frac{d\phi}{dr} = \frac{d\phi}{dt}\frac{dt}{dr}, \tag{3.69}$$

we obtain

$$\frac{d\phi}{dr} = \pm\frac{b}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}}, \tag{3.70}$$

which provides a relation between ϕ and r for the given E, b, and $V(r)$. Here the $+$ and $-$ signs correspond to two different but symmetric parts of the trajectory. The equation above can be used to calculate the deflection angle θ through

$$\theta = \pi - 2\Delta\phi, \tag{3.71}$$

where $\Delta\phi$ is the change in the polar angle when r changes from infinity to its minimum value r_{m}. From Eq. (3.70), we have

$$\begin{aligned} \Delta\phi &= b\int_{r_{\mathrm{m}}}^{\infty} \frac{dr}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}} \\ &= -b\int_{\infty}^{r_{\mathrm{m}}} \frac{dr}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}}. \end{aligned} \tag{3.72}$$

If we use the energy conservation and angular momentum conservation discussed earlier in Eq. (3.67) and Eq. (3.68), we can show that r_{m} is given by

$$1 - \frac{b^2}{r_{\mathrm{m}}^2} - \frac{V(r_{\mathrm{m}})}{E} = 0, \tag{3.73}$$

which is the result of zero r-component velocity, that is, $\dot{r} = 0$. Because of the change in the polar angle $\Delta\phi = \pi/2$ for $V(r) = 0$, we can rewrite Eq. (3.71) as

$$\theta = 2b\left[\int_b^{\infty} \frac{dr}{r^2\sqrt{1 - b^2/r^2}} - \int_{r_{\mathrm{m}}}^{\infty} \frac{dr}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}}\right]. \tag{3.74}$$

The real reason for rewriting the constant π as an integral in the above expression for θ is a numerical strategy to reduce possible errors coming from the truncation of the integration region at both ends of the second term. The integrand in the first integral diverges as $r \to b$ in much the same way as the integrand in the second integral does as $r \to r_{\mathrm{m}}$. The errors from the first and second terms cancel each other, at least partially, because they are of opposite signs.

Now we demonstrate how to calculate the differential cross section for a given potential. Let us take the Yukawa potential

$$V(r) = \frac{\kappa}{r} e^{-r/a} \tag{3.75}$$

as an illustrative example. Here κ and a are positive parameters that reflect, respectively, the range and the strength of the potential and can be adjusted. We use the secant method to solve Eq. (3.73) to obtain r_{m} for the given b and E. Then we use the Simpson rule to calculate the integrals in Eq. (3.74). After that, we apply the three-point formula for the first-order derivative to obtain $d\theta/db$. Finally, we put all these together to obtain the differential cross section of Eq. (3.66).

For simplicity, we choose $E = m = \kappa = 1$. The following program is an implementation of the scheme outlined above.

```
// An example of calculating the differential cross section
// of classical scattering on the Yukawa potential.

import java.lang.*;
public class Collide {
  static final int n = 10000, m = 20;
  static final double a = 100, e = 1;
  static double b;
  public static void main(String argv[]) {
    int nc = 20, ne = 2;
    double del = 1e-6, db = 0.5, b0 = 0.01, h = 0.01;
    double g1, g2;
    double theta[] = new double[n+1];
    double fi[] = new double[n+1];
    double sig[] = new double[m+1];
    for (int i=0; i<=m; ++i) {
      b = b0+i*db;

   // Calculate the first term of theta
      for (int j=0; j<=n; ++j) {
        double r = b+h*(j+1);
        fi[j] = 1/(r*r*Math.sqrt(fb(r)));
      }
      g1 = simpson(fi, h);

   // Find r_m from 1-b*b/(r*r)-V/E = 0
      double rm = secant(nc, del, b, h);

   // Calculate the second term of theta
      for (int j=0; j<=n; ++j) {
        double r = rm+h*(j+1);
        fi[j] = 1/(r*r*Math.sqrt(f(r)));
      }
      g2 = simpson(fi, h);
      theta[i] = 2*b*(g1-g2);
    }

 // Calculate d theta/d b
    sig = firstOrderDerivative(db, theta, ne);

 // Put the cross section in log form with the exact
```

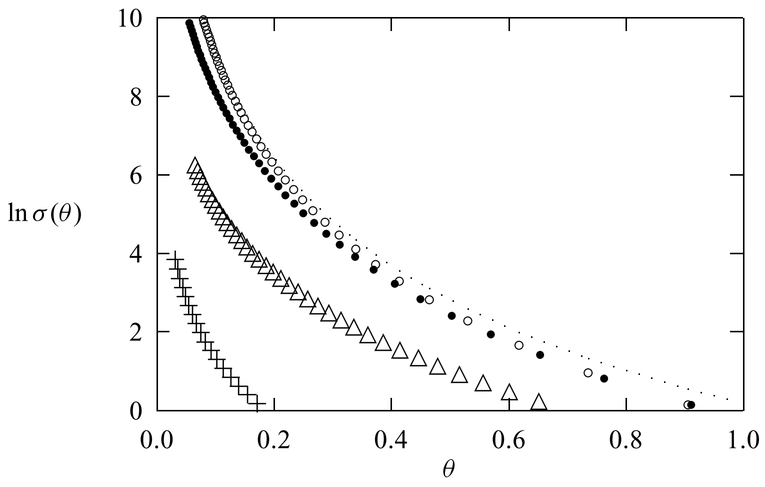


Fig. 3.3 This figure shows the differential cross section for scattering from the Yukawa potential with $a = 0.1$ (crosses), $a = 1$ (triangles), $a = 10$ (solid circles), and $a = 100$ (open circles), together with the analytical result for Coulomb scattering (dots). Other parameters used are $E = m = \kappa = 1$.

```
 // result of the Coulomb scattering (ruth)
    for (int i=m; i>=0; --i) {
      b = b0+i*db;
      sig[i] = b/(Math.abs(sig[i])*Math.sin(theta[i]));
      double ruth = 1/Math.pow(Math.sin(theta[i]/2),4);
      ruth /= 16;
      double si = Math.log(sig[i]);
      double ru = Math.log(ruth);
      System.out.println("theta = " + theta[i]);
      System.out.println("ln sigma(theta) = " + si);
      System.out.println("ln sigma_r(theta) = " + ru);
      System.out.println();
    }
  }

  public static double simpson(double y[], double h)
    {...}

  public static double secant(int n, double del,
    double x, double dx) {...}

  public static double[] firstOrderDerivative(double h,
    double f[], int m) {...}

  public static double aitken(double x, double xi[],
    double fi[]) {...}

// Method to provide function f(x) for the root search.

  public static double f(double x) {
    return 1-b*b/(x*x)-Math.exp(-x/a)/(x*e);
  }

// Method to provide function 1-b*b/(x*x).

  public static double fb(double x) {
    return 1-b*b/(x*x);
  }
}
```

The methods called in the program are exactly those given earlier in this the preceding chapter. We have also included the analytical result for Coulomb scattering, which is a special case of the Yukawa potential with $a \to \infty$. The differential cross section for Coulomb scattering is

$$\sigma(\theta) = \left(\frac{\kappa}{4E}\right)^2 \frac{1}{\sin^4(\theta/2)}; \tag{3.76}$$

this expression is commonly referred to as the Rutherford formula. The results obtained with the above program for different values of a are shown in Fig. 3.3. It is clear that when a is increased, the differential cross section becomes closer to that of the Coulomb scattering, as expected.

Exercises

3.1 Write a program that obtains the first-order and second-order derivatives from the five-point formulas. Check its accuracy with the function $f(x) = \cos x \sinh x$ with 101 uniformly spaced points for $0 \le x \le \pi/2$. Discuss the procedure used for dealing with the boundary points.

3.2 The derivatives of a function can be obtained by first performing the interpolation of $f(x_i)$ and then obtaining the derivatives of the interpolation polynomial. Show that: if $p_1(x)$ is used to interpolate the function and $x_0 = x - h$ and $x_1 = x + h$ are used as the data points, the three-point formula for the first-order derivative is recovered; if $p_2(x)$ is used to interpolate the function and $x_0 = x - h$, $x_1 = x$, and $x_2 = x + h$ are used as the data points, the three-point formula for the second-order derivative is recovered; if $p_3(x)$ is used to interpolate the function and $x_0 = x - 2h$, $x_1 = x - h$, $x_2 = x + h$, and $x_3 = x + 2h$ are used as the data points, the five-point formula for the first-order derivative is recovered; and if $p_4(x)$ is used to interpolate the function and $x_0 = x - 2h$, $x_1 = x - h$, $x_2 = x$, $x_3 = x + h$, and $x_4 = x + 2h$ are used as the data points, the five-point formula for the second-order derivative is recovered.

3.3 The Richardson extrapolation is a recursive scheme that can be used to improve the accuracy of the evaluation of derivatives. From the Taylor expansions of $f(x - h)$ and $f(x + h)$, we know that

$$\Delta_1(h) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + \sum_{k=1}^{\infty} c_{2k} h^{2k}.$$

Starting from $\Gamma_{k0} = \Delta_1(h/2^k)$, show that the Richardson extrapolation

$$\Gamma_{kl} = \frac{4^l}{4^l - 1}\Gamma_{kl-1} - \frac{1}{4^l - 1}\Gamma_{k-1l-1},$$

for $0 \le l \le k$, leads to

$$f'(x) = \Gamma_{kl} - \sum_{m=l+1}^{\infty} B_{ml} \left(\frac{h}{2^k}\right)^{2m},$$

where

$$B_{kl} = \left(\frac{4^l - 4^k}{4^l - 1}\right) B_{kl-1},$$

with $B_{kk} = 0$. Write a subprogram that generates the Richardson extrapolation and evaluates the first-order derivative $f'(x) \simeq \Gamma_{nn}$ for a given n. Test the subprogram with $f(x) = \sin x$.

3.4 Repeat Exercise 3.3 with $\Delta_1(h)$ replaced by

$$\Delta_2(h) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

and $f'(x)$ replaced by $f''(x)$. Is there any significant difference?

3.5 Using the fact that

$$\Delta_2(h) - 4\Delta_2(h/2) = -3f''(x) + O(h^4),$$

construct a subprogram that calculates the second-order derivative $f''(x)$ adaptively. (Δ_2 is defined in Exercise 3.4.) Then apply the subprogram to $f(x) = e^{-x} \ln x$. Is there any significant difference between the scheme here and the adaptive scheme for the first-order derivative introduced in Section 3.1?

3.6 Derive the Simpson rule with a pair of slices with an equal interval by using the Taylor expansion of $f(x)$ around x_i in the region $[x_{i-1}, x_{i+1}]$ and the three-point formulas for the first-order and second-order derivatives. Show that the contribution of the last slice is also correctly given by the formula in Section 3.2 under such an approach.

3.7 Derive the Simpson rule with $f(x) = ax^2 + bx + c$ going through points x_{i-1}, x_i, and x_{i+1}. Determine a, b, and c from the three equations given at the three data points first and then carry out the integration over the two slices with the quadratic curve being the integrand.

3.8 Develop a program that can calculate the integral with a given integrand $f(x)$ in the region $[a, b]$ by the Simpson rule with nonuniform data points. Check its accuracy with the integral $\int_0^\infty e^{-x}dx$ with $x_j = jhe^{j\alpha}$, where h and α are small constants.

3.9 Expand the integrand of $S = \int_a^b f(x)\,dx$ in a Taylor series around $x = a$ and show that

$$\Delta S_0 = S - S_0 \approx -\frac{h^3}{12} f''(a),$$

where $h = b - a$ and S_0 is the trapezoid evaluation of S with $x_i = a$ and $x_{i+1} = b$. If the region $[a, b]$ is divided into two, $[a, c]$ and $[c, b]$, with $c = (a + b)/2$, show that

$$\Delta S_1 = S - S_1 \approx -\frac{h^3}{48} f''(a),$$

where S_1 is the sum of the trapezoid evaluations of the integral in the regions $[a, c]$ and $[c, b]$. Using the fact that $S_1 - S_0 \approx 3\Delta S_1$, write a subprogram that performs the adaptive trapezoid evaluation of an integral to a specified accuracy.

3.10 The Romberg algorithm is a recursive procedure for evaluating an integral based on the adaptive trapezoid rule with

$$S_{kl} = \frac{4^l}{4^l - 1} S_{kl-1} - \frac{1}{4^l - 1} S_{k-1l-1},$$

where S_{k0} is the evaluation of the integral with the adaptive trapezoid rule to the kth level. Show that

$$\lim_{k\to\infty} S_{kl} = S = \int_a^b f(x)\,dx$$

for any l. Find the formula for the error estimate $\Delta S_{kl} = S - S_{kl}$.

3.11 Apply the secant method developed in Section 3.3 to solve $f(x) = e^{x^2} \ln x^2 - x = 0$. Discuss the procedure for dealing with more than one root in a given region.

3.12 Develop a subprogram that implements the Newton method to solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, where both $\mathbf{f}$ and $\mathbf{x}$ are l-dimensional vectors. Test the subprogram with $f_1(x_1, x_2) = e^{x_1^2} \ln x_2 - x_1^2$ and $f_2(x_1, x_2) = e^{x_2} \ln x_1 - x_2^2$.

3.13 Write a routine that returns the minimum of a single-variable function $g(x)$ in a given region $[a, b]$. Assume that the first-order and second-order derivatives of $g(x)$ are not explicitly given. Test the routine with some well-known functions, for example, $g(x) = x^2$.

3.14 Consider clusters of ions $(\mathrm{Na}^+)_n(\mathrm{Cl}^-)_m$, where m and n are small, positive integers. Use the steepest-descent method to obtain the stable geometric structures of the clusters. Use the molecular potential given in Eq. (3.52) for opposite charges but the Coulomb potential for like charges.

3.15 Modify the program in Section 3.5 to evaluate the differential cross section of the Lennard–Jones potential

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right].$$

Choose ε as the energy unit and σ as the length unit.

3.16 Show that the period of a pendulum confined in a vertical plane is

$$T = 4\sqrt{\frac{\ell}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos\theta - \cos\theta_0}},$$

where $\theta_0 < \pi$ is the maximum of the angle between the pendulum and the downward vertical, ℓ is the length of the pendulum, and g is the gravitational acceleration. Evaluate this integral numerically for $\theta_0 = \pi/128, \pi/64, \pi/32, \pi/16, \pi/8, \pi/4, \pi/2$, and compare the numerical results with the small-angle approximation $T \simeq 2\pi\sqrt{\ell/g}$.

3.17 Show that the time taken for a meterstick to fall on a frictionless, horizontal surface from an initial angle θ_0 to a final angle θ with the horizontal is

$$t = \frac{1}{2}\sqrt{\frac{\ell}{3g}} \int_{\theta}^{\theta_0} \sqrt{\frac{1 + 3\cos^2\phi}{\sin\theta_0 - \sin\phi}}\, d\phi,$$

where ℓ is the length of the meterstick (1 meter) and g is the gravitational acceleration. Evaluate this time numerically with $\theta_0 = \pi/8, \pi/4, \pi/2$, and $\theta = 0$.

3.18 For a classical particle of mass m moving in a one-dimensional, symmetric potential well $U(x) = U(-x)$, show that the inverse function is given by

$$x(U) = \frac{1}{2\pi\sqrt{2m}} \int_0^U \frac{T(E)\, dE}{\sqrt{U - E}},$$

where $T(E)$ is the period of the motion for a given total energy E. Find $U(x)$ numerically if $T/T_0 = 1 + \alpha e^{-E/E_0}$, for $\alpha = 0, 0.1, 1, 10$. Use T_0, E_0, and $T_0\sqrt{E_0}/(2\pi\sqrt{2m})$ as the units of the period, energy, and position, respectively. Discuss the accuracy of the numerical results by comparing them with an available analytical result.

Chapter 4
Ordinary differential equations

Most problems in physics and engineering appear in the form of differential equations. For example, the motion of a classical particle is described by Newton's equation, which is a second-order ordinary differential equation involving at least a second-order derivative in time, and the motion of a quantum particle is described by the Schrödinger equation, which is a partial differential equation involving a first-order partial derivative in time and second-order partial derivatives in coordinates. The dynamics and statics of bulk materials such as fluids and solids are all described by differential equations.

In this chapter, we introduce some basic numerical methods for solving ordinary differential equations. We will discuss the corresponding schemes for partial differential equations in Chapter 7 and some more advanced techniques for the many-particle Newton equation and the many-body Schrödinger equation in Chapters 8 and 10. Hydrodynamics and magnetohydrodynamics are treated in Chapter 9.

In general, we can classify ordinary differential equations into three major categories:

(1) initial-value problems, which involve time-dependent equations with given initial conditions;
(2) boundary-value problems, which involve differential equations with specified boundary conditions;
(3) eigenvalue problems, which involve solutions for selected parameters (eigenvalues) in the equations.

In reality, a problem may involve more than just one of the categories listed above. A common situation is that we have to separate several variables by introducing multipliers so that the initial-value problem is isolated from the boundary-value or eigenvalue problem. We can then solve the boundary-value or eigenvalue problem first to determine the multipliers, which in turn are used to solve the related initial-value problem. We will cover separation of variables in Chapter 7. In this chapter, we concentrate on the basic numerical methods for all the three categories listed above and illustrate how to use these techniques to solve problems encountered in physics and other related fields.

4.1 Initial-value problems

Typically, initial-value problems involve dynamical systems, for example, the motion of the Moon, Earth, and the Sun, the dynamics of a rocket, or the propagation of ocean waves. The behavior of a dynamical system can be described by a set of first-order differential equations,

$$\frac{d\mathbf{y}}{dt} = \mathbf{g}(\mathbf{y}, t), \tag{4.1}$$

where

$$\mathbf{y} = (y_1, y_2, \ldots, y_l) \tag{4.2}$$

is the dynamical variable vector, and

$$\mathbf{g}(\mathbf{y}, t) = [g_1(\mathbf{y}, t), g_2(\mathbf{y}, t), \ldots, g_l(\mathbf{y}, t)] \tag{4.3}$$

is the generalized velocity vector, a term borrowed from the definition of the velocity $\mathbf{v}(\mathbf{r}, t) = d\mathbf{r}/dt$ for a particle at position $\mathbf{r}$ and time t. Here l is the total number of dynamical variables. In principle, we can always obtain the solution of the above equation set if the initial condition $\mathbf{y}(t = 0) = \mathbf{y}_0$ is given and a solution exists. For the case of the particle moving in one dimension under an elastic force discussed in Chapter 1, the dynamics is governed by Newton's equation

$$f = ma, \tag{4.4}$$

where a and m are the acceleration and mass of the particle, respectively, and f is the force exerted on the particle. This equation can be viewed as a special case of Eq. (4.1) with $l = 2$: that is, $y_1 = x$ and $y_2 = v = dx/dt$, and $g_1 = v = y_2$ and $g_2 = f/m = -kx/m = -ky_1/m$. Then we can rewrite Newton's equation in the form of Eq. (4.1):

$$\frac{dy_1}{dt} = y_2, \tag{4.5}$$

$$\frac{dy_2}{dt} = -\frac{k}{m}y_1. \tag{4.6}$$

If the initial position $y_1(0)$ and the initial velocity $y_2(0) = v(0)$ are given, we can solve the problem numerically as demonstrated in Chapter 1.

In fact, most higher-order differential equations can be transformed into a set of coupled first-order differential equations in the form of Eq. (4.1). The higher-order derivatives are usually redefined into new dynamical variables during the transformation. The velocity in Newton's equation discussed above is such an example.

4.2 The Euler and Picard methods

For convenience of notation, we will work out our numerical schemes for cases with only one dynamical variable, that is, treating $\mathbf{y}$ and $\mathbf{g}$ in Eq. (4.1) as one-dimensional vectors or signed scalars. Extending the formalism developed here to multivariable cases is straightforward. We will illustrate such an extension in Sections 4.3 and 4.5.

Intuitively, Eq. (4.1) can be solved numerically as was done in Chapter 1 for the problem of a particle moving in one dimension with the time derivative of the dynamical variable approximated by the average generalized velocity as

$$\frac{dy}{dt} \simeq \frac{y_{i+1} - y_i}{t_{i+1} - t_i} \simeq g(y_i, t_i). \tag{4.7}$$

Note that the indices i and $i + 1$ here are for the time steps. We will also use $g_i = g(y_i, t_i)$ to simplify the notation. The approximation of the first-order derivative in Eq. (4.7) is equivalent to the two-point formula, which has a possible error of $O(|t_{i+1} - t_i|)$. If we take evenly spaced time points with a fixed time step $\tau = t_{i+1} - t_i$ and rearrange the terms in Eq. (4.7), we obtain the simplest algorithm for initial-value problems,

$$y_{i+1} = y_i + \tau g_i + O(\tau^2), \tag{4.8}$$

which is commonly known as the *Euler method* and was used in the example of a particle moving in one dimension in Chapter 1. We can reach the same result by considering it as the Taylor expansion of y_{i+1} at t_i by keeping the terms up to the first order. The accuracy of this algorithm is relatively low. At the end of the calculation after a total of n steps, the error accumulated in the calculation is on the order of $nO(\tau^2) \simeq O(\tau)$. To illustrate the relative error in this algorithm, let us use the program for the one-dimensional motion given in Section 1.3. Now if we use a time step of 0.02π instead of $0.000\,02\pi$, the program can accumulate a sizable error. The results for the position and velocity of the particle in the first period are given in Fig. 4.1. The results from a better algorithm with a corrector to be discussed in Section 4.3 and the exact results are also shown for comparison. The accuracy of the Euler algorithm is very low. We have 100 points in one period of the motion, which is a typical number of points chosen in most numerical calculations. If we go on to the second period, the error will increase further. For problems without periodic motion, the results at a later time would be even worse. We can conclude that this algorithm cannot be adopted in actual numerical solutions of most physics problems. How can we improve the algorithm so that it will become practical?

We can formally rewrite Eq. (4.1) as an integral

$$y_{i+j} = y_i + \int_{t_i}^{t_{i+j}} g(y, t)\, dt, \tag{4.9}$$

which is the exact solution of Eq. (4.1) if the integral in the equation can be obtained exactly for any given integers i and j. Because we cannot obtain the integral in Eq. (4.9) exactly in general, we have to approximate it. The accuracy in the approximation of the integral determines the accuracy of the solution. If

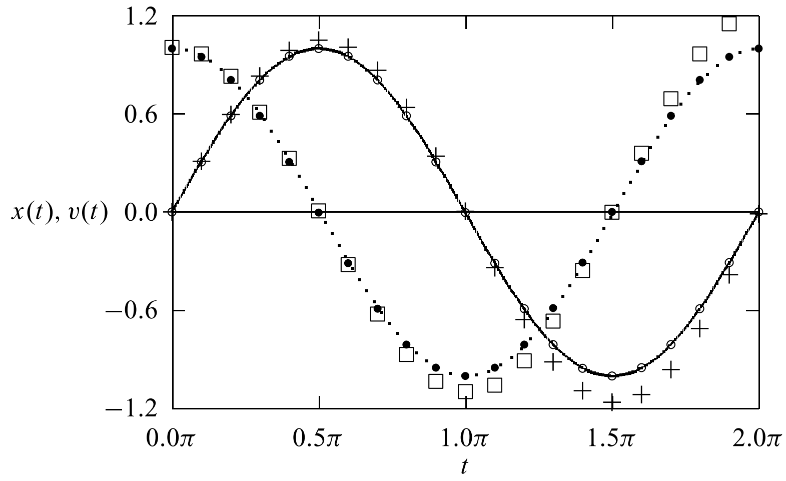


Fig. 4.1 The position (+) and velocity (□) of the particle moving in a one-dimensional space under an elastic force calculated using the Euler method with a time step of 0.02π compared with the position (○) and velocity (●) calculated with the predictor–corrector method and the exact results (solid and dotted lines).

we take the simplest case of $j = 1$ and approximate $g(y, t) \simeq g_i$ in the integral, we recover the Euler algorithm of Eq. (4.8).

The *Picard method* is an adaptive scheme, with each iteration carried out by using the solution from the previous iteration as the input on the right-hand side of Eq. (4.9). For example, we can use the solution from the Euler method as the starting point, and then carry out Picard iterations at each time step. In practice, we need to use a numerical quadrature to carry out the integration on the right-hand side of the equation. For example, if we choose $j = 1$ and use the trapezoid rule for the integral in Eq. (4.9), we obtain the algorithm

$$y_{i+1} = y_i + \frac{\tau}{2}(g_i + g_{i+1}) + O(\tau^3). \tag{4.10}$$

Note that $g_{i+1} = g(y_{i+1}, t_{i+1})$ contains y_{i+1}, which is provided adaptively in the Picard method. For example, we can take the solution at the previous time step as a guess of the solution at the current time, $y_{i+1}^{(0)} = y_i$, and then iterate the solution from right to left in the above equation, namely, $y_{i+1}^{(k+1)} = y_i + \frac{\tau}{2}(g_i + g_{i+1}^{(k)})$. The Picard method can be slow if the initial guess is not very close to the actual solution; it may not even converge if certain conditions are not satisfied. Can we avoid such tedious iterations by an intelligent guess of the solution?

4.3 Predictor–corrector methods

One way to avoid having to perform tedious iterations is to use the so-called predictor–corrector method. We can apply a less accurate algorithm to predict the next value y_{i+1} first, for example, using the Euler algorithm of Eq. (4.8), and then apply a better algorithm to improve the new value, for example, using the Picard algorithm of Eq. (4.10). If we apply this system to the one-dimensional motion studied with the Euler method, we obtain a much better result with the same choice of time step, for example, $\tau = 0.02\pi$. The following program is the implementation of this simplest predictor–corrector method to such a problem.

```
// A program to study the motion of a particle under an
// elastic force in one dimension through the simplest
// predictor-corrector scheme.

import java.lang.*;
public class Motion2 {
  static final int n = 100, j = 5;
  public static void main(String argv[]) {
    double x[] = new double[n+1];
    double v[] = new double[n+1];

 // Assign time step and initial position and velocity
    double dt = 2*Math.PI/n;
    x[0] = 0;
    v[0] = 1;

 // Calculate other position and velocity recursively
    for (int i=0; i<n; ++i) {

   // Predict the next position and velocity
      x[i+1] = x[i]+v[i]*dt;
      v[i+1] = v[i]-x[i]*dt;

   // Correct the new position and velocity
      x[i+1] = x[i]+(v[i]+v[i+1])*dt/2;
      v[i+1] = v[i]-(x[i]+x[i+1])*dt/2;
    }

 // Output the result in every j time steps
    double t = 0;
    double jdt = j*dt;
    for (int i=0; i<=n; i+=j) {
      System.out.println(t +" " + x[i] +" " + v[i]);
      t += jdt;
    }
  }
}
```

The numerical result from the above program is shown in Fig. 4.1 for comparison. The improvement obtained is significant. With 100 mesh points in one period of motion, the errors are less than the size of the smallest symbol used in the figure. Furthermore, the improvement can also sustain long runs with more periods involved.

Another way to improve an algorithm is by increasing the number of mesh points j in Eq. (4.9). Thus we can apply a better quadrature to the integral. For example, if we take $j = 2$ in Eq. (4.9) and then use the linear interpolation scheme to approximate $g(y, t)$ in the integral from g_i and g_{i+1}, we obtain

$$g(y, t) = \frac{(t - t_i)}{\tau} g_{i+1} - \frac{(t - t_{i+1})}{\tau} g_i + O(\tau^2). \tag{4.11}$$

Now if we carry out the integration with $g(y, t)$ given from this equation, we obtain a new algorithm

$$y_{i+2} = y_i + 2\tau g_{i+1} + O(\tau^3), \tag{4.12}$$

which has an accuracy one order higher than that of the Euler algorithm. However, we need the values of the first two points in order to start this algorithm, because $g_{i+1} = g(y_{i+1}, t_{i+1})$. Usually, the dynamical variable and the generalized velocity at the second point can be obtained by the Taylor expansion around the initial point at $t = 0$. For example, we have

$$y_1 = y_0 + \tau g_0 + \frac{\tau^2}{2}\left(\frac{\partial g_0}{\partial t} + g_0\frac{\partial g_0}{\partial y}\right) + O(\tau^3) \tag{4.13}$$

and

$$g_1 = g(y_1, \tau). \tag{4.14}$$

We have truncated y_1 at $O(\tau^3)$ to preserve the same order of accuracy in the algorithm of Eq. (4.12). We have also used the fact that $dy/dt = g$.

Of course, we can always include more points in the integral of Eq. (4.9) to obtain algorithms with apparently higher accuracy, but we will need the values of more points in order to start the algorithm. This becomes impractical if we need more than two points in order to start the algorithm, because the errors accumulated from the approximations of the first few points will eliminate the apparently high accuracy of the algorithm.

We can make the accuracy even higher by using a better quadrature. For example, we can take $j = 2$ in Eq. (4.9) and apply the Simpson rule, discussed in Section 3.2, to the integral. Then we have

$$y_{i+2} = y_i + \frac{\tau}{3}(g_{i+2} + 4g_{i+1} + g_i) + O(\tau^5). \tag{4.15}$$

This implicit algorithm can be used as the corrector if the algorithm in Eq. (4.12) is used as the predictor.

Let us take the simple model of a motorcycle jump over a gap as an illustrating example. The air resistance on a moving object is roughly given by $\mathbf{f}_r = -\kappa v\mathbf{v} = -cA\rho v\mathbf{v}$, where A is cross section of the moving object, ρ is the density of the air, and c is a coefficient that accounts for all the other factors on the order of 1. So the motion of the system is described by the equation set

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}, \tag{4.16}$$

$$\frac{d\mathbf{v}}{dt} = \mathbf{a} = \frac{\mathbf{f}}{m}, \tag{4.17}$$

where

$$\mathbf{f} = -mg\hat{\mathbf{y}} - \kappa v\mathbf{v} \tag{4.18}$$

is the total force on the system of a total mass m. Here $\hat{\mathbf{y}}$ is the unit vector pointing upward. Assuming that we have the first point given, that is, $\mathbf{r}_0$ and $\mathbf{v}_0$ at $t = 0$, the next point is then obtained from the Taylor expansions and the equation set with

$$\mathbf{r}_1 = \mathbf{r}_0 + \tau\mathbf{v}_0 + \frac{\tau^2}{2}\mathbf{a}_0 + O(\tau^3), \tag{4.19}$$

$$\mathbf{v}_1 = \mathbf{v}_0 + \tau\mathbf{a}_0 + \frac{\tau^2}{2}\frac{d\mathbf{a}_0}{dt} + O(\tau^3), \tag{4.20}$$

where

$$\frac{d\mathbf{a}_0}{dt} = -\frac{\kappa}{m}\left(v_0\mathbf{a}_0 + \frac{\mathbf{v}_0 \cdot \mathbf{a}_0}{v_0}\mathbf{v}_0\right). \tag{4.21}$$

The following program calculates the trajectory of the motorcycle with a given taking-off angle.

```
// An example of modeling a motorcycle jump with the
// two-point predictor-corrector scheme.

import java.lang.*;
public class Jump {
  static final int n = 100, j = 2;
  public static void main(String argv[]) {
    double x[] = new double[n+1];
    double y[] = new double[n+1];
    double vx[] = new double[n+1];
    double vy[] = new double[n+1];
    double ax[] = new double[n+1];
    double ay[] = new double[n+1];

 // Assign all the constants involved
    double g = 9.80;
    double angle = 42.5*Math.PI/180;
    double speed = 67;
    double mass = 250;
    double area = 0.93;
    double density = 1.2;
    double k = area*density/(2*mass);
    double dt = 2*speed*Math.sin(angle)/(g*n);
    double d = dt*dt/2;

 // Assign the quantities for the first two points
    x[0] = y[0] = 0;
    vx[0] = speed*Math.cos(angle);
    vy[0] = speed*Math.sin(angle);
    double v = Math.sqrt(vx[0]*vx[0]+vy[0]*vy[0]);
    ax[0] = -k*v*vx[0];
    ay[0] = -g-k*v*vy[0];
    double p = vx[0]*ax[0]+vy[0]*ay[0];
    x[1] = x[0]+dt*vx[0]+d*ax[0];
    y[1] = y[0]+dt*vy[0]+d*ay[0];
    vx[1] = vx[0]+dt*ax[0]-d*k*(v*ax[0]+p*vx[0]/v);
    vy[1] = vy[0]+dt*ay[0]-d*k*(v*ay[0]+p*vy[0]/v);
```

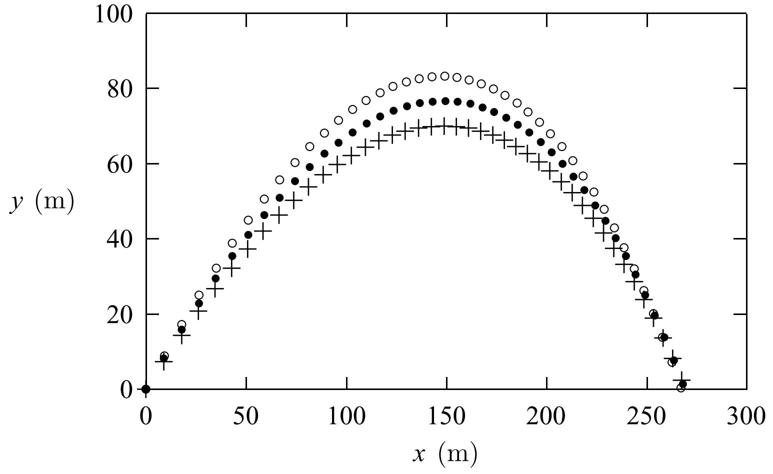


Fig. 4.2 The trejactory of the motorcycle as calculated in the example program with different taking-off angles: 40° (+), 42.5° (•), and 45° (○).

```
    v = Math.sqrt(vx[1]*vx[1]+vy[1]*vy[1]);
    ax[1] = -k*v*vx[1];
    ay[1] = -g-k*v*vy[1];

 // Calculate other position and velocity recursively
    double d2 = 2*dt;
    double d3 = dt/3;
    for (int i=0; i<n-1; ++i) {

   // Predict the next position and velocity
      x[i+2] = x[i]+d2*vx[i+1];
      y[i+2] = y[i]+d2*vy[i+1];
      vx[i+2] = vx[i]+d2*ax[i+1];
      vy[i+2] = vy[i]+d2*ay[i+1];
      v = Math.sqrt(vx[i+2]*vx[i+2]+vy[i+2]*vy[i+2]);
      ax[i+2] = -k*v*vx[i+2];
      ay[i+2] = -g-k*v*vy[i+2];

   // Correct the new position and velocity
      x[i+2] = x[i]+d3*(vx[i+2]+4*vx[i+1]+vx[i]);
      y[i+2] = y[i]+d3*(vy[i+2]+4*vy[i+1]+vy[i]);
      vx[i+2] = vx[i]+d3*(ax[i+2]+4*ax[i+1]+ax[i]);
      vy[i+2] = vy[i]+d3*(ay[i+2]+4*ay[i+1]+ay[i]);
    }

 // Output the result in every j time steps
    for (int i=0; i<=n; i+=j)
      System.out.println(x[i] +" " + y[i]);
  }
}
```

In the above program, we have used the cross section $A = 0.93$ m^2, the taking-off speed $v_0 = 67$ m/s, the air density $\rho = 1.2$ kg/m^3, the combined mass of the motorcycle and the person 250 kg, and the coefficient $c = 1$. The results for three different taking-off angles are plotted in Fig. 4.2. The maximum range is about 269 m at a taking-off angle of about 42.5°.

In principle, we can go further by including more points in the integration quadrature of Eq. (4.9) and interested readers can find many multiple-point formulas in Davis and Polonsky (1965).

4.4 The Runge–Kutta method

The accuracy of the methods that we have discussed so far can be improved only by including more starting points, which is not always practical, because a problem associated with a dynamical system usually has only the first point, namely, the initial condition, given. A more practical method that requires only the first point in order to start or to improve the algorithm is the *Runge–Kutta method*, which is derived from two different Taylor expansions of the dynamical variables and their derivatives defined in Eq. (4.1).

Formally, we can expand $y(t+\tau)$ in terms of the quantities at t with the Taylor expansion

$$\begin{aligned} y(t+\tau) &= y + \tau y' + \frac{\tau^2}{2} y'' + \frac{\tau^3}{3!} y^{(3)} + \cdots \\ &= y + \tau g + \frac{\tau^2}{2}(g_t + g g_y) + \frac{\tau^3}{6}\left(g_{tt} + 2 g g_{ty} + g^2 g_{yy} + g g_y^2 + g_t g_y\right) + \cdots, \end{aligned} \tag{4.22}$$

where the subscript indices denote partial derivatives for example, $g_{yt} = \partial^2 g/\partial y \partial t$. We can also formally write the solution at $t+\tau$ as

$$y(t+\tau) = y(t) + \alpha_1 c_1 + \alpha_2 c_2 + \cdots + \alpha_m c_m, \tag{4.23}$$

with

$$\begin{aligned} c_1 &= \tau g(y, t), \\ c_2 &= \tau g(y + \nu_{21} c_1, t + \nu_{21}\tau), \\ c_3 &= \tau g(y + \nu_{31} c_1 + \nu_{32} c_2, t + \nu_{31}\tau + \nu_{32}\tau), \\ &\vdots \\ c_m &= \tau g\left(y + \sum_{i=1}^{m-1} \nu_{mi} c_i, t + \tau \sum_{i=1}^{m-1} \nu_{mi}\right), \end{aligned} \tag{4.24}$$

where α_i (with $i = 1, 2, \ldots, m$) and ν_{ij} (with $i = 2, 3, \ldots, m$ and $j < i$) are parameters to be determined. We can expand Eq. (4.23) into a power series of τ by carrying out Taylor expansions for all c_i with $i = 1, 2, \ldots, m$. Then we can compare the resulting expression of $y(t+\tau)$ from Eq. (4.23) with the expansion in Eq. (4.22) term by term. A set of equations for α_i and ν_{ij} is obtained by keeping the coefficients for the terms with the same power of τ on both sides equal. By truncating the expansion to the term $O(\tau^m)$, we obtain m equations but with $m + m(m-1)/2$ parameters (α_i and ν_{ij}) to be determined. Thus, there are still options left in choosing these parameters.

Let us illustrate this scheme by working out the case for $m = 2$ in detail. If only the terms up to $O(\tau^2)$ are kept in Eq. (4.22), we have

$$y(t+\tau) = y + \tau g + \frac{\tau^2}{2}(g_t + g g_y). \tag{4.25}$$

We can obtain an expansion up to the same order by truncating Eq. (4.23) at $m = 2$,

$$y(t+\tau) = y(t) + \alpha_1 c_1 + \alpha_2 c_2, \tag{4.26}$$

with

$$c_1 = \tau g(y, t), \tag{4.27}$$

$$c_2 = \tau g(y + \nu_{21} c_1, t + \nu_{21}\tau). \tag{4.28}$$

Now if we perform the Taylor expansion for c_2 up to the term $O(\tau^2)$, we have

$$c_2 = \tau g + \nu_{21}\tau^2(g_t + g g_y). \tag{4.29}$$

Substituting c_1 and the expansion of c_2 above back into Eq. (4.26) yields

$$y(t+\tau) = y(t) + (\alpha_1 + \alpha_2)\tau g + \alpha_2 \tau^2 \nu_{21}(g_t + g g_y). \tag{4.30}$$

If we compare this expression with Eq. (4.25) term by term, we have

$$\alpha_1 + \alpha_2 = 1, \tag{4.31}$$

$$\alpha_2 \nu_{21} = \frac{1}{2}. \tag{4.32}$$

As pointed out earlier, there are only m (2 in this case) equations available but there are $m + m(m-1)/2$ (3 in this case) parameters to be determined. We do not have a unique solution for all the parameters; thus we have flexibility in assigning their values as long as they satisfy the m equations. We could choose $\alpha_1 = \alpha_2 = 1/2$ and $\nu_{21} = 1$, or $\alpha_1 = 1/3$, $\alpha_2 = 2/3$, and $\nu_{21} = 3/4$. The flexibility in choosing the parameters provides one more way to increase the numerical accuracy in practice. We can adjust the parameters according to the specific problems involved.

The most commonly known and widely used Runge–Kutta method is the one with Eqs. (4.22) and (4.23) truncated at the terms of $O(\tau^4)$. We will give the result here and leave the derivation as an exercise to the reader. This well-known fourth-order Runge–Kutta algorithm is given by

$$y(t+\tau) = y(t) + \frac{1}{6}(c_1 + 2c_2 + 2c_3 + c_4), \tag{4.33}$$

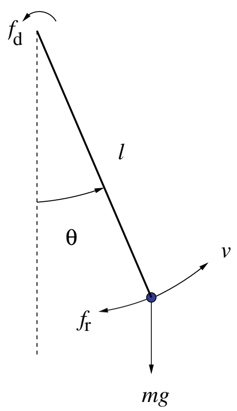


Fig. 4.3 A sketch of a driven pendulum under damping: f_d is the driving force and f_r is the resistive force.

with

$$c_1 = \tau g(y, t), \tag{4.34}$$

$$c_2 = \tau g\left(y + \frac{c_1}{2}, t + \frac{\tau}{2}\right), \tag{4.35}$$

$$c_3 = \tau g\left(y + \frac{c_2}{2}, t + \frac{\tau}{2}\right), \tag{4.36}$$

$$c_4 = \tau g(y + c_3, t + \tau). \tag{4.37}$$

We can easily show that the above selection of parameters does satisfy the required equations. As pointed out earlier, this selection is not unique and can be modified according to the problem under study.

4.5 Chaotic dynamics of a driven pendulum

Before discussing numerical methods for solving boundary-value and eigenvalue problems, let us apply the Runge–Kutta method to the initial-value problem of a dynamical system. Even though we are going to examine only one special system, the approach, as shown below, is quite general and suitable for all other problems.

Consider a pendulum consisting of a light rod of length l and a point mass m attached to the lower end. Assume that the pendulum is confined to a vertical plane, acted upon by a driving force f_d and a resistive force f_r as shown in Fig. 4.3. The motion of the pendulum is described by Newton's equation along the tangential direction of the circular motion of the point mass,

$$ma_t = f_g + f_d + f_r, \tag{4.38}$$

where $f_g = -mg \sin\theta$ is the contribution of gravity along the direction of motion, with θ being the angle made by the rod with respect to the vertical line, and $a_t = ld^2\theta/dt^2$ is the acceleration along the tangential direction. Assume that the time dependency of the driving force is periodic as

$$f_d(t) = f_0 \cos \omega_0 t, \tag{4.39}$$

and the resistive force $f_r = -\kappa v$, where $v = l d\theta/dt$ is the velocity of the mass and κ is a positive damping parameter. This is a reasonable assumption for a pendulum set in a dense medium under a harmonic driving force. If we rewrite Eq. (4.38) in a dimensionless form with $\sqrt{l/g}$ chosen as the unit of time, we have

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \sin\theta = b\cos\omega_0 t, \tag{4.40}$$

where $q = \kappa/m$ and $b = f_0/ml$ are redefined parameters. As discussed at the beginning of this chapter, we can write the derivatives as variables. We can thus transform higher-order differential equations into a set of first-order differential equations. If we choose $y_1 = \theta$ and $y_2 = \omega = d\theta/dt$, we have

$$\frac{dy_1}{dt} = y_2, \tag{4.41}$$

$$\frac{dy_2}{dt} = -qy_2 - \sin y_1 + b\cos\omega_0 t, \tag{4.42}$$

which are in the form of Eq. (4.1). In principle, we can use any method discussed so far to solve this equation set. However, considering the accuracy required for long-time behavior, we use the fourth-order Runge–Kutta method here.

As we will show later from the numerical solutions of Eqs. (4.41) and (4.42), in different regions of the parameter space (q, b, ω_0) the system has quite different dynamics. Specifically, in some parameter regions the motion of the pendulum is totally chaotic.

If we generalize the fourth-order Runge–Kutta method discussed in the preceding section to multivariable cases, we have

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{1}{6}(\mathbf{c}_1 + 2\mathbf{c}_2 + 2\mathbf{c}_3 + \mathbf{c}_4), \tag{4.43}$$

with

$$\mathbf{c}_1 = \tau\mathbf{g}(\mathbf{y}_i, t_i), \tag{4.44}$$

$$\mathbf{c}_2 = \tau\mathbf{g}\left(\mathbf{y}_i + \frac{\mathbf{c}_1}{2}, t_i + \frac{\tau}{2}\right), \tag{4.45}$$

$$\mathbf{c}_3 = \tau\mathbf{g}\left(\mathbf{y}_i + \frac{\mathbf{c}_2}{2}, t_i + \frac{\tau}{2}\right), \tag{4.46}$$

$$\mathbf{c}_4 = \tau\mathbf{g}(\mathbf{y}_i + \mathbf{c}_3, t_i + \tau), \tag{4.47}$$

where $\mathbf{y}_i$ for any i and $\mathbf{c}_j$ for $j = 1, 2, 3, 4$ are multidimensional vectors. Note that generalizing an algorithm for the initial-value problem from the single-variable case to the multivariable case is straightforward. Other algorithms we have discussed can be generalized in exactly the same fashion.

In principle, the pendulum problem has three dynamical variables: the angle between the rod and the vertical line, θ, its first-order derivative $\omega = d\theta/dt$, and the phase of the driving force $\phi = \omega_0 t$. This is important because a dynamical system cannot be chaotic unless it has three or more dynamical variables. However in practice, we only need to worry about θ and ω because $\phi = \omega_0 t$ is the solution of ϕ.

Any physical quantities that are functions of θ are periodic: for example, $\omega(\theta) = \omega(\theta \pm 2n\pi)$, where n is an integer. Therefore, we can confine θ in the region $[-\pi, \pi]$. If θ is outside this region, it can be transformed back with $\theta' = \theta \pm 2n\pi$ without losing any generality. The following program is an implementation of the fourth-order Runge–Kutta algorithm as applied to the driven pendulum under damping.

```
// A program to study the driven pendulum under damping
// via the fourth-order Runge-Kutta algorithm.

import java.lang.*;
public class Pendulum {
  static final int n = 100, nt = 10, m = 5;
  public static void main(String argv[]) {
    double y1[] = new double[n+1];
    double y2[] = new double[n+1];
    double y[] = new double[2];

 // Set up time step and initial values
    double dt = 3*Math.PI/nt;
    y1[0] = y[0] = 0;
    y2[0] = y[1] = 2;

 // Perform the 4th-order Runge-Kutta integration
    for (int i=0; i<n; ++i) {
      double t = dt*i;
      y = rungeKutta(y, t, dt);
      y1[i+1] = y[0];
      y2[i+1] = y[1];

   // Bring theta back to the region [-pi, pi]
      int np = (int) (y1[i+1]/(2*Math.PI)+0.5);
      y1[i+1] -= 2*Math.PI*np;
    }

 // Output the result in every m time steps
    for (int i=0; i<=n; i+=m) {
      System.out.println("Angle: " + y1[i]);
      System.out.println("Angular velocity: " + y2[i]);
      System.out.println();
    }
  }

// Method to complete one Runge-Kutta step.

  public static double[] rungeKutta(double y[],
    double t, double dt) {
    int l = y.length;
    double c1[] = new double[l];
    double c2[] = new double[l];
    double c3[] = new double[l];
    double c4[] = new double[l];
```

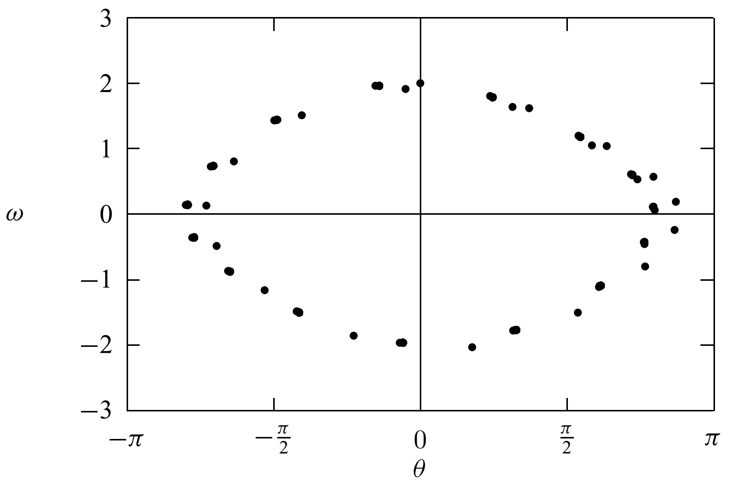


Fig. 4.4 The angular velocity ω versus the angle θ, with parameters $\omega_0 = 2/3$, $q = 0.5$, and $b = 0.9$. Under the given condition the system is apparently periodic. Here 1000 points from 10 000 time steps are shown.

```
    c1 = g(y, t);
    for (int i=0; i<l; ++i) c2[i] = y[i] + dt*c1[i]/2;
    c2 = g(c2, t+dt/2);
    for (int i=0; i<l; ++i) c3[i] = y[i] + dt*c2[i]/2;
    c3 = g(c3, t+dt/2);
    for (int i=0; i<l; ++i) c4[i] = y[i] + dt*c3[i];
    c4 = g(c4, t+dt);
    for (int i=0; i<l; ++i)
      c1[i] = y[i] + dt*(c1[i]+2*(c2[i]+c3[i])+c4[i])/6;
    return c1;
  }

// Method to provide the generalized velocity vector.

  public static double[] g(double y[], double t) {
    int l = y.length;
    double q = 0.5, b = 0.9, omega0 = 2.0/3;
    double v[] = new double[l];
    v[0] = y[1];
    v[1] = -Math.sin(y[0])+b*Math.cos(omega0*t);
    v[1] -= q*y[1];
    return v;
  }
}
```

Depending on the choice of the three parameters, q, b, and ω_0, the system can be periodic or chaotic. In Fig. 4.4 and Fig. 4.5, we show two typical numerical results. The dynamical behavior of the pendulum shown in Fig. 4.4 is periodic in the selected parameter region, and the dynamical behavior shown in Fig. 4.5 is chaotic in another parameter region. We can modify the program developed here to explore the dynamics of the pendulum through the whole parameter space and many important aspects of chaos. Interested readers can find discussions on these aspects in Baker and Gollub (1996).

Several interesting features appear in the results shown in Fig. 4.4 and Fig. 4.5. In Fig. 4.4, the motion of the system is periodic, with a period $T = 2T_0$, where

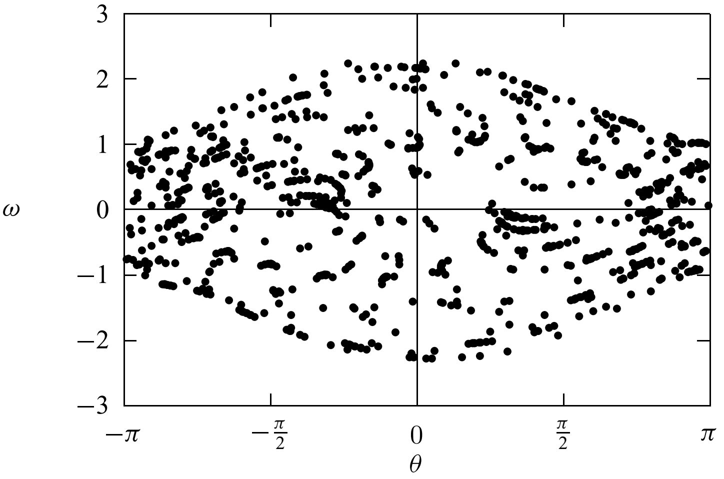


Fig. 4.5 The same plot as in Fig. 4.4, with parameters $\omega_0 = 2/3$, $q = 0.5$, and $b = 1.15$. The system at this point of the parameter space is apparently chaotic. Here 1000 points from 10 000 time steps are shown.

$T_0 = 2\pi/\omega_0$ is the period of the driving force. If we explore other parameter regions, we would find other periodic motions with $T = nT_0$, where n is an even, positive integer. The reason why n is even is that the system is moving away from being periodic to being chaotic; period doubling is one of the routes for a dynaimcal system to develop chaos. The chaotic behavior shown in Fig. 4.5 appears to be totally irregular; however, detailed analysis shows that the phase-space diagram (the ω–θ plot) has self-similarity at all length scales, as indicated by the fractal structure in chaos.

4.6 Boundary-value and eigenvalue problems

Another class of problems in physics requires the solving of differential equations with the values of physical quantities or their derivatives given at the boundaries of a specified region. This applies to the solution of the Poisson equation with a given charge distribution and known boundary values of the electrostatic potential or of the stationary Schrödinger equation with a given potential and boundary conditions.

A typical boundary-value problem in physics is usually given as a second-order differential equation

$$u'' = f(u, u'; x), \tag{4.48}$$

where u is a function of x, u' and u'' are the first-order and second-order derivatives of u with respect to x, and $f(u, u'; x)$ is a function of u, u', and x. Either u or u' is given at each boundary point. Note that we can always choose a coordinate system so that the boundaries of the system are at $x = 0$ and $x = 1$ without losing any generality if the system is finite. For example, if the actual boundaries are at $x = x_1$ and $x = x_2$ for a given problem, we can always bring them back to $x' = 0$

and $x' = 1$ with a transformation

$$x' = (x - x_1)/(x_2 - x_1). \tag{4.49}$$

For problems in one dimension, we can have a total of four possible types of boundary conditions:

(1) $u(0) = u_0$ and $u(1) = u_1$;
(2) $u(0) = u_0$ and $u'(1) = v_1$;
(3) $u'(0) = v_0$ and $u(1) = u_1$;
(4) $u'(0) = v_0$ and $u'(1) = v_1$.

The boundary-value problem is more difficult to solve than the similar initial-value problem with the differential equation. For example, if we want to solve an initial-value problem that is described by the differential equation given in Eq. (4.48) with x replaced by time t and the initial conditions $u(0) = u_0$ and $u'(0) = v_0$, we can first transform the differential equation as a set of two first-order differential equations with a redefinition of the first-order derivative as a new variable. The solution will follow if we adopt one of the algorithms discussed earlier in this chapter. However, for the boundary-value problem, we know only $u(0)$ or $u'(0)$, which is not sufficient to start an algorithm for the initial-value problem without some further work.

Typical eigenvalue problems are even more complicated, because at least one more parameter, that is, the eigenvalue, is involved in the equation: for example,

$$u'' = f(u, u'; x; \lambda), \tag{4.50}$$

with a set of given boundary conditions, defines an eigenvalue problem. Here the eigenvalue λ can have only some selected values in order to yield acceptable solutions of the equation under the given boundary conditions.

Let us take the longitudinal vibrations along an elastic rod as an illustrative example here. The equation describing the stationary solution of elastic waves is

$$u''(x) = -k^2 u(x), \tag{4.51}$$

where $u(x)$ is the displacement from equilibrium at x and the allowed values of k^2 are the eigenvalues of the problem. The wavevector k in the equation is related to the phase speed c of the wave along the rod and the allowed angular frequency ω by the dispersion relation

$$\omega = ck. \tag{4.52}$$

If both ends ($x = 0$ and $x = 1$) of the rod are fixed, the boundary conditions are then $u(0) = u(1) = 0$. If one end ($x = 0$) is fixed and the other end ($x = 1$) is free, the boundary conditions are then $u(0) = 0$ and $u'(1) = 0$. For this problem, we can obtain an analytical solution. For example, if both ends of the rod are fixed, the eigenfunctions

$$u_l(x) = \sqrt{2}\sin k_l x \tag{4.53}$$

are the possible solutions of the differential equation. Here the eigenvalues are given by

$$k_l^2 = (l\pi)^2, \tag{4.54}$$

with $l = 1, 2, \ldots, \infty$. The complete solution of the longitudinal waves along the elastic rod is given by a linear combination of all the eigenfunctions with their associated initial-value solutions as

$$u(x,t) = \sum_{l=1}^{\infty}(a_l \sin\omega_l t + b_l \cos\omega_l t)u_l(x), \tag{4.55}$$

where $\omega_l = ck_l$, and a_l and b_l are the coefficients to be determined by the initial conditions. We will come back to this problem in Chapter 7 when we discuss the solution of a partial differential equation.

4.7 The shooting method

A simple method for solving the boundary-value problem of Eq. (4.48) and the eigenvalue problem of Eq. (4.50) with a set of given boundary conditions is the so-called *shooting method*. We will first discuss how this works for the boundary-value problem and then generalize it to the eigenvalue problem.

We first convert the second-order differential equation into two first-order differential equations by defining $y_1 = u$ and $y_2 = u'$, namely,

$$\frac{dy_1}{dx} = y_2, \tag{4.56}$$

$$\frac{dy_2}{dx} = f(y_1, y_2; x). \tag{4.57}$$

To illustrate the method, let us assume that the boundary conditions are $u(0) = u_0$ and $u(1) = u_1$. Any other types of boundary conditions can be solved in a similar manner.

The key here is to make the problem look like an initial-value problem by introducing an adjustable parameter; the solution is then obtained by varying the parameter. Because $u(0)$ is given already, we can make a guess for the first-order derivative at $x = 0$, for example, $u'(0) = \alpha$. Here α is the parameter to be adjusted. For a specific α, we can integrate the equation to $x = 1$ with any of the algorithms discussed earlier for initial-value problems. Because the initial choice of α can hardly be the actual derivative at $x = 0$, the value of the function $u_\alpha(1)$, resulting from the integration with $u'(0) = \alpha$ to $x = 1$, would not be the same as u_1. The idea of the shooting method is to use one of the root search algorithms to find the appropriate α that ensures $f(\alpha) = u_\alpha(1) - u_1 = 0$ within a given tolerance δ.

Let us take an actual numerical example to illustrate the scheme. Assume that we want to solve the differential equation

$$u'' = -\frac{\pi^2}{4}(u+1), \tag{4.58}$$

with the given boundary conditions $u(0) = 0$ and $u(1) = 1$. We can define the new variables $y_1 = u$ and $y_2 = u'$; then we have

$$\frac{dy_1}{dx} = y_2, \tag{4.59}$$

$$\frac{dy_2}{dx} = -\frac{\pi^2}{4}(y_1 + 1). \tag{4.60}$$

Now assume that this equation set has the initial values $y_1(0) = 0$ and $y_2(0) = \alpha$. Here α is a parameter to be adjusted in order to have $f(\alpha) = u_\alpha(1) - 1 = 0$. We can combine the secant method for the root search and the fourth-order Runge–Kutta method for initial-value problems to solve the above equation set. The following program is an implementation of such a combined approach to the boundary-value problem defined in Eq. (4.58) or Eqs. (4.59) and (4.60) with the given boundary conditions.

```
// An example of solving a boundary-value problem via
// the shooting method.  The Runge-Kutta and secant
// methods are used for integration and root search.

import java.lang.*;
public class Shooting {
  static final int n = 100, ni=10, m = 5;
  static final double h = 1.0/n;
  public static void main(String argv[]) {
    double del = 1e-6, alpha0 = 1, dalpha = 0.01;
    double y1[] = new double [n+1];
    double y2[] = new double [n+1];
    double y[] = new double [2];

 // Search for the proper solution of the equation
    y1[0] = y[0] = 0;
    y2[0] = y[1] = secant(ni, del, alpha0, dalpha);
    for (int i=0; i<n; ++i) {
      double x = h*i;
      y = rungeKutta(y, x, h);
      y1[i+1] = y[0];
      y2[i+1] = y[1];
    }

 // Output the result in every m points
    for (int i=0; i<=n; i+=m)
      System.out.println(y1[i]);
  }

  public static double secant(int n, double del,
    double x, double dx) {...}

// Method to provide the function for the root search.

  public static double f(double x) {
```

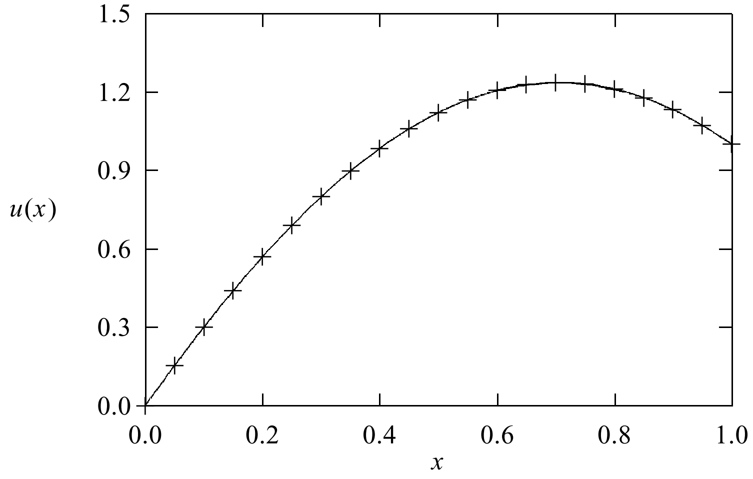


Fig. 4.6 The numerical solution of the boundary-value problem of Eq. (4.58) by the shooting method (+) compared with the analytical solution (solid line) of the same problem.

```
    double y[] = new double[2];
    y[0] = 0;
    y[1] = x;
    for (int i=0; i<n-1; ++i) {
      double xi = h*i;
      y = rungeKutta(y, xi, h);
    }
    return y[0]-1;
  }

  public static double[] rungeKutta(double y[],
    double t, double dt) {...}

// Method to provide the generalized velocity vector.

  public static double[] g(double y[], double t) {
    int k = y.length;
    double v[] = new double[k];
    v[0] = y[1];
    v[1] = -Math.PI*Math.PI*(y[0]+1)/4;
    return v;
  }
}
```

Note how we have combined the secant and Runge–Kutta methods. The boundary-value problem solved in the above program can also be solved exactly with an analytical solution

$$u(x) = \cos\frac{\pi x}{2} + 2\sin\frac{\pi x}{2} - 1. \tag{4.61}$$

We can easily check that the above expression does satisfy the equation and the boundary conditions. We plot both the numerical result obtained from the shooting method and the analytical solution in Fig. 4.6. Note that the shooting method provides a very accurate solution of the boundary-value problem. It is also a very general method for both the boundary-value and eigenvalue problems.

Boundary-value problems with other types of boundary conditions can be solved in a similar manner. For example, if $u'(0) = v_0$ and $u(1) = u_1$ are given, we can make a guess of $u(0) = \alpha$ and then integrate the equation set of y_1 and y_2 to $x = 1$. The root to be sought is from $f(\alpha) = u_\alpha(1) - u_1 = 0$. Here $u_\alpha(1)$ is the numerical result of the equation with $u(0) = \alpha$. If $u'(1) = v_1$ is given, the equation $f(\alpha) = u'_\alpha(1) - v_1 = 0$ is solved instead.

When we apply the shooting method to an eigenvalue problem, the parameter to be adjusted is no longer a parameter introduced but the eigenvalue of the problem. For example, if $u(0) = u_0$ and $u(1) = u_1$ are given, we can integrate the equation with $u'(0) = \alpha$, a small quantity. Then we search for the root of $f(\lambda) = u_\lambda(1) - u_1 = 0$ by varying λ. When $f(\lambda) = 0$ is satisfied, we obtain an approximate eigenvalue λ and the corresponding eigenstate from the normalized solution of $u_\lambda(x)$. The introduced parameter α is not relevant here, because it will be automatically modified to be the first-order derivative when the solution in normalized. In other words, we can choose the first-order derivative at the boundary arbitrarily and it will be adjusted to an appropriate value when the solutions are made orthonormal. We will demonstrate this in Section 4.9 with examples.

4.8 Linear equations and the Sturm–Liouville problem

Many eigenvalue or boundary-value problems are in the form of linear equations, such as

$$u'' + d(x)u' + q(x)u = s(x), \tag{4.62}$$

where $d(x)$, $q(x)$, and $s(x)$ are functions of x. Assume that the boundary conditions are $u(0) = u_0$ and $u(1) = u_1$. If all $d(x)$, $q(x)$, and $s(x)$ are smooth, we can solve the equation with the shooting method developed in the preceding section. In fact, we can show that an extensive search for the parameter α from $f(\alpha) = u_\alpha(1) - u_1 = 0$ is unnecessary in this case, because of the principle of superposition of linear equations: any linear combination of the solutions is also a solution of the equation. We need only two trial solutions $u_{\alpha_0}(x)$ and $u_{\alpha_1}(x)$, where α_0 and α_1 are two different parameters. The correct solution of the equation is given by

$$u(x) = au_{\alpha_0}(x) + bu_{\alpha_1}(x), \tag{4.63}$$

where a and b are determined from $u(0) = u_0$ and $u(1) = u_1$. Note that $u_{\alpha_0}(0) = u_{\alpha_1}(0) = u(0) = u_0$. So we have

$$a + b = 1, \tag{4.64}$$

$$u_{\alpha_0}(1)a + u_{\alpha_1}(1)b = u_1, \tag{4.65}$$

which lead to

$$a = \frac{u_{\alpha_1}(1) - u_1}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}, \tag{4.66}$$

$$b = \frac{u_1 - u_{\alpha_0}(1)}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}. \tag{4.67}$$

With a and b given by the above equations, we reach the solution of the differential equation from Eq. (4.63).

Here we would like to demonstrate the application of the above scheme to the linear equation problem defined in Eq. (4.58). The following example program is an implementation of the scheme.

```
// An example of solving the boundary-value problem of
// a linear equation via the Runge-Kutta method.

import java.lang.*;
public class LinearDEq {
  static final int n = 100, m = 5;
  public static void main(String argv[]) {
    double y1[] = new double [n+1];
    double y2[] = new double [n+1];
    double y[] = new double [2];
    double h = 1.0/n;

 // Find the 1st solution via the Runge-Kutta method
    y[1]  = 1;
    for (int i=0; i<n; ++i) {
      double x = h*i;
      y = rungeKutta(y, x, h);
      y1[i+1] = y[0];
    }

 // Find the 2nd solution via the Runge-Kutta method
    y[0] = 0;
    y[1] = 2;
    for (int i=0; i<n; ++i) {
      double x = h*i;
      y = rungeKutta(y, x, h);
      y2[i+1] = y[0];
    }

 // Superpose the two solutions found
    double a = (y2[n]-1)/(y2[n]-y1[n]);
    double b = (1-y1[n])/(y2[n]-y1[n]);
    for (int i=0; i<=n; ++i)
      y1[i] = a*y1[i]+b*y2[i];

 // Output the result in every m points
    for (int i=0; i<=n; i+=m)
      System.out.println(y1[i]);
  }

  public static double[] rungeKutta(double y[],
    double t, double dt) {...}

  public static double[] g(double y[], double t) {...}
}
```

Note that we have only integrated the equation twice in the above example program. This is in clear contrast to the general shooting method, in which many more integrations may be needed depending on how fast the solution converges.

An important class of linear equations in physics is known as the Sturm–Liouville problem, defined by

$$[p(x)u'(x)]' + q(x)u(x) = s(x), \tag{4.68}$$

which has the first-order derivative term combined with the second-order derivative term. Here $p(x)$, $q(x)$, and $s(x)$ are the coefficient functions of x. For most actual problems, $s(x) = 0$ and $q(x) = -r(x) + \lambda w(x)$, where λ is the eigenvalue of the equation and $r(x)$ and $w(x)$ are the redefined coefficient functions. The Legendre equation, the Bessel equation, and their related equations in physics are examples of the Sturm–Liouville problem.

Our goal here is to construct an accurate algorithm which can integrate the Sturm–Liouville problem, that is, Eq. (4.68). In Chapter 3, we obtained the three-point formulas for the first-order and second-order derivatives from the combinations

$$\Delta_1 = \frac{u_{i+1} - u_{i-1}}{2h} = u_i' + \frac{h^2 u_i^{(3)}}{6} + O(h^4) \tag{4.69}$$

and

$$\Delta_2 = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = u_i'' + \frac{h^2 u_i^{(4)}}{12} + O(h^4). \tag{4.70}$$

Now if we multiply Eq. (4.69) by p_i' and Eq. (4.70) by p_i and add them together, we have

$$p_i'\Delta_1 + p_i\Delta_2 = (p_i u_i')' + \frac{h^2}{12}\left(p_i u_i^{(4)} + 2p_i' u_i^{(3)}\right) + O(h^4). \tag{4.71}$$

If we replace the first term on the right-hand side with $s_i - q_i u_i$ by applying the original differential equation and drop the second term, we obtain the simplest numerical algorithm for the Sturm–Liouville problem:

$$(2p_i + hp_i')u_{i+1} + (2p_i - hp_i')u_{i-1} = 4p_i u_i + 2h^2(s_i - q_i u_i), \tag{4.72}$$

which is accurate up to $O(h^4)$. Before we discuss how to improve the accuracy of this algorithm, let us work on an illustrating example. The Legendre equation is given by

$$\frac{d}{dx}\left[(1 - x^2)\frac{du}{dx}\right] + l(l+1)u = 0, \tag{4.73}$$

with $l = 0, 1, \ldots, \infty$ and $x \in [-1, 1]$. The solutions of the Legendre equation are the Legendre polynomials $P_l(x)$. Let us assume that we do not know the value of l but know the first two points of $P_1(x) = x$; then we can treat the problem as an eigenvalue problem.

The following program is an implementation of the simplest algorithm for the Sturm–Liouville problem, in combination with the secant method for the root search, to solve for the eigenvalue $l = 1$ of the Legendre equation.

```
// An example of implementing the simplest method to
// solve the Sturm-Liouville problem.

import java.lang.*;
public class Sturm {
  static final int n = 100, ni = 10;
  public static void main(String argv[]) {
    double del = 1e-6, l = 0.5, dl = 0.1;
    l = secant(ni, del, l, dl);

 // Output the eigenvalue obtained
    System.out.println("The eigenvalue is: " + l);
  }

  public static double secant(int n, double del,
    double x, double dx) {...}

// Method to provide the function for the root search.

  public static double f(double l) {
    double u[] = new double[n+1];
    double p[] = new double[n+1];
    double q[] = new double[n+1];
    double s[] = new double[n+1];
    double p1[] = new double[n+1];
    double h = 1.0/n;
    double u0 = 0;
    double u1 = h;

    for (int i=0; i<=n; ++i){
      double x = h*i;
      p[i] = 1-x*x;
      p1[i] = -2*x;
      q[i]  = l*(l+1);
      s[i]  = 0;
    }
    u = sturmLiouville(h, u0, u1, p, p1, q, s);
    return u[n]-1;
  }

// Method to integrate the Sturm-Liouville problem.

  public static double[] sturmLiouville(double h,
    double u0, double u1, double p[], double p1[],
    double q[], double s[]) {
    int n = p.length-1;
    double u[] = new double[n+1];
    double h2 =  h*h;
    u[0] = u0;
    u[1] = u1;
    for (int i=1; i<n; ++i){
      double c2 = 2*p[i]+h*p1[i];
      double c1 = 4*p[i]-2*h2*q[i];
      double c0 = 2*p[i]-h*p1[i];
      double d  = 2*h2*s[i];
      u[i+1] = (c1*u[i]-c0*u[i-1]+d)/c2;
    }
    return u;
  }
}
```

The eigenvalue obtained from the above program is 1.000 000 000 01, which contains an error on the order of 10^{-11}, in comparison with the exact result of $l = 1$. The error is vanishingly small considering the possible rounding error and the inaccuracy in the algorithm. This is accidental of course. The expected error is on the order of 10^{-6}, given by the tolerance set in the program.

Note that the procedure adopted in the above example is quite general and it is the shooting method for the eigenvalue problem. For equations other than the Sturm–Liouville problem, we can follow exactly the same steps.

If we want to have higher accuracy in the algorithm for the Sturm–Liouville problem, we can differentiate Eq. (4.68) twice. Then we have

$$pu^{(4)} + 2p'u^{(3)} = s'' - 3p''u'' - p^{(3)}u' - p'u^{(3)} - q''u - 2q'u' - qu'', \tag{4.74}$$

where $u^{(3)}$ on the right-hand side can be replaced with

$$u^{(3)} = \frac{1}{p}(s' - p''u' - 2p'u'' - q'u - qu'), \tag{4.75}$$

which is the result of taking the first-order derivative of Eq. (4.68). If we combine Eqs. (4.71), (4.74), and (4.75), we obtain a better algorithm:

$$c_{i+1}u_{i+1} + c_{i-1}u_{i-1} = c_i u_i + d_i + O(h^6), \tag{4.76}$$

where c_{i+1}, c_{i-1}, c_i, and d_i are given by

$$\begin{aligned} c_{i+1} &= 24p_i + 12hp_i' + 2h^2q_i + 6h^2p_i'' - 4h^2(p_i')^2/p_i \\ &\quad + h^3p_i^{(3)} + 2h^3q_i' - h^3p_i'q_i/p_i - h^3p_i'p_i''/p_i, && (4.77) \\ c_{i-1} &= 24p_i - 12hp_i' + 2h^2q_i + 6h^2p_i'' - 4h^2(p_i')^2/p_i \\ &\quad - h^3p_i^{(3)} - 2h^3q_i' + h^3p_i'q_i/p_i + h^3p_i'p_i''/p_i, && (4.78) \\ c_i &= 48p_i - 20h^2q_i - 8h^2(p_i')^2/p_i + 12h^2p'' \\ &\quad + 2h^4p_i'q_i'/p_i - 2h^4q_i'', && (4.79) \\ d_i &= 24h^2s_i2h^4s_i'' - 2h^4p_i's_i'/p_i, && (4.80) \end{aligned}$$

which can be evaluated easily if $p(x)$, $q(x)$, and $s(x)$ are explicitly given. When some of the derivatives needed are not easily obtained analytically, we can evaluate them numerically. In order to maintain the high accuracy of the algorithm, we need to use compatible numerical formulas.

For the special case with $p(x) = 1$, the above coefficients reduce to much simpler forms. Without sacrificing the apparently high accuracy of the algorithm, we can apply the three-point formulas to the first- and second-order derivatives of $q(x)$ and $s(x)$. Then we have

$$c_{i+1} = 1 + \frac{h^2}{24}(q_{i+1} + 2q_i - q_{i-1}), \tag{4.81}$$

$$c_{i-1} = 1 + \frac{h^2}{24}(q_{i-1} + 2q_i - q_{i+1}), \tag{4.82}$$

$$c_i = 2 - \frac{5h^2}{60}(q_{i+1} + 8q_i + q_{i-1}), \tag{4.83}$$

$$d_i = \frac{h^2}{12}(s_{i+1} + 10s_i + s_{i-1}), \tag{4.84}$$

which are slightly different from the commonly known Numerov algorithm, which is an extremely accurate scheme for linear differential equations without the first-order derivative term, that is, Eq. (4.62) with $d(x) = 0$ or Eq. (4.68) with $p(x) = 1$. Many equations in physics have this form, for example, the Poisson equation with spherical symmetry or the one-dimensional Schrödinger equation.

The Numerov algorithm is derived from Eq. (4.70) after applying the three-point formula to the second-order derivative and the fourth-order derivative given in the form of a second-order derivative from Eq. (4.68),

$$u^{(4)}(x) = \frac{d^2}{dx^2}[-q(x)u(x) + s(x)]. \tag{4.85}$$

If we apply the three-point formula to the second-order derivative on the right-hand side of the above equation, we obtain

$$u^{(4)}(x) = \frac{(s_{i+1} - q_{i+1}u_{i+1}) - 2(s_i - q_iu_i) + (s_{i-1} - q_{i-1}u_{i-1})}{h^2}. \tag{4.86}$$

Combining the above equation with $\Delta_2 = (u_{i+1} - 2u_i + u_{i-1})/h^2 = u_i'' + h^2u^{(4)}/12$ and $u_i'' = s_i - q_iu_i$, we obtain the Numerov algorithm in the form of Eq. (4.76) with the corresponding coefficients given as

$$c_{i+1} = 1 + \frac{h^2}{12}q_{i+1}, \tag{4.87}$$

$$c_{i-1} = 1 + \frac{h^2}{12}q_{i-1}, \tag{4.88}$$

$$c_i = 2 - \frac{5h^2}{6}q_i, \tag{4.89}$$

$$d_i = \frac{h^2}{12}(s_{i+1} + 10s_i + s_{i-1}). \tag{4.90}$$

Note that even though the apparent local accuracy of the Numerov algorithm and the algorithm we derived earlier in this section for the Sturm–Liouville problem is $O(h^6)$, the actual global accuracy of the algorithm is only $O(h^4)$ because of the repeated use of the three-point formulas. For more discussion on this issue, see Simos (1993). The algorithms discussed here can be applied to initial-value problems as well as to boundary-value or eigenvalue problems. The Numerov algorithm and the algorithm for the Sturm–Liouville problem usually have lower accuracy than the fourth-order Runge–Kutta algorithm when applied to the same problem, and this is different from what the apparent local accuracies imply. For more numerical examples of the Sturm–Liouville problem and the Numerov algorithm in the related problems, see Pryce (1993) and Onodera (1994).

4.9 The one-dimensional Schrödinger equation

The solutions associated with the one-dimensional Schrödinger equation are of importance in understanding quantum mechanics and quantum processes. For example, the energy levels and transport properties of electrons in nanostructures such as quantum wells, dots, and wires are crucial in the development of the next generation of electronic devices. In this section, we will apply the numerical methods that we have introduced so far to solve both the eigenvalue and transport problems defined through the one-dimensional Schrödinger equation

$$-\frac{\hbar^2}{2m}\frac{d^2\phi(x)}{dx^2} + V(x)\phi(x) = \varepsilon\phi(x), \tag{4.91}$$

where m is the mass of the particle, $\hbar$ is the Planck constant, ε is the energy level, $\phi(x)$ is the wavefunction, and $V(x)$ is the external potential. We can rewrite the Schrödinger equation as

$$\phi''(x) + \frac{2m}{\hbar^2}\left[\varepsilon - V(x)\right]\phi(x) = 0, \tag{4.92}$$

which is in the same form as the Sturm–Liouville problem with $p(x) = 1, q(x) = 2m\left[\varepsilon - V(x)\right]/\hbar^2$, and $s(x) = 0$.

The eigenvalue problem

For the eigenvalue problem, the particle is confined by the potential well $V(x)$, so that $\phi(x) \to 0$ with $|x| \to \infty$. A sketch of a typical $V(x)$ is shown in Fig. 4.7. In order to solve this eigenvalue problem, we can integrate the equation with the Numerov algorithm from left to right or from right to left of the potential region. Because the wavefunction goes to zero as $|x| \to \infty$, the integration from one side to another requires integrating from an exponentially increasing region to an oscillatory region and then into an exponentially decreasing region.

The error accumulated will become significant if we integrate the solution from the oscillatory region into the exponentially decreasing region. This is because an exponentially increasing solution is also a possible solution of the equation and can easily enter the numerical integration to destroy the accuracy of the algorithm. The rule of thumb is to avoid integrating into the exponential regions, that is, to carry out the solutions from both sides and then match them in the well region. Usually the matching is done at one of the turning points, where the energy is equal to the potential energy, such as x_{l} and x_{r} in Fig. 4.7. The

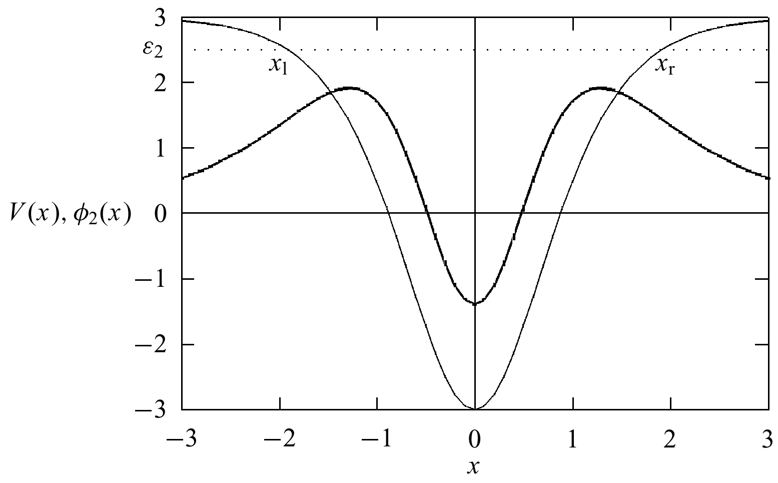


Fig. 4.7 The eigenvalue problem of the one-dimensional Schrödinger equation. Here a potential well $V(x)$ (solid thin line) and the corresponding eigenvalue ε_2 (dotted line) and eigenfunction $\phi_2(x)$ (solid thick line on a relative scale) for the second excited state are illustrated. The turning points x_l and x_r are also indicated.

so-called matching here is to adjust the trial eigenvalue until the solution integrated from the right, $\phi_\mathrm{r}(x)$, and the solution integrated from the left, $\phi_\mathrm{l}(x)$, satisfy the continuity conditions at one of the turning points. If we choose the right turning point as the matching point, the continuity conditions are

$$\phi_\mathrm{l}(x_\mathrm{r}) = \phi_\mathrm{r}(x_\mathrm{r}), \tag{4.93}$$

$$\phi_\mathrm{l}'(x_\mathrm{r}) = \phi_\mathrm{r}'(x_\mathrm{r}). \tag{4.94}$$

If we combine these two conditions, we have

$$\frac{\phi_\mathrm{l}'(x_\mathrm{r})}{\phi_\mathrm{l}(x_\mathrm{r})} = \frac{\phi_\mathrm{r}'(x_\mathrm{r})}{\phi_\mathrm{r}(x_\mathrm{r})}. \tag{4.95}$$

If we use the three-point formula for the first-order derivatives in the above equation, we have

$$\begin{aligned} f(\varepsilon) &= \frac{[\phi_\mathrm{l}(x_\mathrm{r}+h) - \phi_\mathrm{l}(x_\mathrm{r}-h)] - [\phi_\mathrm{r}(x_\mathrm{r}+h) - \phi_\mathrm{r}(x_\mathrm{r}-h)]}{2h\phi(x_\mathrm{r})} \\ &= 0, \end{aligned} \tag{4.96}$$

which can be ensured by a root search scheme. Note that $f(\varepsilon)$ is a function of only ε because $\phi_\mathrm{l}(x_\mathrm{r}) = \phi_\mathrm{r}(x_\mathrm{r}) = \phi(x_\mathrm{r})$ can be used to rescale the wavefunctions.

We now outline the numerical procedure for solving the eigenvalue problem of the one-dimensional Schrödinger equation:

(1) Choose the region of the numerical solution. This region should be large enough compared with the effective region of the potential to have a negligible effect on the solution.
(2) Provide a reasonable guess for the lowest eigenvalue ε_0. This can be found approximately from the analytical result of the case with an infinite well and the same range of well width.

(3) Integrate the equation for $\phi_l(x)$ from the left to the point $x_r + h$ and the one for $\phi_r(x)$ from the right to $x_r - h$. We can choose zero to be the value of the first points of $\phi_l(x)$ and $\phi_r(x)$, and a small quantity to be the value of the second points of $\phi_l(x)$ and $\phi_r(x)$, to start the integration, for example, with the Numerov algorithm. Before matching the solutions, rescale one of them to ensure that $\phi_l(x_r) = \phi_r(x_r)$. For example, we can multiply $\phi_l(x)$ by $\phi_r(x_r)/\phi_l(x_r)$ up to $x = x_r + h$. This rescaling also ensures that the solutions have the correct nodal structure, that is, changing the sign of $\phi_l(x)$ if it is incorrect.
(4) Evaluate $f(\varepsilon_0) = [\phi_r(x_r - h) - \phi_r(x_r + h) - \phi_l(x_r - h) + \phi_l(x_r + h)]/2h\phi_r(x_r)$.
(5) Apply a root search method to obtain ε_0 from $f(\varepsilon_0) = 0$ within a given tolerance.
(6) Carry out the above steps for the next eigenvalue. We can start the search with a slightly higher value than the last eigenvalue. We need to make sure that no eigenstate is missed. This can easily be done by counting the nodes in the solution; the nth state has a total number of n nonboundary nodes, with $n = 0$ for the ground state. A node is where $\phi(x) = 0$. This also provides a way of pinpointing a specific eigenstate.

Now let us look at an actual example with a particle bound in a potential well

$$V(x) = \frac{\hbar^2}{2m}\alpha^2\lambda(\lambda - 1)\left[\frac{1}{2} - \frac{1}{\cosh^2(\alpha x)}\right], \tag{4.97}$$

where both α and λ are given parameters. The Schrödinger equation with this potential can be solved exactly with the eigenvalues

$$\varepsilon_n = \frac{\hbar^2}{2m}\alpha^2\left[\frac{\lambda(\lambda - 1)}{2} - (\lambda - 1 - n)^2\right], \tag{4.98}$$

for $n = 0, 1, 2, \ldots$. We have solved this problem numerically in the region $[-10, 10]$ with 501 points uniformly spaced. The potential well, eigenvalue, and eigenfunction shown in Fig. 4.7 are from this problem with $\alpha = 1$, $\lambda = 4$, and $n = 2$. We have also used $\hbar = m = 1$ in the numerical solution for convenience. The program below implements this scheme.

```
// An example of solving the eigenvalue problem of the
// one-dimensional Schroedinger equation via the secant
// and Numerov methods.

import java.lang.*;
import java.io.*;
public class Schroedinger {
  static final int nx = 500, m = 10, ni = 10;
  static final double x1 = -10, x2 = 10, h = (x2-x1)/nx;
  static int nr, nl;
  static double ul[] = new double[nx+1];
  static double ur[] = new double[nx+1];
  static double ql[] = new double[nx+1];
  static double qr[] = new double[nx+1];
  static double s[] = new double[nx+1];
  static double u[] = new double[nx+1];
  public static void main(String argv[]) throws
    FileNotFoundException {
    double del = 1e-6, e = 2.4, de = 0.1;

 // Find the eigenvalue via the secant search
    e = secant(ni, del, e, de);

 // Output the wavefunction to a file
    PrintWriter w = new PrintWriter
      (new FileOutputStream("wave.data"), true);
    double x = x1;
    double mh = m*h;
    for (int i=0; i<=nx; i+=m) {
      w.println( x + " " + u[i]);
      x += mh;
    }

 // Output the eigenvalue obtained
    System.out.println("The eigenvalue: " + e);
  }

  public static double secant(int n, double del,
    double x, double dx) {...}

// Method to provide the function for the root search.

  public static double f(double x) {
    wave(x);
    double f0 = ur[nr-1]+ul[nl-1]-ur[nr-3]-ul[nl-3];
    return f0/(2*h*ur[nr-2]);
  }

// Method to calculate the wavefunction.

  public static void wave(double energy) {
    double y[] = new double [nx+1];
    double u0 = 0, u1 = 0.01;

 // Set up function q(x) in the equation
    for (int i=0; i<=nx; ++i) {
      double x = x1+i*h;
      ql[i] = 2*(energy-v(x));
      qr[nx-i] = ql[i];
    }

 // Find the matching point at the right turning point
    int im = 0;
    for (int i=0; i<nx; ++i)
      if (((ql[i]*ql[i+1])<0) && (ql[i]>0)) im = i;

 // Carry out the Numerov integrations
    nl = im+2;
    nr = nx-im+2;
    ul = numerov(nl, h, u0, u1, ql, s);
    ur = numerov(nr, h, u0, u1, qr, s);

 // Find the wavefunction on the left
    double ratio = ur[nr-2]/ul[im];
    for (int i=0; i<=im; ++i) {
      u[i] = ratio*ul[i];
      y[i] = u[i]*u[i];
    }
 // Find the wavefunction on the right
    for (int i=0; i<nr-1; ++i) {
      u[i+im] = ur[nr-i-2];
      y[i+im] = u[i+im]*u[i+im];
    }
 // Normalize the wavefunction
    double sum = simpson(y, h);
    sum = Math.sqrt(sum);
    for (int i=0; i<=nx; ++i) u[i] /= sum;
  }

// Method to perform the Numerov integration.

  public static double[] numerov(int m, double h,
    double u0, double u1, double q[], double s[]) {
    double u[] = new double[m];
    u[0] = u0;
    u[1] = u1;
    double g = h*h/12;
    for (int i=1; i<m-1; ++i) {
      double c0 = 1+g*q[i-1];
      double c1 = 2-10*g*q[i];
      double c2 = 1+g*q[i+1];
      double d  = g*(s[i+1]+s[i-1]+10*s[i]);
      u[i+1] = (c1*u[i]-c0*u[i-1]+d)/c2;
    }
    return u;
  }

  public static double simpson(double y[], double h)
    {...}

// Method to provide the given potential in the problem.

  public static double v(double x) {
    double alpha = 1, lambda = 4;
    return alpha*alpha*lambda*(lambda-1)
           *(0.5-1/Math.pow(cosh(alpha*x),2))/2;
  }

// Method to provide the hyperbolic cosine needed.

  public static double cosh(double x) {
    return (Math.exp(x)+Math.exp(-x))/2;
  }
}
```

Running the above program gives $\varepsilon_2 = 2.499\,999$, which is what we expect, comparing with the exact result $\varepsilon_2 = 2.5$, under the given tolerance of 1.0×10^{-6}. Note that we have used the Numerov algorithm with the set of coefficients provided in Eqs. (4.87)–(4.90). We can use the set of coefficients in Eqs. (4.81)–(4.84) and the result will remain the same within the accuracy of the algorithm.

Quantum scattering

Now let us turn to the problem of unbound states, that is, the scattering problem. Assume that the potential is nonzero in the region of $x \in [0, a]$ and that the incident particle comes from the left. We can write the general solution of the Schrödinger equation outside the potential region as

$$\phi(x) = \begin{cases} \phi_1(x) = e^{ikx} + Ae^{-ikx} & \text{for } x < 0, \\ \phi_3(x) = Be^{ik(x-a)} & \text{for } x > a, \end{cases} \tag{4.99}$$

where A and B are the parameters to be determined and k can be found from

$$\varepsilon = \frac{\hbar^2 k^2}{2m}, \tag{4.100}$$

where ε is the energy of the incident particle. The solution $\phi(x) = \phi_2(x)$ in the region of $x \in [0, a]$ can be obtained numerically. During the process of solving $\phi_2(x)$, we will also obtain A and B, which are necessary for calculating the reflectivity $R = |A|^2$ and transmissivity $T = |B|^2$. Note that $T + R = 1$. The boundary conditions at $x = 0$ and $x = a$ are

$$\phi_1(0) = \phi_2(0), \tag{4.101}$$

$$\phi_2(a) = \phi_3(a), \tag{4.102}$$

$$\phi_1'(0) = \phi_2'(0), \tag{4.103}$$

$$\phi_2'(a) = \phi_3'(a), \tag{4.104}$$

which give

$$\phi_2(0) = 1 + A, \tag{4.105}$$

$$\phi_2(a) = B, \tag{4.106}$$

$$\phi_2'(0) = ik(1 - A), \tag{4.107}$$

$$\phi_2'(a) = ikB. \tag{4.108}$$

Note that the wavefunction is now a complex function, as are the parameters A and B. We outline here a combined numerical scheme that utilizes either the Numerov or the Runge–Kutta method to integrate the equation and a minimization scheme to adjust the solution to the desired accuracy. We first outline the scheme through the Numerov algorithm:

(1) For a given particle energy $\varepsilon = \hbar^2k^2/2m$, guess a complex parameter $A = A_\text{r} + iA_\text{i}$. We can use the analytical results for a square potential that has the same range and average strength of the given potential as the initial guess. Because the convergence is very fast, the initial guess is not very important.

(2) Perform the Numerov integration of the Schrödinger equation

$$\phi_2''(x) + \left[k^2 - \frac{2m}{\hbar^2}V(x)\right]\phi_2(x) = 0 \tag{4.109}$$

from $x = 0$ to $x = a$ with the second point given by the Taylor expansion of $\phi_2(x)$ around $x = 0$ to the second order,

$$\phi_2(h) = \phi_2(0) + h\phi_2'(0) + \frac{h^2}{2}\phi_2''(0) + O(h^3), \tag{4.110}$$

where $\phi_2(0) = \phi_1(0) = 1 + A$, $\phi_2'(0) = \phi_1'(0) = ik(1 - A)$, and $\phi_2''(0) = [2mV(0^+)/\hbar^2 - k^2]\phi_2(0) = [2mV(0^+)/\hbar^2 - k^2](1 + A)$. Note that we have used $V(0^+) = \lim_{\delta\to 0} V(\delta)$ for $\delta > 0$. The second-order derivative here is obtained from the Schrödinger equation. Truncation at the first order would also work, but with a little less accuracy.

(3) We can then obtain the approximation for B after the first integration to the point $x = a$ with

$$B = \phi_2(a). \tag{4.111}$$

(4) Using this B value, we can integrate the equation from $x = a$ back to $x = 0$ with the same Numerov algorithm with the second point given by the Taylor expansion of $\phi_2(x)$ around $x = a$ as

$$\phi_2(a - h) = \phi_2(a) - h\phi_2'(a) + \frac{h^2}{2}\phi_2''(a) + O(h^3), \tag{4.112}$$

where $\phi_2(a) = \phi_3(a) = B$ and $\phi_2'(a) = \phi_3'(a) = ikB$ from the continuity conditions. But the second-order derivative is obtained from the equation, and we have $\phi_2''(a) = [2mV(a^-)/\hbar^2 - k^2]\phi_2(a) = [2mV(a^-)/\hbar^2 - k^2]B$. Note that we have used $V(a^-) = \lim_{\delta\to 0} V(a - \delta)$ for $\delta > 0$.

(5) From the backward integration, we can obtain a new $A^{\text{new}} = A_{\text{r}}^{\text{new}} + iA_{\text{i}}^{\text{new}}$ from $\phi_2^{\text{new}}(0) = 1 + A^{\text{new}}$. We can then construct a real function $g(A_{\text{r}}, A_{\text{i}}) = (A_{\text{r}} - A_{\text{r}}^{\text{new}})^2 + (A_{\text{i}} - A_{\text{i}}^{\text{new}})^2$. Note that $A_{\text{r}}^{\text{new}}$ and $A_{\text{i}}^{\text{new}}$ are both the implicit functions of $(A_{\text{r}}, A_{\text{i}})$.

(6) Now the problem becomes an optimization problem of minimizing $g(A_{\text{r}}, A_{\text{i}})$ as A_{r} and A_{i} vary. We can use the steepest-descent scheme introduced in Chapter 3 or other optimization schemes given in Chapter 5.

Now let us illustrate the scheme outlined above with an actual example. Assume that we are interested in the quantum scattering of a double-barrier potential

$$V(x) = \begin{cases} V_0 & \text{if } 0 \le x \le x_1 \quad \text{or} \quad x_2 \le x \le a, \\ 0 & \text{elsewhere.} \end{cases} \tag{4.113}$$

This is a very interesting problem, because it is one of the basic elements

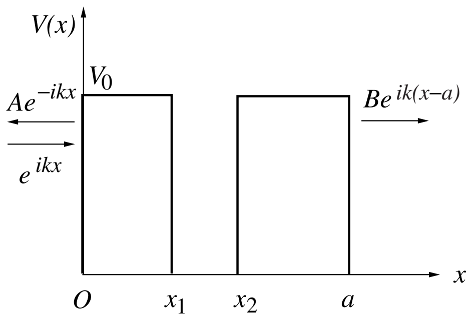


Fig. 4.8 The double-barrier structure of the example. For a system made of GaAs and $Ga_{1-x}Al_xAs$ layers, the barrier regions are formed with $Ga_{1-x}Al_xAs$.

conceived for the next generation of electronic devices. A sketch of the system is given in Fig. 4.8. The problem is solved with the Numerov algorithm and an optimization scheme. We use the steepest-descent method introduced in Chapter 3 in the following implementation.

```
// An example of studying quantum scattering in one
// dimension through the Numerov and steepest-descent
// methods.

import java.lang.*;
import java.io.*;
public class Scattering {
  static final int nx = 100;
  static final double hartree = 0.0116124;
  static final double bohr = 98.964, a = 125/bohr;
  static double e;

  public static void main(String argv[]) throws
    IOException {
    double x[] = new double[2];

// Read in the particle energy
    System.out.println("Enter particle energy: ");
    InputStreamReader c
      = new InputStreamReader(System.in);
    BufferedReader b = new BufferedReader(c);
    String energy = b.readLine();
    e = Double.valueOf(energy).doubleValue();
    e /= hartree;
    x[0] =  0.1;
    x[1] = -0.1;
    double d = 0.1, del = 1e-6;
    steepestDescent(x, d, del);
    double r = x[0]*x[0]+x[1]*x[1];
    double t = 1-r;
    System.out.println("The reflectivity: " + r);
    System.out.println("The transmissivity: " + t);
  }

  public static void steepestDescent(double x[],
    double a, double del) {...}
```

```
// Method to provide function f=gradient g(x).
  public static double[] f(double x[], double h) {...}

// Method to provide function g(x).

  public static double g(double x[]) {
    double h = a/nx;
    double ar = x[0];
    double ai = x[1];
    double ur[] = new double[nx+1];
    double ui[] = new double[nx+1];
    double qf[] = new double[nx+1];
    double qb[] = new double[nx+1];
    double s[] = new double[nx+1];

    for (int i=0; i<=nx; ++i) {
      double xi = h*i;
      s[i] = 0;
      qf[i] = 2*(e-v(xi));
      qb[nx-i] = qf[i];
    }

 // Perform forward integration
    double delta = 1e-6;
    double k = Math.sqrt(2*e);
    double ur0 = 1+ar;
    double ur1 = ur0+k*ai*h
      +h*h*(v(delta)-e)*ur0;
    double ui0 = ai;
    double ui1 = ui0+k*(1-ar)*h
      +h*h*(v(delta)-e)*ui0;
    ur = numerov(nx+1, h, ur0, ur1, qf, s);
    ui = numerov(nx+1, h, ui0, ui1, qf, s);

 // Perform backward integration
    ur0 = ur[nx];
    ur1 = ur0+k*ui[nx]*h
      +h*h*(v(a-delta)-e)*ur0;
    ui0 = ui[nx];
    ui1 = ui0-k*ur[nx]*h
      +h*h*(v(a-delta)-e)*ui0;
    ur = numerov(nx+1, h, ur0, ur1, qb, s);
    ui = numerov(nx+1, h, ui0, ui1, qb, s);

 // Return the function value |a-a_new|*|a-a_new|
    return (1+ar-ur[nx])*(1+ar-ur[nx])
      +(ai-ui[nx])*(ai-ui[nx]);
  }

// Method to provide the potential V(x).

  public static double v(double x) {
    double v0 = 0.3/hartree;
    double x1 = 25/bohr;
    double x2 = 75/bohr;
    if (((x<x1)&&(x>0)) || ((x<a)&&(x>x2)))
      return v0;
    else return 0;
  }
```

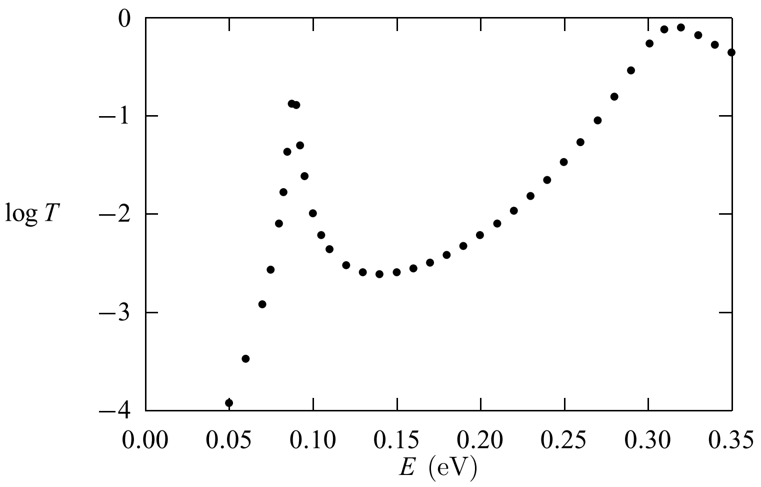


Fig. 4.9 The energy dependence of the transmissivity for a double-barrier potential with a barrier height of 0.3 eV, barrier widths of 25 Å and 50 Å, and the width of the well between the barriers of 50 Å. The transmissivity is plotted on a logarithmic scale.

```
// Method to perform the Numerov integration.
  public static double[] numerov(int m, double h,
    double u0, double u1, double q[], double s[])
    {...}
}
```

We have used the parameters associated with GaAs in the above program with the effective mass of the electron $m = 0.067m_e$ and electric permittivity $\epsilon = 12.53\epsilon_0$, where m_e is the mass of a free electron and ϵ_0 is the electric permittivity of vacuum. Under this choice of effective mass and permittivity, we have given the energy in the unit of the effective Hartree (about 11.6 meV) and length in the unit of the effective Bohr radius (about 99.0 Å).

In Fig. 4.9, we plot the transmissivity of the electron obtained for $V_0 = 0.3$ eV, $x_1 = 25$ Å, $x_2 = 75$ Å, and $a = 125$ Å. Note that there is a significant increase in the transmissivity around a resonance energy $\varepsilon \simeq 0.09$ eV, which is a virtual energy level. The second peak appears at an energy slightly above the barriers. Study of the symmetric barrier case (Pang, 1995) shows that the transmissivity can reach at least 0.999 997 at the resonance energy $\varepsilon \simeq 0.089\,535$ eV, for the case with $x_1 = 50$ Å, $x_2 = 100$ Å, and $a = 150$ Å.

We can also use the Runge–Kutta algorithm to integrate the equation if we choose $y_1(x) = \phi_2(x)$ and $y_2(x) = \phi_2'(x)$. Using the initial conditions for $\phi_2(x)$ and $\phi_2'(x)$ at $x = 0$, we integrate the equation to $x = a$. From $y_1(a)$ and $y_2(a)$, we obtain two different values of B,

$$B_1 = y_1(a), \tag{4.114}$$

$$B_2 = -\frac{i}{k} y_2(a), \tag{4.115}$$

which are both implicit functions of the initial guess of $A = A_r + iA_i$. This means that we can construct an implicit function $g(A_r, A_i) = |B_1 - B_2|^2$ and then optimize it. Note that both B_1 and B_2 are complex, as are the functions $y_1(x)$ and $y_2(x)$. The Runge–Kutta algorithm in this case is much more accurate, because no approximation for the second point is needed. For more details on the application of the Runge–Kutta algorithm and the optimization method in the scattering problem, see Pang (1995).

The procedure can be simplified in the simple potential case, as suggested by R. Zimmermann (personal communication), if we realize that the Schrödinger equation in question is a linear equation. We can take $B = 1$ and then integrate the equation from $x = a$ back to $x = -h$ with either the Numerov or the Runge–Kutta scheme. The solution at $x = 0$ and $x = -h$ satisfies

$$\phi(x) = A_1 e^{ikx} + A_2 e^{-ikx}, \tag{4.116}$$

and we can solve for A_1 and A_2 with the numerical results of $\phi(0)$ and $\phi(-h)$. The reflectivity and transmissivity are then given by $R = |A_2/A_1|^2$ and $T = 1/|A_1|^2$, respectively. Note that it is not now necessary to have the minimization in the scheme. However, for a more general potential, for example, a nonlinear potential, a combination of an integration scheme and an optimization scheme becomes necessary.

Exercises

4.1 Consider two charged particles of masses m_1 and m_2, and charges q_1 and q_2, respectively. They are moving in the xy plane under the influence of a constant electric field $\mathbf{E} = E_0\hat{\mathbf{x}}$ and a constant magnetic induction $\mathbf{B} = B_0\hat{\mathbf{z}}$, where $\hat{\mathbf{x}}$ and $\hat{\mathbf{z}}$ are unit vectors along the positive x and z axes, respectively. Implement the two-point predictor–corrector method to study the system. Is there a parameter region in which the system is chaotic? What happens if each particle encounters a random force that is in the Gaussian distribution?

4.2 Derive the fourth-order Runge–Kutta algorithm for solving the differential equation

$$\frac{dy}{dt} = g(y, t)$$

with a given initial condition. Discuss the options in the selection of the parameters involved.

4.3 Construct a subprogram that solves the differential equation set

$$\frac{d\mathbf{y}}{dt} = \mathbf{g}(\mathbf{y}, t)$$

with the fourth-order Runge–Kutta method with different parameters from those given in the text. Use the total number of components in $\mathbf{y}$ and the initial condition $\mathbf{y}(0) = \mathbf{y}_0$ as the input to the subprogram. Test the fitness of the parameters by comparing the numerical result from the subprogram and the known exact result for the motion of Earth.

4.4 Study the driven pendulum under damping numerically and plot the bifurcation diagram (ω–b plot at a fixed θ) with $q = 1/2$, $\omega_0 = 2/3$, and $b \in [1, 1.5]$.

4.5 Modify the example program for the driven pendulum under damping to study the cases with the driving force changed to a square wave with $f_d(t) = f_0$ for $0 < t < T_0/2$ and $f_d(t) = -f_0$ for $T_0/2 < t < T_0$, and to a triangular wave with $f_d(t) = f_0(2t/T_0 - 1)$ for $0 < t < T_0$, where T_0 is the period of the driving force that repeats in other periods. Is there a parameter region in which the system is chaotic?

4.6 The Duffing model is given by

$$\frac{d^2x}{dt^2} + g\frac{dx}{dt} + x^3 = b\cos t.$$

Write a program to solve the Duffing model in a different parameter region of (g, b). Discuss the behavior of the system from the phase diagram of (x, v), where $v = dx/dt$. Is there a parameter region in which the system is chaotic?

4.7 The Henón–Heiles model is used to describe stellar orbits and is given by a two-dimensional potential

$$V(x, y) = \frac{m\omega^2}{2}(x^2 + y^2) + \lambda\left(x^2y - \frac{y^3}{3}\right),$$

where m is the mass of the star, and ω and λ are two additional model parameters. Derive the differential equation set that describes the motion of the star under the Henón–Heiles model and solve the equation set numerically. In what parameter region does the orbit become chaotic?

4.8 The Lorenz model is used to study climate change and is given by

$$\frac{dy_1}{dt} = a(y_2 - y_1),$$
$$\frac{dy_2}{dt} = (b - y_3)y_1 - y_2,$$
$$\frac{dy_3}{dt} = y_1y_2 - cy_3,$$

where a, b, and c are positive parameters in the model. Solve this model numerically and find the parameter region in which the system is chaotic.

4.9 Consider three objects in the solar system, the Sun, Earth, and Mars. Find the modification of the next period of Earth due to the appearance of Mars starting at the beginning of January 1, 2006.

4.10 Study the dynamics of the two electrons in a classical helium atom. Explore the properties of the system under different initial conditions. Can the system ever be chaotic?

4.11 Apply the Numerov algorithm to solve

$$u''(x) = -4\pi^2 u(x),$$

with $u(0) = 1$ and $u'(0) = 0$. Discuss the accuracy of the result by comparing it with the solution obtained via the fourth-order Runge–Kutta algorithm and with the exact result.

4.12 Apply the shooting method to solve the eigenvalue problem

$$u''(x) = \lambda u(x),$$

with $u(0) = u(1) = 0$. Discuss the accuracy of the result by comparing it with the exact result.

4.13 Develop a program that applies the fourth-order Runge–Kutta and bisection methods to solve the eigenvalue problem of the stationary one-dimensional Schrödinger equation. Find the two lowest eigenvalues and eigenfunctions for an electron in the potential well

$$V(x) = \begin{cases} V_0 \dfrac{x}{x_0} & \text{if } 0 < x < x_0, \\ V_0 & \text{elsewhere.} \end{cases}$$

Atomic units (a.u.), that is, $m_e = e = \hbar = c = 1$, $x_0 = 5$ a.u., and $V_0 = 50$ a.u. can be used.

4.14 Apply the fourth-order Runge–Kutta algorithm to solve the quantum scattering problem in one dimension. The optimization can be achieved with the bisection method through varying A_r and A_i, respectively, within the region $[-1, 1]$. Test the program with the double-barrier potential of Eq. (4.113).

4.15 Find the angle dependence of the angular velocity and the center-of-mass velocity of a meterstick falling with one end on a horizontal plane. Assume that the meterstick is released from an initial angle of $\theta_0 \in [0, \pi/2]$ between the meterstick and the horizontal and that there is kinetic friction at the contact point, with the kinetic friction coefficient $0 \leq \mu_k \leq 0.9$.

4.16 Implement the full-accuracy algorithm for the Sturm–Liouville problem derived in the text in a subprogram. Test it by applying the algorithm to the spherical Bessel equation

$$(x^2 u')' + [x^2 - l(l+1)]u = 0,$$

where $l(l+1)$ are the eigenvalues. Use the known analytic solutions to set up the first two points. Evaluate l numerically and compare it with the exact result. Is the apparent accuracy of $O(h^6)$ in the algorithm delivered?

4.17 Study the dynamics of a compact disk after it is set in rotation on a horizontal table with the rotational axis vertical and along its diameter. Establish a model and set up the relevant equation set that describes the motion of the disk, including the process of the disk falling onto the table. Write a program that solves the equation set. Does the solution describe the actual experiment well?

4.18 The tippe top is a symmetric top that can spin on both ends. When the top is set in rotation on the end that is closer to the center of mass of the top, it will gradually slow down and flip over to spin on the end that is farther away from the center of mass. Establish a model and set up the relevant equation set that describes the motion of the tippe top, including the process of toppling over. Write a program that solves the equation set. Does the solution describe the actual experiment well?

4.19 The system of two coupled rotors is described by the following equation set:

$$\frac{d^2y_1}{dt^2} + \gamma_1 \frac{dy_1}{dt} + \epsilon\left(\frac{dy_1}{dt} - \frac{dy_2}{dt}\right) = f_1(y_1) + F_1(t),$$
$$\frac{d^2y_2}{dt^2} + \gamma_2 \frac{dy_2}{dt} + \epsilon\left(\frac{dy_2}{dt} - \frac{dy_1}{dt}\right) = f_2(y_2) + F_2(t),$$

where ϵ is the coupling constant, $f_{1,2}$ are periodic functions of period 2π, and $F_{1,2} = \alpha_{1,2} + \beta_{1,2}\sin(\omega_{1,2}t + \phi_{1,2})$. Write a program to study this system. Consider a special case with $\alpha_i = \phi_i = 0$, $\gamma_1 = \gamma_2$, $\omega_1 = \omega_2$, and $f_i(x) = f_0 \sin x$. Is there a parameter region in which the system is chaotic?

Chapter 5
Numerical methods for matrices

Matrix operations are involved in many numerical and analytical scientific problems. Schemes developed for the matrix problems can be applied to the related problems encountered in ordinary and partial differential equations. For example, an eigenvalue problem given in the form of a partial differential equation can be rewritten as a matrix problem. A boundary-value problem after discretization is essentially a linear algebra problem.

5.1 Matrices in physics

Many problems in physics can be formulated in a matrix form. Here we give a few examples to illustrate the importance of matrix operations in physics and related fields.

If we want to study the vibrational spectrum of a molecule with n vibrational degrees of freedom, the first step is to investigate the harmonic oscillations of the system by expanding the potential energy up to the second order of the generalized coordinates around the equilibrium structure. Then we have the potential energy

$$U(q_1, q_2, \ldots, q_n) \simeq \frac{1}{2} \sum_{i,j=1}^{n} A_{ij} q_i q_j, \tag{5.1}$$

where q_i are the generalized coordinates and A_{ij} are the elements of the generalized elastic constant matrix that can be obtained through, for example, a quantum chemistry calculation or an experimental measurement. We have taken the equilibrium potential energy as the zero point. Usually the kinetic energy of the system can be expressed as

$$T(\dot{q}_1, \dot{q}_2, \ldots, \dot{q}_n) \simeq \frac{1}{2} \sum_{i,j=1}^{n} M_{ij} \dot{q}_i \dot{q}_j, \tag{5.2}$$

where $\dot{q}_i = dq_i/dt$ are the generalized velocities, and M_{ij} are the elements of the generalized mass matrix whose values depend upon the specifics of the molecule. Now if we apply the Lagrange equation

$$\frac{\partial \mathcal{L}}{\partial q_i} - \frac{d}{dt} \frac{\partial \mathcal{L}}{\partial \dot{q}_i} = 0, \tag{5.3}$$

where $\mathcal{L} = T - U$ is the Lagrangian of the system, we have

$$\sum_{j=1}^{n}(A_{ij}q_j + M_{ij}\ddot{q}_j) = 0, \tag{5.4}$$

for $i = 1, 2, \ldots, n$. If we assume that the time dependence of the generalized coordinates is oscillatory with

$$q_j = x_j e^{-i\omega t}, \tag{5.5}$$

we have

$$\sum_{j=1}^{n}(A_{ij} - \omega^2 M_{ij})x_j = 0, \tag{5.6}$$

for $i = 1, 2, \ldots, n$. This equation can be rewritten in a matrix form:

$$\begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \vdots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \lambda \begin{pmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \vdots & \vdots \\ M_{n1} & \cdots & M_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \tag{5.7}$$

or equivalently,

$$\mathbf{Ax} = \lambda\mathbf{Mx}, \tag{5.8}$$

where $\lambda = \omega^2$ is the eigenvalue and $\mathbf{x}$ is the corresponding eigenvector of the above eigenequation. Note that it is a homogeneous linear equation set. In order to have a nontrivial (not all zero components) solution of the equation set, the determinant of the coefficient matrix has to vanish, that is,

$$|\mathbf{A} - \lambda\mathbf{M}| = 0. \tag{5.9}$$

The roots of this secular equation, λ_k with $k = 1, 2, \ldots, n$, give all the possible vibrational angular frequencies $\omega_k = \sqrt{\lambda_k}$ of the molecule.

Another example we illustrate here deals with the problems associated with electrical circuits. We can apply the Kirchhoff rules to obtain a set of equations for the voltages and currents, and then we can solve the equation set to find the unknowns. Let us take the unbalanced Wheatstone bridge shown in Fig. 5.1 as an example. There is a total of three independent loops. We can choose the first as going through the source and two upper bridges and the second and third as the loops on the left and right of the ammeter. Each loop results in one of three independent equations:

$$r_s i_1 + r_1 i_2 + r_2 i_3 = v_0, \tag{5.10}$$

$$-r_x i_1 + (r_1 + r_x + r_a)i_2 - r_a i_3 = 0, \tag{5.11}$$

$$-r_3 i_1 - r_a i_2 + (r_2 + r_3 + r_a)i_3 = 0, \tag{5.12}$$

where r_1, r_2, r_3, r_x, r_a, and r_s are the resistances of the upper left bridge, upper right bridge, lower right bridge, lower left bridge, ammeter, and external source;

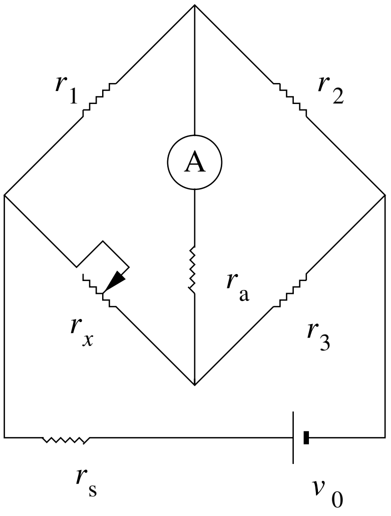


Fig. 5.1 The unbalanced Wheatstone bridge with all the resistors indicated.

i_1, i_2, and i_3 are the currents through the source, upper left bridge, and upper right bridge; and v_0 is the voltage of the external source. The above equation set can be rewritten in a matrix form

$$\mathbf{Ri} = \mathbf{v}, \tag{5.13}$$

where

$$\mathbf{R} = \begin{pmatrix} r_s & r_1 & r_2 \\ -r_x & r_1 + r_x + r_a & -r_a \\ -r_3 & -r_a & r_2 + r_3 + r_a \end{pmatrix} \tag{5.14}$$

is the resistance coefficient matrix, and

$$\mathbf{i} = \begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} \quad \text{and} \quad \mathbf{v} = \begin{pmatrix} v_0 \\ 0 \\ 0 \end{pmatrix} \tag{5.15}$$

are the current and voltage arrays that are in the form of column matrices. If we multiply both sides of the equation by the inverse matrix $\mathbf{R}^{-1}$, we have

$$\mathbf{i} = \mathbf{R}^{-1}\mathbf{v}, \tag{5.16}$$

which is solved if we know $\mathbf{R}^{-1}$. We will see in Section 5.3 that, in general, it is not necessary to know the inverse of the coefficient matrix in order to solve a linear equation set. However, as soon as we know the solution of the linear equation set, we can obtain the inverse of the coefficient matrix using the above equation.

A third example lies the calculation of the electronic structure of a many-electron system. Let us examine a very simple, but still meaningful system, H_3^+. Three protons are arranged in an equilateral triangle. The two electrons in the system are shared by all three protons. Assuming that we can describe the system by a simple Hamiltonian $\mathcal{H}$, containing one term for the hopping of an electron from one site to another and another for the repulsion between the two electrons on a doubly occupied site, we have

$$\mathcal{H} = -t \sum_{i \neq j} a_{i\sigma}^{\dagger} a_{j\sigma} + U \sum n_{i\uparrow} n_{i\downarrow}, \tag{5.17}$$

where $a_{i\sigma}^{\dagger}$ and $a_{j\sigma}$ are creation and annihilation operators of an electron with either spin up ($\sigma = \uparrow$) or spin down ($\sigma = \downarrow$), and $n_{i\sigma} = a_{i\sigma}^{\dagger} a_{i\sigma}$ is the corresponding occupancy at the ith site. This Hamiltonian is called the Hubbard model when it is used to describe highly correlated electronic systems. The parameters t and U in the Hamiltonian can be obtained from either a quantum chemistry calculation or experimental measurement. The Schrödinger equation for the system is

$$\mathcal{H}|\Psi_k\rangle = \mathcal{E}_k|\Psi_k\rangle, \tag{5.18}$$

where $\mathcal{E}_k$ and $|\Psi_k\rangle$ are the kth eigenvalue and eigenstate of the Hamiltonian. Because each site has only one orbital, we have a total of 15 possible states for the two electrons under the single-particle representation,

$$\begin{aligned}
&|\phi_1\rangle = a_{1\uparrow}^{\dagger} a_{1\downarrow}^{\dagger}|0\rangle, \ |\phi_2\rangle = a_{2\uparrow}^{\dagger} a_{2\downarrow}^{\dagger}|0\rangle, \ |\phi_3\rangle = a_{3\uparrow}^{\dagger} a_{3\downarrow}^{\dagger}|0\rangle, \\
&|\phi_4\rangle = a_{1\uparrow}^{\dagger} a_{2\downarrow}^{\dagger}|0\rangle, \ |\phi_5\rangle = a_{1\uparrow}^{\dagger} a_{3\downarrow}^{\dagger}|0\rangle, \ |\phi_6\rangle = a_{2\uparrow}^{\dagger} a_{3\downarrow}^{\dagger}|0\rangle, \\
&|\phi_7\rangle = a_{1\downarrow}^{\dagger} a_{2\uparrow}^{\dagger}|0\rangle, \ |\phi_8\rangle = a_{1\downarrow}^{\dagger} a_{3\uparrow}^{\dagger}|0\rangle, \ |\phi_9\rangle = a_{2\downarrow}^{\dagger} a_{3\uparrow}^{\dagger}|0\rangle, \\
&|\phi_{10}\rangle = a_{1\uparrow}^{\dagger} a_{2\uparrow}^{\dagger}|0\rangle, \ |\phi_{11}\rangle = a_{1\uparrow}^{\dagger} a_{3\uparrow}^{\dagger}|0\rangle, \ |\phi_{12}\rangle = a_{2\uparrow}^{\dagger} a_{3\uparrow}^{\dagger}|0\rangle, \\
&|\phi_{13}\rangle = a_{1\downarrow}^{\dagger} a_{2\downarrow}^{\dagger}|0\rangle, \ |\phi_{14}\rangle = a_{1\downarrow}^{\dagger} a_{3\downarrow}^{\dagger}|0\rangle, \ |\phi_{15}\rangle = a_{2\downarrow}^{\dagger} a_{3\downarrow}^{\dagger}|0\rangle,
\end{aligned} \tag{5.19}$$

with $|0\rangle$ is the vacuum state. Then we can cast the Schrödinger equation as a matrix equation

$$\mathbf{H}\mathbf{\Psi}_k = \mathcal{E}_k \mathbf{\Psi}_k, \tag{5.20}$$

where $H_{ij} = \langle\phi_i|\mathcal{H}|\phi_j\rangle$ with $i, j = 1, 2, \ldots, 15$, and $\Psi_{ki} = \langle\phi_i|\Psi_k\rangle$. We leave it as an exercise for the reader to figure out the Hamiltonian elements. The 15 roots of the secular equation

$$|\mathbf{H} - \mathcal{E}\mathbf{I}| = 0 \tag{5.21}$$

are the eigenenergies of the system. Here $\mathbf{I}$ is a unit matrix with $I_{ij} = \delta_{ij}$. Note that the quantum problem here has a mathematical structure similar to that of the classical problem of molecular vibrations. We could have simplified the problem by exploiting the symmetries of the system. The total spin and the z component of the total spin commute with the Hamiltonian; thus they are good quantum numbers. We can reduce the Hamiltonian matrix into block-diagonal form with a maximum block size of 2×2. After we obtain all the eigenvalues and eigenvectors of the system, we can analyze the electronic, optical, and magnetic properties of H_3^+ easily.

5.2 Basic matrix operations

An $n \times m$ matrix $\mathbf{A}$ is defined through its elements A_{ij} with the row index $i = 1, 2, \ldots, n$ and the column index $j = 1, 2, \ldots, m$. It is called a square matrix if $n = m$. We will consider mainly the problems associated with square matrices in this chapter.

A variable array $\mathbf{x}$ with elements $x_1, x_2, \ldots, x_n$ arranged into a column is viewed as an $n \times 1$ matrix, or an n-element column matrix. A typical set of linear algebraic equations is given by

$$\sum_{j=1}^{n} A_{ij} x_j = b_i, \tag{5.22}$$

for $i = 1, 2, \ldots, n$, where x_j are the unknowns to be solved, A_{ij} are the given coefficients, and b_i are the given constants. Equation (5.22) can be written as

$$\mathbf{Ax} = \mathbf{b}, \tag{5.23}$$

with $\mathbf{Ax}$ defined from the standard matrix multiplication

$$C_{ij} = \sum_{k} A_{ik} B_{kj}, \tag{5.24}$$

for $\mathbf{C} = \mathbf{AB}$. The summation over k requires the number of columns of the first matrix to be the same as the number of rows of the second matrix. Otherwise, the product does not exist. Basic matrix operations are extremely important. For example, the inverse of a square matrix $\mathbf{A}$ (written as $\mathbf{A}^{-1}$) is defined by

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}, \tag{5.25}$$

where $\mathbf{I}$ is a unit matrix with the elements $I_{ij} = \delta_{ij}$. The determinant of an $n \times n$ matrix $\mathbf{A}$ is defined as

$$|\mathbf{A}| = \sum_{i=1}^{n} (-1)^{i+j} A_{ij} |\mathbf{R}_{ij}| \tag{5.26}$$

for any $j = 1, 2, \ldots, n$, where $|\mathbf{R}_{ij}|$ is the determinant of the residual matrix $\mathbf{R}_{ij}$ of $\mathbf{A}$ with its ith row and jth column removed. The combination $C_{ij} = (-1)^{i+j}|\mathbf{R}_{ij}|$ is called a *cofactor* of $\mathbf{A}$. The determinant of a 1×1 matrix is the element itself. The determinant of a triangular matrix is the product of the diagonal elements. In principle, the inverse of $\mathbf{A}$ can be obtained through

$$A^{-1}{}_{ij} = \frac{C_{ji}}{|\mathbf{A}|}. \tag{5.27}$$

If a matrix has an inverse or nonzero determinant, it is called a nonsingular matrix. Otherwise, it is a singular matrix.

The trace of a matrix $\mathbf{A}$ is the sum of all its diagonal elements, written as

$$\mathrm{Tr}\,\mathbf{A} = \sum_{i=1}^{n} A_{ii}. \tag{5.28}$$

The transpose of a matrix $\mathbf{A}$ (written as $\mathbf{A}^{\mathrm{T}}$) has elements with the row and column indices of $\mathbf{A}$ interchanged, that is,

$$A^{\mathrm{T}}_{ij} = A_{ji}. \tag{5.29}$$

We call $\mathbf{A}$ an orthogonal matrix if $\mathbf{A}^{\mathrm{T}} = \mathbf{A}^{-1}$. The complex conjugate of $\mathbf{A}^{\mathrm{T}}$ is called the Hermitian operation of $\mathbf{A}$ (written as $\mathbf{A}^{\dagger}$) with $A^{\dagger}_{ij} = A^{*}_{ji}$. We call $\mathbf{A}$ a Hermitian matrix if $\mathbf{A}^{\dagger} = \mathbf{A}$ and a unitary matrix if $\mathbf{A}^{\dagger} = \mathbf{A}^{-1}$.

Another useful matrix operation is to add one row (or column) multiplied by a factor λ to another row (or column), such as

$$A'_{ij} = A_{ij} + \lambda A_{kj}, \tag{5.30}$$

for $j = 1, 2, \dots, n$, where i and k are row indices, which can be different or the same. This operation preserves the determinant, that is, $|\mathbf{A}'| = |\mathbf{A}|$, and can be represented by a matrix multiplication

$$\mathbf{A}' = \mathbf{MA}, \tag{5.31}$$

where $\mathbf{M}$ has unit diagonal elements plus a single nonzero off-diagonal element $M_{ik} = \lambda$. If we interchange any two rows or columns of a matrix, its determinant changes only its sign.

A matrix eigenvalue problem, such as the problem associated with the electronic structure of H_3^+, is defined by the equation

$$\mathbf{Ax} = \lambda \mathbf{x}, \tag{5.32}$$

where $\mathbf{x}$ and λ are an eigenvector and its corresponding eigenvalue of the matrix, respectively. Note that this includes the eigenvalue problem associated with molecular vibrations, $\mathbf{Ax} = \lambda \mathbf{Mx}$, if we take $\mathbf{M}^{-1}\mathbf{A} = \mathbf{B}$ as the matrix of the problem. Then we have $\mathbf{Bx} = \lambda \mathbf{x}$.

The matrix eigenvalue problem can also be viewed as a linear equation set problem solved iteratively. For example, if we want to solve the eigenvalues and eigenvectors of the above equation, we can carry out the recursion

$$\mathbf{Ax}^{(k+1)} = \lambda^{(k)} \mathbf{x}^{(k)}, \tag{5.33}$$

where $\lambda^{(k)}$ and $\mathbf{x}^{(k)}$ are the approximate eigenvalue and eigenvector at the kth iteration. This is known as the *iteration method*. In Section 5.5 we will discuss a related scheme in more detail, but here we just want to point out that the problem is equivalent to the solution of the linear equation set at each iteration. We can also show that the eigenvalues of a matrix under a similarity transformation

$$\mathbf{B} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S} \tag{5.34}$$

are the same as the ones of the original matrix **A** because

$$\mathbf{B}\mathbf{y} = \lambda\mathbf{y} \tag{5.35}$$

is equivalent to

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}, \tag{5.36}$$

if we choose $\mathbf{x} = \mathbf{S}\mathbf{y}$. This, of course, assumes that matrix **S** is nonsingular. It also means that

$$|\mathbf{B}| = |\mathbf{A}| = \prod_{i=1}^{n} \lambda_i. \tag{5.37}$$

These basic aspects of matrix operations are quite important and will be used throughout this chapter. Readers who are not familiar with these aspects should consult one of the standard textbooks, for example, Lewis (1991).

5.3 Linear equation systems

A matrix is called an upper-triangular (lower-triangular) matrix if the elements below (above) the diagonal are all zero. The simplest scheme for solving matrix problems is *Gaussian elimination*, which uses very basic matrix operations to transform the coefficient matrix into an upper (lower) triangular one first and then finds the solution of the equation set by means of backward (forward) substitutions. The inverse and the determinant of a matrix can also be obtained in such a manner.

Let us take the solution of the linear equation set

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{5.38}$$

as an illustrative example. If we assume that $|\mathbf{A}| \neq 0$ and $\mathbf{b} \neq \mathbf{0}$, then the system has a unique solution. In order to show the steps of the Gaussian elimination, we will label the original matrix $\mathbf{A} = \mathbf{A}^{(0)}$ with $\mathbf{A}^{(j)}$ being the resultant matrix after j matrix operations. Similar notation is used for the transformed **b** as well.

Gaussian elimination

The basic idea of Gaussian elimination is to transform the original linear equation set to one that has an upper-triangular or lower-triangular coefficient matrix, but has the same solution. Here we want to transform the coefficient matrix into an upper triangular matrix. We can interchange the roles of the rows and the columns if we want to transform the coefficient matrix into a lower triangular one. In each step of transformation, we eliminate the elements of a column that are not part of the upper triangle. The final linear equation set is then given by

$$\mathbf{A}^{(n-1)}\mathbf{x} = \mathbf{b}^{(n-1)}, \tag{5.39}$$

with $A_{ij}^{(n-1)} = 0$ for $i > j$. The procedure is quite simple. We first multiply the first equation by $-A_{i1}^{(0)}/A_{11}^{(0)}$, that is, all the elements of the first row in the coefficient matrix and $b_1^{(0)}$, and then add it to the ith equation for $i > 1$. The first element of every row except the first row in the coefficient matrix is then eliminated. Now we denote the new matrix $\mathbf{A}^{(1)}$ and multiply the second equation by $-A_{i2}^{(1)}/A_{22}^{(1)}$. Then we add the modified second equation to all the other equations with $i > 2$, the second element of every row except the first row and the second row of the coefficient matrix is eliminated. This procedure can be continued with the third, fourth, ..., and $(n-1)$th equations, and then the coefficient matrix becomes an upper-triangular matrix $\mathbf{A}^{(n-1)}$. A linear equation set with an upper-triangular coefficient matrix can easily be solved with backward substitutions.

Because all the diagonal elements are used in the denominators, the scheme would fail if any of them happened to be zero or a very small quantity. This problem can be circumvented in most cases by interchanging the rows and/or columns to have the elements used for divisions being the ones with largest magnitudes possible. This is the so-called pivoting procedure. This procedure will not change the solutions of the linear equation set but will put them in a different order. Here we will consider only the partial-pivoting scheme, which searches for the pivoting element (the one used for the next division) only from the remaining elements of the given column. A full-pivoting scheme searches for the pivoting element from all the remaining elements. After considering both the computing speed and accuracy, the partial-pivoting scheme seems to be a good compromise.

We first search for the element with the largest magnitude from $|A_{i1}^{(0)}|$ for $i = 1, 2, \ldots, n$. Assuming that the element obtained is $A_{k_1 1}^{(0)}$, we then interchange the first row and the k_1th row and eliminate the first element of each row except the first row. Similarly, we can search for the second pivoting element with the largest magnitude from $|A_{i2}^{(1)}|$ for $i = 2, 3, \ldots, n$. Assuming that the element obtained is $A_{k_2 2}^{(1)}$ with $k_2 > 1$, we can then interchange the second row and the k_2th row and eliminate the second element of each row except the first and second rows. This procedure is continued to complete the elimination and transform the original matrix into an upper-triangular matrix.

Physically, we do not need to interchange the rows of the matrix when searching for pivoting elements. An index can be used to record the order of the pivoting elements taken from all the rows. Furthermore, the ratios $A_{ij}^{(j-1)}/A_{k_j j}^{(j-1)}$ for $i > j$ are also needed to modify $\mathbf{b}^{(j-1)}$ into $\mathbf{b}^{(j)}$ and can be stored in the same two-dimensional array that stores matrix $\mathbf{A}^{(n-1)}$ at the locations where $A_{ij}^{(n-1)} = 0$, that is, the locations below the diagonal. When we determine the pivoting element, we also rescale the element from each row by the largest element magnitude of that row in order to have a fair comparison. This rescaling also reduces some potential rounding errors. The following method is an implementation of the partial-pivoting Gaussian elimination outlined above.

```
// Method to carry out the partial-pivoting Gaussian
// elimination.  Here index[] stores pivoting order.

  public static void gaussian(double a[][],
    int index[]) {
    int n = index.length;
    double c[] = new double[n];

 // Initialize the index
    for (int i=0; i<n; ++i) index[i] = i;

 // Find the rescaling factors, one from each row
    for (int i=0; i<n; ++i) {
      double c1 = 0;
      for (int j=0; j<n; ++j) {
        double c0 = Math.abs(a[i][j]);
        if (c0 > c1) c1 = c0;
      }
      c[i] = c1;
    }

 // Search for the pivoting element from each column
    int k = 0;
    for (int j=0; j<n-1; ++j) {
      double pi1 = 0;
      for (int i=j; i<n; ++i) {
        double pi0 = Math.abs(a[index[i]][j]);
        pi0 /= c[index[i]];
        if (pi0 > pi1) {
          pi1 = pi0;
          k = i;
        }
      }

   // Interchange rows according to the pivoting order
      int itmp = index[j];
      index[j] = index[k];
      index[k] = itmp;
      for (int i=j+1; i<n; ++i) {
        double pj = a[index[i]][j]/a[index[j]][j];

     // Record pivoting ratios below the diagonal
        a[index[i]][j] = pj;

     // Modify other elements accordingly
        for (int l=j+1; l<n; ++l)
          a[index[i]][l] -= pj*a[index[j]][l];
      }
    }
  }
```

The above method destroys the original matrix. If we want to preserve it, we can just duplicate it in the method and work on the new matrix and return it after completing the procedure as done in the Lagrange interpolation programs for the data array in Chapter 2.

Determinant of a matrix

The determinant of the original matrix can easily be obtained after we have transformed it into an upper-triangular matrix through Gaussian elimination. As we have pointed out, the procedure used in the partial-pivoting Gaussian elimination does not change the value of the determinant only its sign, which can be fixed with the knowledge of the order of pivoting elements. For an upper-triangular matrix, the determinant is given by the product of all its diagonal elements. Therefore, we can obtain the determinant of a matrix as soon as it is transformed into an upper-triangular matrix. Here is an example of obtaining the determinant of a matrix with the partial-pivoting Gaussian elimination.

```
// An example of evaluating the determinant of a matrix
// via the partial-pivoting Gaussian elimination.

import java.lang.*;
public class  Det {
  public static void main(String argv[]) {
    double a[][]= {{ 1, -3,  2, -1, -2},
                   {-2,  2, -1,  2,  3},
                   { 3, -3, -2,  1, -1},
                   { 1, -2,  1, -3,  2},
                   {-3, -1,  2,  1, -3}};
    double d = det(a);
    System.out.println("The determinant is: " + d);
  }

// Method to evaluate the determinant of a matrix.

  public static double det(double a[][]) {
    int n = a.length;
    int index[] = new int[n];

 // Transform the matrix into an upper triangle
    gaussian(a, index);

 // Take the product of the diagonal elements
    double d = 1;
    for (int i=0; i<n; ++i) d = d*a[index[i]][i];

 // Find the sign of the determinant
    int sgn = 1;
    for (int i=0; i<n; ++i) {
      if (i != index[i]) {
        sgn = -sgn;
        int j = index[i];
        index[i] = index[j];
        index[j] = j;
      }
    }
    return sgn*d;
  }

  public static void gaussian(double a[][],
    int index[]) {...}
}
```

We obtain the determinant $|\mathbf{A}| = 262$ after running the above program. Note that we have used the recorded pivoting order to obtain the correct sign of the determinant. The method developed here is quite efficient and powerful and can be used in real problems that require repeated evaluation of the determinant of a large matrix, such as the local energy for a Fermi system in the variational and diffusion quantum Monte Carlo simulations, to be discussed in Chapter 10.

Solution of a linear equation set

Similarly, we can solve a linear equation set after transforming its coefficient matrix into an upper-triangular matrix through Gaussian elimination. The solution is then obtained through backward substitutions with

$$x_i = \frac{1}{A^{(n-1)}_{k_i i}} \left(b^{(n-1)}_{k_i} - \sum_{j=i+1}^{n} A^{(n-1)}_{k_i j} x_j \right), \tag{5.40}$$

for $i = n-1, n-2, \ldots, 1$, starting with $x_n = b^{(n-1)}_{k_n} / A^{(n-1)}_{k_n n}$. We have used k_i as the row index of the pivoting element from the ith column. We can easily show that the above recursion satisfies the linear equation set $\mathbf{A}^{(n-1)}\mathbf{x} = \mathbf{b}^{(n-1)}$. Here we demonstrate it using the example of finding the currents in the Wheatstone bridge, highlighted in Section 5.1 with $r_s = r_1 = r_2 = r_x = r_a = 100\ \Omega$ and $v_0 = 200$ V.

```
// An example of solving a linear equation set via the
// partial-pivoting Gaussian elimination.

import java.lang.*;
public class LinearEq {
  public static void main(String argv[]) {
    int n = 3;
    double x[] = new double[n];
    double b[] = {200,0,0};
    double a[][]= {{ 100,  100,  100},
                   {-100,  300, -100},
                   {-100, -100,  300}};
    int index[] = new int[n];
    x = solve(a, b, index);

    for (int i=0; i<n; i++)
      System.out.println("I_" + (i+1) + " = " + x[i]);
  }

// Method to solve the equation a[][] x[] = b[] with
// the partial-pivoting Gaussian elimination.

  public static double[] solve(double a[][],
    double b[], int index[]) {
    int n = b.length;
    double x[] = new double[n];

 // Transform the matrix into an upper triangle
    gaussian(a, index);

 // Update the array b[i] with the ratios stored
    for(int i=0; i<n-1; ++i) {
      for(int j =i+1; j<n; ++j) {
        b[index[j]] -= a[index[j]][i]*b[index[i]];
      }
    }

 // Perform backward substitutions
    x[n-1] = b[index[n-1]]/a[index[n-1]][n-1];
    for (int i=n-2; i>=0; --i) {
      x[i] = b[index[i]];
      for (int j=i+1; j<n; ++j) {
        x[i] -= a[index[i]][j]*x[j];
      }
      x[i] /= a[index[i]][i];
    }

 // Put x[i] into the correct order
    order(x,index);

    return x;
  }

  public static void gaussian(double a[][], int index[])
    {...}

// Method to sort an array x[i] from the lowest to the
// highest value of index[i].

  public static void order(double x[], int index[]){
    int m = x.length;
    for (int i = 0; i<m; ++i) {
      for (int j = i+1; j<m; ++j) {
        if (index[i] > index[j]) {
          int itmp = index[i];
          index[i] = index[j];
          index[j] = itmp;
          double xtmp = x[i];
          x[i] = x[j];
          x[j] = xtmp;
        }
      }
    }
  }
}
```

We obtain $i_1 = 1$ A and $i_2 = i_3 = 0.5$ A after running the above program.

Inverse of a matrix

The inverse of a matrix **A** can be obtained in exactly the same fashion if we realize that

$$A^{-1}{}_{ij} = x_{ij}, \tag{5.41}$$

for $i, j = 1, 2, \ldots, n$, where x_{ij} is the solution of the equation $\mathbf{A}\mathbf{x}_j = \mathbf{b}_j$, with $b_{ij} = \delta_{ij}$. If we expand **b** into a unit matrix in the program for solving a linear equation set, the solution corresponding to each column of the unit matrix forms the corresponding column of $\mathbf{A}^{-1}$. With a little modification of the above routine for solving the linear equation set, we obtain the program for matrix inversion.

```
// An example of performing matrix inversion through the
// partial-pivoting Gaussian elimination.
import java.lang.*;
public class Inverse {
  public static void main(String argv[]) {
    double a[][]= {{ 100,  100,  100},
                   {-100,  300, -100},
                   {-100, -100,  300}};
    int n = a.length;
    double d[][] = invert(a);
    for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
        System.out.println(d[i][j]);
  }

  public static double[][] invert(double a[][]) {
    int n = a.length;
    double x[][] = new double[n][n];
    double b[][] = new double[n][n];
    int index[] = new int[n];
    for (int i=0; i<n; ++i) b[i][i] = 1;

 // Transform the matrix into an upper triangle
    gaussian(a, index);

 // Update the matrix b[i][j] with the ratios stored
    for (int i=0; i<n-1; ++i)
      for (int j=i+1; j<n; ++j)
        for (int k=0; k<n; ++k)
          b[index[j]][k]
            -= a[index[j]][i]*b[index[i]][k];

 // Perform backward substitutions
    for (int i=0; i<n; ++i) {
      x[n-1][i] = b[index[n-1]][i]/a[index[n-1]][n-1];
      for (int j=n-2; j>=0; --j) {
        x[j][i] = b[index[j]][i];
        for (int k=j+1; k<n; ++k) {
          x[j][i] -= a[index[j]][k]*x[k][i];
        }
        x[j][i] /= a[index[j]][j];
      }
    }
  return x;
  }

  public static void gaussian(double a[][], int index[])
    {...}
}
```

After running the above program, we obtain the inverse of the input matrix used in the program. Even though we have only used a very simple matrix to illustrate the method, the scheme itself is efficient enough for most real problems.

LU decomposition

A scheme related to the Gaussian elimination is called *LU decomposition*, which splits a nonsingular matrix into the product of a lower-triangular and an upper-triangular matrix. A simple example of the LU decomposition of a tridiagonal matrix was introduced in Section 2.4 in association with the cubic spline. Here we examine the general case.

As pointed out earlier in this section, the Gaussian elimination corresponds to a set of $n-1$ matrix operations that can be represented by a matrix multiplication

$$\mathbf{A}^{(n-1)} = \mathbf{MA}, \tag{5.42}$$

where

$$\mathbf{M} = \mathbf{M}^{(n-1)}\mathbf{M}^{(n-2)}\cdots\mathbf{M}^{(1)}, \tag{5.43}$$

with each $\mathbf{M}^{(r)}$, for $r = 1, 2, \ldots, n-1$, completing one step of the elimination. It is easy to show that $\mathbf{M}$ is a lower-triangular matrix with all the diagonal elements being -1 and $M_{k_i j} = -A^{(j-1)}_{k_i j}/A^{(j-1)}_{k_j j}$ for $i > j$. So Eq. (5.42) is equivalent to

$$\mathbf{A} = \mathbf{LU}, \tag{5.44}$$

where $\mathbf{U} = \mathbf{A}^{(n-1)}$ is an upper-triangular matrix and $\mathbf{L} = \mathbf{M}^{-1}$ is a lower-triangular matrix. Note that the inverse of a lower-triangular matrix is still a lower-triangular matrix. Furthermore, because $M_{ii} = -1$, we must have $L_{ii} = 1$ and $L_{ij} = -M_{ij} = A^{(j-1)}_{k_i j}/A^{(j-1)}_{k_j j}$ for $i > j$, which are the ratios stored in the matrix in the method introduced earlier in this section to perform the partial-pivoting Gaussian elimination. The method performs the LU decomposition with nonzero elements of $\mathbf{U}$ and nonzero, nondiagonal elements of $\mathbf{L}$ stored in the returned matrix. The choice of $L_{ii} = 1$ in the LU decomposition is called the Doolittle factorization. An alternative is to choose $U_{ii} = 1$; this is called the Crout factorization. The Crout factorization is equivalent to rescaling U_{ij} by U_{ii} for $i < j$ and multipling L_{ij} by U_{jj} for $i > j$ from the Doolittle factorization.

In general the elements in $\mathbf{L}$ and $\mathbf{U}$ can be obtained from comparison of the elements of the product of $\mathbf{L}$ and $\mathbf{U}$ with the elements of $\mathbf{A}$ in Eq. (5.44). The result can be cast into a recursion with

$$L_{ij} = \frac{1}{U_{jj}}\left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}\right), \tag{5.45}$$

$$U_{ij} = \frac{1}{L_{ii}}\left(A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj}\right), \tag{5.46}$$

where $L_{i1} = A_{i1}/U_{11}$ and $\mathrm{U}_{1j} = A_{1j}/L_{11}$ are the starting values. If we implement the LU decomposition following the above recursion, we still need to consider the issue of pivoting, which shows up in the denominator of the expression for U_{ij} or L_{ij}. We can manage it in exactly the same way as in the Gaussian elimination. In practice, we can always store both $\mathbf{L}$ and $\mathbf{U}$ in one matrix, with the elements below the diagonal for **L**, those above the diagonal for **U**, and the diagonal elements for either U_{ii} or L_{ii} depending on which of them is not equal to 1.

The determinant of **A** is given by the product of the diagonal elements of **L** and **U** because

$$|\mathbf{A}| = |\mathbf{L}||\mathbf{U}| = \prod_{i=1}^{n} L_{ii} U_{ii}. \tag{5.47}$$

As soon as we obtain **L** and **U** for a given matrix **A**, we can solve the linear equation set $\mathbf{Ax} = \mathbf{b}$ with one set of forward substitutions and another set of backward substitutions because the linear equation set can now be rewritten as

$$\mathbf{Ly} = \mathbf{b}, \tag{5.48}$$

$$\mathbf{Ux} = \mathbf{y}. \tag{5.49}$$

If we compare the elements on both sides of these equations, we have

$$y_i = \frac{1}{L_{ii}} \left(b_i - \sum_{k=1}^{i-1} L_{ik} y_k \right), \tag{5.50}$$

with $y_1 = b_1/L_{11}$ and $i = 2, 3, \ldots, n$. Then we can obtain the solution of the original linear equation set from

$$x_i = \frac{1}{U_{ii}} \left(y_i - \sum_{k=i+1}^{n} U_{ik} x_k \right), \tag{5.51}$$

for $i = n-1, n-2, \ldots, 1$, starting with $x_n = y_n/U_{nn}$.

We can also obtain the inverse of a matrix by a method similar to that used in the matrix inversion through the Gaussian elimination. We can choose a set of vectors with elements $b_{ij} = \delta_{ij}$ with $i, j = 1, 2, \ldots, n$. Here i refers to the element and j to a specific vector. The inverse of **A** is then given by $A^{-1}{}_{ij} = x_{ij}$ solving from $\mathbf{Ax}_j = \mathbf{b}_j$.

5.4 Zeros and extremes of multivariable functions

The numerical solution of a linear equation set discussed in the preceding section is important in computational physics and scientific computing because many problems there appear in the form of a linear equation set. A few examples of them are given in the exercises for this chapter.

However, there is another class of problems that are nonlinear in nature but can be solved iteratively with the linear schemes just developed. Examples include the solution of a set of nonlinear multivariable equations and a search for the maxima or minima of a multivariable function. In this section, we will show how to extend the matrix methods discussed so far to study nonlinear problems. We will also demonstrate the application of these numerical schemes to an actual physics problem, the stable geometric configuration of a multicharge system, which is relevant in the study of the geometry of molecules and the formation of solid salts. From the numerical point of view, this is also a problem of searching for the global or a local minimum on a potential energy surface.

The multivariable Newton method

Nonlinear equations can also be solved with matrix techniques. In Chapter 3, we introduced the Newton method in obtaining the zeros of a single-variable function. Now we would like to extend our study to the solution of a set of multivariable equations. Assume that the equation set is given by

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \tag{5.52}$$

with $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ and $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. If the equation has at least one solution at $\mathbf{x}_\mathrm{r}$, we can perform the Taylor expansion around a neighboring point $\mathbf{x}$ with

$$\mathbf{f}(\mathbf{x}_\mathrm{r}) = \mathbf{f}(\mathbf{x}) + \Delta\mathbf{x} \cdot \boldsymbol{\nabla}\mathbf{f}(\mathbf{x}) + O(\Delta\mathbf{x}^2) \simeq \mathbf{0}, \tag{5.53}$$

where $\Delta\mathbf{x} = \mathbf{x}_\mathrm{r} - \mathbf{x}$. This is a linear equation set that can be cast into a matrix form

$$\mathbf{A}\Delta\mathbf{x} = \mathbf{b}, \tag{5.54}$$

where

$$A_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j} \tag{5.55}$$

and $b_i = -f_i$. Next we can find the root of the nonlinear equation set iteratively by carrying out the solutions of Eq. (5.54) at every point of iteration. The Newton method is then given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k, \tag{5.56}$$

where $\Delta\mathbf{x}_k$ is the solution of Eq. (5.54) with $\mathbf{x} = \mathbf{x}_k$. The Newton method for a single-variable equation discussed in Chapter 3 is the special case with $n = 1$.

As with the secant method discussed in Chapter 3, if the expression for the first-order derivative is not easy to obtain, the two-point formula can be used for the partial derivative

$$A_{ij} = \frac{f_i(\mathbf{x} + h_j\hat{\mathbf{x}}_j) - f_i(\mathbf{x})}{h_j}, \tag{5.57}$$

where h_j is a finite interval and $\hat{\mathbf{x}}_j$ is the unit vector along the direction of x_j. A rule of thumb in practice is to choose

$$h_j \simeq \delta_0 x_j, \tag{5.58}$$

where δ_0 is roughly the square root of the tolerance of the floating-point data used in the calculation; it is related to the total number of bits allocated for that data type either by the programming language or by the processor of the computer. For example, the tolerance is about $2^{-31} \approx 5 \times 10^{-10}$ for the 32-bit floating-point data.

As a matter of fact, both the Newton and the secant method converge locally if the function is smooth enough, and the main computing cost is in solving the linear equation set defined in Eq. (5.54). Here we demonstrate the scheme using the following example in which we find the root of the set of two equations, $f_1(x, y) = e^{x^2} \ln y - x^2 = 0$ and $f_2(x, y) = e^{y^2} \ln x - y^2 = 0$, around $x = y = 1.5$.

```
// An example of searching for the root via the secant method
// for f_i[x_k] for i=1,2,...,n.

import java.lang.*;
public class Rootm {
  public static void main(String argv[]) {
    int n = 2;
    int ni = 10;
    double del = 1e-6;
    double x[] = {1.5, 1.5};
    secantm(ni, x, del);

 // Output the root obtained
    System.out.println("The root is at x = " + x[0]
      + "; y = " + x[1]);
  }

// Method to carry out the multivariable secant search.

  public static void secantm(int ni, double x[],
    double del) {
    int n = x.length;
    double h = 2e-5;
    int index[] = new int[n];
    double a[][] = new double[n][n];

    int k = 0;
    double dx = 0.1;
    while ((Math.abs(dx)>del) && (k<ni)) {
      double b[] = f(x);
      for (int i=0; i<n; ++i) {
        for (int j=0; j<n; ++j) {
          double hx = x[j]*h;
          x[j] += hx;
          double c[] = f(x);
          a[i][j] = (c[i]-b[i])/hx;
        }
      }
      for (int i=0; i<n; ++i) b[i] = -b[i];
      double d[] = solve(a, b, index);
      dx = 0;
      for (int i=0; i<n; ++i) {
        dx += d[i]*d[i];
        x[i] += d[i];
      }
      dx = Math.sqrt(dx/n);
      k++;
    }
    if (k==n) System.out.println("Convergence not" +
      " found after " + ni + " iterations");
  }

  public static double[] solve(double a[][], double b[],
    int index[]) {...}

  public static void gaussian(double a[][], int index[])
    {...}

// Method to provide function f_i[x_k].

  public static double[] f(double x[]) {
    double fx[] = new double[2];
    fx[0] = Math.exp(x[0]*x[0])*Math.log(x[1])
            -x[0]*x[0];
    fx[1] = Math.exp(x[1])*Math.log(x[0])-x[1]*x[1];
    return fx;
  }
}
```

Running the above program we obtain the root, which is at $x \simeq 1.559\,980$ and $y \simeq 1.238\,122$. The accuracy can be improved if we reduce the tolerance further.

Extremes of a multivariable function

Knowing the solution of a nonlinear equation set, we can develop numerical schemes to obtain the minima or maxima of a multivariable function. The extremes of a multivariable function $g(\mathbf{x})$ are the solutions of a nonlinear equation set

$$\mathbf{f}(\mathbf{x}) = \boldsymbol{\nabla} g(\mathbf{x}) = \mathbf{0}, \tag{5.59}$$

where $\mathbf{f}(\mathbf{x})$ is an n-dimensional vector with each component given by a partial derivative of $g(\mathbf{x})$.

In Chapter 3, we introduced the steepest-descent method in the search of an extreme of a multivariable function. We can also use the multivariable Newton or secant method introduced above for such a purpose, except that special care is needed to ensure that $g(\mathbf{x})$ decreases (increases) during the updates for a minimum (maximum) of $g(\mathbf{x})$. For example, if we want to obtain a minimum of $g(\mathbf{x})$, we can update the position vector $\mathbf{x}$ following Eq. (5.56). We can also modify the matrix defined in Eq. (5.55) to

$$A_{ij} = \frac{\partial f_i}{\partial x_j} + \mu\delta_{ij}, \tag{5.60}$$

with μ taken as a small positive quantity to ensure that the modified matrix $\mathbf{A}$ is positive definite, that is, $\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} \geq 0$ for any nonzero $\mathbf{w}$. This will force the updates always to move in the direction of decreasing $g(\mathbf{x})$.

A special choice of μ, known as the BFGS (Broyden, 1970; Fletcher, 1970; Goldfarb, 1970; Shanno, 1970) updating scheme, is to have

$$\mathbf{A}_k = \mathbf{A}_{k-1} + \frac{\mathbf{y}\mathbf{y}^{\mathrm{T}}}{\mathbf{y}^{\mathrm{T}}\mathbf{w}} - \frac{\mathbf{A}_{k-1}\mathbf{w}\mathbf{w}^{\mathrm{T}}\mathbf{A}_{k-1}}{\mathbf{w}^{\mathrm{T}}\mathbf{A}_{k-1}\mathbf{w}}, \tag{5.61}$$

where

$$\mathbf{w} = \mathbf{x}_k - \mathbf{x}_{k-1} \tag{5.62}$$

and

$$\mathbf{y} = \mathbf{f}_k - \mathbf{f}_{k-1}. \tag{5.63}$$

The BFGS scheme ensures that the updating matrix is positive definite; thus the search is always moving toward a minimum of $g(\mathbf{x})$. This scheme has been very successful in many practical problems. The reason behind its success is still unclear. If we want to find the maximum of $g(\mathbf{x})$, we can carry out exactly the same steps to find the minimum of $-g(\mathbf{x})$ with $\mathbf{f}(\mathbf{x}) = -\boldsymbol{\nabla} g(\mathbf{x})$. If the gradient is not available analytically, we can use the finite difference instead, which is in the spirit of the secant method. For more details on the optimization of a function, see Dennis and Schnabel (1983). Much work has been done to develop a better numerical method for optimization of a multivariable function in the last few decades. However, the problem of global optimization of a multivariable function with many local minima is still open and may never be solved. For some discussions on the problem, see Wenzel and Hamacher (1999), Wales and Scheraga (1999), and Baletto *et al.* (2004).

Geometric structures of multicharge clusters

We now turn to a physics problem, the stable geometric structure of a multicharge cluster, which is extremely important in the analysis of small clusters of atoms, ions, and molecules. We will take clusters of $(\mathrm{Na}^+)_l(\mathrm{Cl}^-)_m$ with small positive integers l and m as illustrative examples. We will study geometric structures of the clusters with $2 \leq n = l + m \leq 6$. If we take the empirical form of the interaction potential between two ions in the NaCl crystal,

$$V(r_{ij}) = \eta_{ij}\frac{e^2}{4\pi\epsilon_0 r_{ij}} + \delta_{ij} V_0 e^{-r_{ij}/r_0}, \tag{5.64}$$

where $\eta_{ij} = -1$ and $\delta_{ij} = 1$ if the particles are of opposite charges; otherwise, $\eta_{ij} = 1$ and $\delta_{ij} = 0$. Here V_0 and r_0 are empirical parameters, which can be determined from either experiment or first-principles calculations. For solid NaCl, $V_0 = 1.09 \times 10^3$ eV and $r_0 = 0.321$ Å (Kittel, 1995). We will use these two quantities in what follows.

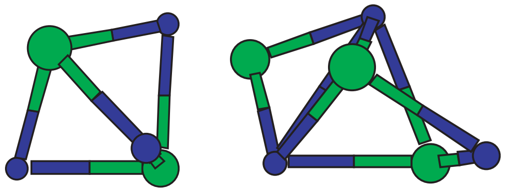


Fig. 5.2 Top view of stable structures of $Na^+(NaCl)_2$ (left) and $(NaCl)_3$ (right).

The function to be optimized is the total interaction potential energy of the system,

$$U(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n) = \sum_{i>j}^{n} V(r_{ij}). \tag{5.65}$$

A local optimal structure of the cluster is obtained when U reaches a local minimum. This minimum cannot be a global minimum for large clusters, because no relaxation is allowed and a large cluster can have many local minima. There are $3n$ coordinates as independent variables in $U(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n)$. However, because the position of the center of mass and rotation around the center of mass do not change the total potential energy U, we remove the center-of-mass motion and the rotational motion around the center of mass during the optimization. We can achieve this by imposing several restrictions on the cluster. First, we fix one of the particles at the origin of the coordinates. This removes three degrees of freedom of the cluster. Second, we restrict another particle on an axis, for example, the x axis. This removes two more degrees of freedom of the cluster. Finally, we restrict the third particle in a plane, for example, the xy plane. This removes the last (sixth) degree of freedom in order to fix the center of mass and make the system irrotational. The potential energy U now only has $3n - 6$ independent variables (coordinates).

The BFGS scheme has been applied to search for the local minima of U for NaCl, $Na^+(NaCl)$, $(NaCl)_2$, $Na^+(NaCl)_2$, and $(NaCl)_3$. The stable structures for the first three systems are very simple: a dimer, a straight line, and a square. The stable structures obtained for $Na^+(NaCl)_2$ and $(NaCl)_3$ are shown in Fig. 5.2. In order to avoid reaching zero separation of any two ions, a term $b(c/r_{ij})^{12}$ has been added in the interaction, with $b = 1$ eV and $c = 0.1$ Å, which adds on an amount of energy on the order of 10^{-18} eV to the total potential energy. It is worth pointing out that the structure of $(NaCl)_3$ is similar to the structure of $(H_2O)_6$ discovered by Liu *et al.* (1996). This might be an indication that water molecules are actually partially charged because of the intermolecular polarization.

5.5 Eigenvalue problems

Matrix eigenvalue problems are very important in physics. They are defined by

$$\mathbf{Ax} = \lambda \mathbf{x}, \tag{5.66}$$

where λ is the eigenvalue corresponding to the eigenvector $\mathbf{x}$ of the matrix $\mathbf{A}$, determined from the secular equation

$$|\mathbf{A} - \lambda\mathbf{I}| = 0, \tag{5.67}$$

where $\mathbf{I}$ is a unit matrix. An $n \times n$ matrix has a total of n eigenvalues. The eigenvalues do not have to be different. If two or more eigenstates have the same eigenvalue, they are degenerate. The degeneracy of an eigenvalue is the total number of the corresponding eigenstates.

The eigenvalue problem is quite general in physics, because the matrix in Eq. (5.66) can come from many different problems, for example, the Lagrange equation for the vibrational modes of a large molecule, or the Schrödinger equation for H_3^+ discussed in Section 5.1. In this section, we will discuss the most common and useful methods for obtaining eigenvalues and eigenvectors in a matrix eigenvalue problem.

Eigenvalues of a Hermitian matrix

In many problems in physics and related fields, the matrix in question is Hermitian with

$$\mathbf{A}^\dagger = \mathbf{A}. \tag{5.68}$$

The simplicity of the Hermitian eigenvalue problem is due to three important properties associated with a Hermitian matrix:

(1) the eigenvalues of a Hermitian matrix are all real;
(2) the eigenvectors of a Hermitian matrix can be made orthonormal;
(3) a Hermitian matrix can be transformed into a diagonal matrix with the same set of eigenvalues under a similarity transformation of a unitary matrix that contains all its eigenvectors.

Furthermore, the eigenvalue problem of an $n \times n$ complex Hermitian matrix is equivalent to that of a $2n \times 2n$ real symmetric matrix. We can show this easily. For a complex matrix, we can always separate its real part from its imaginary part with

$$\mathbf{A} = \mathbf{B} + i\mathbf{C}, \tag{5.69}$$

where $\mathbf{B}$ and $\mathbf{C}$ are the real and imaginary parts of $\mathbf{A}$, respectively. If $\mathbf{A}$ is Hermitian, then $\mathbf{B}$ is a real symmetric matrix and $\mathbf{C}$ is a real skew symmetric matrix; they satisfy

$$B_{ij} = B_{ji}, \tag{5.70}$$

$$C_{ij} = -C_{ji}. \tag{5.71}$$

If we decompose the eigenvector $\mathbf{z}$ in a similar fashion, we have $\mathbf{z} = \mathbf{x} + i\mathbf{y}$. The original eigenvalue problem becomes

$$(\mathbf{B} + i\mathbf{C})(\mathbf{x} + i\mathbf{y}) = \lambda(\mathbf{x} + i\mathbf{y}), \tag{5.72}$$

which is equivalent to

$$\begin{pmatrix} \mathbf{B} & -\mathbf{C} \\ \mathbf{C} & \mathbf{B} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}. \tag{5.73}$$

This is a real symmetric eigenvalue problem with the same set of eigenvalues that have an overall double degeneracy. Therefore we need to solve only the real symmetric eigenvalue problem if the matrix is Hermitian.

The matrix used in similarity transformation of a real symmetric matrix to a diagonal matrix is orthogonal instead of unitary. This simplifies the procedure considerably. The problem can be simplified further if we separate the procedure into two steps:

(1) use an orthogonal matrix to perform a similarity transformation of the real symmetric matrix defined in Eq. (5.73) into a real symmetric tridiagonal matrix;
(2) solve the eigenvalue problem of the resulting tridiagonal matrix.

Note that the similarity transformation preserves the eigenvalues of the original matrix, and the eigenvectors are the columns or rows of the orthogonal matrix used in the transformation. The most commonly used method for tridiagonalizing a real symmetric matrix is the *Householder method*. Sometimes the method is also called the *Givens method*. Here we give a brief description of the scheme. The tridiagonalization is achieved with a total of $n - 2$ consecutive transformations, each operating on a row and a column of the matrix. The transformations can be cast into a recursion

$$\mathbf{A}^{(k)} = \mathbf{O}_k^{\mathrm{T}} \mathbf{A}^{(k-1)} \mathbf{O}_k, \tag{5.74}$$

for $k = 1, 2, \ldots, n - 2$, where $\mathbf{O}_k$ is an orthogonal matrix that works on the row elements with $i = k + 2, \ldots, n$ of the kth column and the column elements with $j = k + 2, \ldots, n$ of the kth row. The recursion begins with $\mathbf{A}^{(0)} = \mathbf{A}$. To be specific, we can write the orthogonal matrix as

$$\mathbf{O}_k = \mathbf{I} - \frac{1}{\eta_k} \mathbf{w}_k \mathbf{w}_k^{\mathrm{T}}, \tag{5.75}$$

where the lth component of the vector $\mathbf{w}_k$ is given by

$$w_{kl} = \begin{cases} 0 & \text{for } l \leq k, \\ A_{kk+1}^{(k-1)} + \alpha_k & \text{for } l = k + 1, \\ A_{kl}^{(k-1)} & \text{for } l \geq k + 2, \end{cases} \tag{5.76}$$

with

$$\alpha_k = \pm \sqrt{\sum_{l=k+1}^{n} \left[A_{kl}^{(k-1)} \right]^2} \tag{5.77}$$

and

$$\eta_k = \alpha_k \left[\alpha_k + A_{kk+1}^{(k-1)} \right]. \tag{5.78}$$

Even though the sign of α_k is arbitrary in the above equations, it is always taken to be the same as that of $A_{kk+1}^{(k-1)}$ in practice in order to avoid any possible cancelation, which can make the algorithm ill-conditioned (with a zero denominator in $\mathbf{O}_k$). We can show that $\mathbf{O}_k$ defined above is the desired matrix, which is orthogonal and converts the row elements with $i = k+2, \ldots, n$ for the kth column and the column elements with $j = k+2, \ldots, n$ for the kth row of $\mathbf{A}^{(k-1)}$ to zero just by comparing the elements on the both sides of Eq. (5.74). We are not going to provide a program here for the Householder scheme for tridiagonalizing a real symmetric matrix because it is in all standard mathematical libraries and reference books, for example, Press *et al.* (2002).

After we obtain the tridiagonalized matrix, the eigenvalues can be found using one of the root search routines available. Note that the secular equation $|\mathbf{A} - \lambda \mathbf{I}| = 0$ is equivalent to a polynomial equation $p_n(\lambda) = 0$. Because of the simplicity of the symmetric tridiagonal matrix, the polynomial $p_n(\lambda)$ can be generated recursively with

$$p_i(\lambda) = (a_i - \lambda) p_{i-1}(\lambda) - b_{i-1}^2 p_{i-2}(\lambda), \tag{5.79}$$

where $a_i = A_{ii}$ and $b_i = A_{ii+1} = A_{i+1i}$. The polynomial $p_i(\lambda)$ is given from the submatrix of A_{jk} with $j, k = 1, 2, \ldots, i$ with the starting values $p_0(\lambda) = 1$ and $p_1(\lambda) = a_1 - \lambda$. Note that this recursion is similar to that of the orthogonal polynomials introduced in Section 2.2 and can be generated easily using the following method.

```
// Method to generate the polynomial for the secular
// equation of a symmetric tridiagonal matrix.

  public static double polynomial(double a[],
    double b[], double x, int i) {
    if (i==0) return 1;
     else if (i==1) return a[0]-x;
      else return (a[i-1]-x)*polynomial(a, b, x, i-1)
        -b[i-2]*b[i-2]*polynomial(a, b, x, i-2);
  }
```

In principle, we can use any of the root searching routines to find the eigenvalues from the secular equation as soon as the polynomial is generated. However, two properties associated with the zeros of $p_n(\lambda)$ are useful in developing a fast and accurate routine to obtain the eigenvalues of a symmetric tridiagonal matrix.

(1) All the roots of $p_n(\lambda) = 0$ lie in the interval $[-\|\mathbf{A}\|, \|\mathbf{A}\|]$, where

$$\|\mathbf{A}\| = \max\left\{ \sum_{j=1}^{n} |A_{ij}| \right\}, \tag{5.80}$$

for $i = 1, 2, \ldots, n$, is the column modulus of the matrix. We can also use the row modulus of the matrix in the above statement. The row modulus of a matrix is given by

$$\|\tilde{\mathbf{A}}\| = \max\left\{\sum_{i=1}^{n} |A_{ij}|\right\}, \tag{5.81}$$

for $j = 1, 2, \ldots, n$.

(2) The number of roots for $p_n(\lambda) = 0$ with $\lambda \geq \lambda_0$ is given by the number of agreements of the signs of $p_j(\lambda_0)$ and $p_{j-1}(\lambda_0)$ for $j = 1, 2, \ldots, n$. If any of the polynomials, for example, $p_j(\lambda_0)$, is zero, the sign of the previous polynomial $p_{j-1}(\lambda_0)$ is assigned to that polynomial.

With the help of these properties, we can develop a quite simple but fast algorithm in connection with the bisection method discussed in Chapter 3 to obtain the eigenvalues of a real symmetric matrix. We outline this algorithm below.

We first evaluate the column modulus of the matrix with Eq. (5.80). This sets the boundaries for the eigenvalues. Note that each summation in Eq. (5.80) has only two or three terms, $|A_{ii-1}|$, $|A_{ii}|$, and $|A_{ii+1}|$. We can then start the search for the first eigenvalue in the region of $[-\|\mathbf{A}\|, \|\mathbf{A}\|]$. Because there is a total of n eigenvalues, we need to decide which of those are sought. For example, if we are only interested in the ground state, in other words, the lowest eigenvalue, we can design an algorithm to target it directly.

A specific scheme can be devised and altered, based on the following general procedure. We can bisect the maximum region of $[-\|\mathbf{A}\|, \|\mathbf{A}\|]$ and evaluate the signs of $p_i(0)$ for $i = 1, 2, \ldots, n$. Note that $p_0(\lambda) = 1$. Based on what we have discussed, we will know the number of eigenvalues lying within either $[-\|\mathbf{A}\|, 0]$ or $[0, \|\mathbf{A}\|]$. This includes any possible degeneracy. We can divide the two subintervals into four equal regions and check the signs of the polynomials at each bisected point. This procedure can be continued to narrow down the region where each eigenvalue resides. After l steps of bisection, each eigenvalue sought is narrowed down to a region with a size of $\|\mathbf{A}\|/2^{l-1}$. Note that we can work either on a specific eigenvalue, for example, the one associated with the ground state, or on a group of eigenvalues simultaneously. The errorbars are bounded by $\|\mathbf{A}\|/2^{l-1}$ after l steps of bisection. A more realistic estimate of the errorbar is obtained from the improvement of a specific eigenvalue at each step of bisection, as in the estimates, made in Chapter 3.

Because the Householder scheme for a real symmetric matrix is carried out in two steps, transformation of the original matrix into a tridiagonal matrix and solution of the eigenvalue problem of the tridiagonal matrix, we can design different algorithms to achieve each of these two steps separately, depending on the specific problems involved. Interested readers can find several of these algorithms in Wilkinson (1963; 1965). It is worth emphasizing again that a Hermitian matrix eigenvalue problem can be converted into a real symmetric matrix eigenvalue problem with the size of the matrix expanded to twice that of the original matrix along each direction. This seems to be a reasonable approach in most cases, as long as the size of the matrix is not limited by the available resources.

Eigenvalues of general matrices

Even though most problems in physics are likely to be concerned with Hermitian matrices, there are still problems that require us to deal with general matrices. Here we discuss briefly how to obtain the eigenvalues of a general nondefective matrix. A matrix is nondefective if it can be diagonalized under a matrix transformation and its eigenvectors can form a complete vector space. Here we consider nondefective matrices only, Because there is always a well-defined eigenvalue equation $\mathbf{Ax} = \lambda\mathbf{x}$ for a nondefective matrix $\mathbf{A}$, and we seldom encounter defective matrices in physics.

In most cases, we can define a matrix function $f(\mathbf{A})$ with $\mathbf{A}$ playing the same role as x in $f(x)$. For example,

$$f(\mathbf{A}) = e^{-\alpha\mathbf{A}} = \sum_{k=0}^{\infty} \frac{(-\alpha\mathbf{A})^k}{k!} \tag{5.82}$$

is similar to

$$f(x) = e^{-\alpha x} = \sum_{k=0}^{\infty} \frac{(-\alpha x)^k}{k!}, \tag{5.83}$$

except that the power of a matrix is treated as matrix multiplications with $\mathbf{A}^0 = \mathbf{I}$. When the matrix function operates on an eigenvector of the matrix, the matrix can be replaced by the corresponding eigenvalue, such as

$$f(\mathbf{A})\mathbf{x}_i = f(\lambda_i)\mathbf{x}_i, \tag{5.84}$$

where $\mathbf{x}_i$ and λ_i satisfy $\mathbf{Ax}_i = \lambda_i\mathbf{x}_i$. If the function involves the inverse of the matrix and the eigenvalue happens to be zero, we can always add a term $\eta\mathbf{I}$ to the original matrix to remove the singularity. The modified matrix has the eigenvalue $\lambda_i - \eta$ for the corresponding eigenvector $\mathbf{x}_i$. The eigenvalue of the original matrix is recovered by taking $\eta \to 0$ after the problem is solved. Based on this property of nondefective matrices, we can construct a recursion

$$\mathbf{x}^{(k)} = \frac{1}{\sqrt{\mathcal{N}_k}}(\mathbf{A} - \mu\mathbf{I})^{-1}\mathbf{x}^{(k-1)} \tag{5.85}$$

to extract the eigenvalue that is closest to the parameter μ. Here $k = 1, 2, \ldots,$ and $\mathcal{N}_k$ is a normalization constant to ensure that

$$\left\langle \mathbf{x}^{(k)} \middle| \mathbf{x}^{(k)} \right\rangle = \left(\mathbf{x}^{(k)}\right)^{\dagger} \mathbf{x}^{(k)} = 1. \tag{5.86}$$

To analyze this recursive procedure, let us express the trial state $\mathbf{x}^{(0)}$ in terms of a linear combination of all the eigenstates with

$$\mathbf{x}^{(0)} = \sum_{i=1}^{n} a_i^{(0)}\mathbf{x}_i, \tag{5.87}$$

which is always possible, because the eigenstates form a complete vector space. If we substitute the above trial state into the recursion, we have

$$\mathbf{x}^{(k)} = \frac{1}{\sqrt{\prod_{j=1}^{k} \mathcal{N}_j}} \sum_{i=1}^{n} \frac{a_i^{(0)} \mathbf{x}_i}{(\lambda_i - \mu)^k} = \sum_{i=1}^{n} a_i^{(k)} \mathbf{x}_i. \tag{5.88}$$

If we normalize the state at each step of iteration, only the state $\mathbf{x}_j$, with the eigenvalue λ_j that is the closest to μ, survive after a large number of iterations. All other coefficients vanish approximately as

$$a_i^{(k)} \sim \left(\frac{\lambda_j - \mu}{\lambda_i - \mu} \right)^k \frac{a_i^{(0)}}{a_j^{(0)}} \tag{5.89}$$

for $i \neq j$, after k iterations. However, $a_j^{(k)}$ will grow with k to approach 1. The corresponding eigenvalue is obtained with

$$\lambda_j = \mu + \lim_{k \to \infty} \frac{1}{\sqrt{\mathcal{N}_k}} \frac{x_l^{(k-1)}}{x_l^{(k)}}, \tag{5.90}$$

where l is the index for a specific nonzero component (usually the one with the maximum magnitude) of $\mathbf{x}^{(k)}$ or $\mathbf{x}^{(k-1)}$.

Here $\mathbf{x}^{(0)}$ is a trial state, which should have a nonzero overlap with the eigenvector $\mathbf{x}_j$. We have to be very careful to make sure the overlap is nonzero. Otherwise, we end up with the eigenvalue that belongs to the eigenvector with a nonzero overlap with $\mathbf{x}^{(0)}$ but which is still closer to μ than the rest. One way to avoid a zero overlap of $\mathbf{x}^{(0)}$ with $\mathbf{x}_j$ is to generate each component of $\mathbf{x}^{(0)}$ with a random-number generator and check each result with at least two different trial states.

The method outlined here is usually called the *inverse iteration method.* It is quite straightforward if we have an efficient algorithm to perform the matrix inversion. We can also rewrite the recursion at each step as a linear equation set

$$\sqrt{\mathcal{N}_k}(\mathbf{A} - \mu\mathbf{I})\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)}, \tag{5.91}$$

which can be solved with Gaussian elimination, for example, from one iteration to another. Note that this iterative scheme also solves the eigenvectors of the matrix at the same time. This is a very useful feature in many applications for which the eigenvectors and eigenvalues are both needed.

Eigenvectors of matrices

We sometimes also need the eigenvectors of an eigenequation. For a nondefective matrix, we can always obtain a complete set of eigenvectors, including the degenerate eigenvalue cases.

First, if the matrix is symmetric, it is much easier to obtain its eigenvectors. In principle, we can transform a real symmetric matrix $\mathbf{A}$ into a tridiagonal matrix $\mathbf{T}$ with a similarity transformation

$$\mathbf{T} = \mathbf{O}^{\mathrm{T}}\mathbf{A}\mathbf{O}, \tag{5.92}$$

where $\mathbf{O}$ is an orthogonal matrix. The eigenvalue equation then becomes

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} = \mathbf{O}\mathbf{T}\mathbf{O}^{\mathrm{T}}\mathbf{x}, \tag{5.93}$$

which is equivalent to solving $\mathbf{y}$ from

$$\mathbf{T}\mathbf{y} = \lambda\mathbf{y} \tag{5.94}$$

and then $\mathbf{x}$ is obtained from

$$\mathbf{x} = \mathbf{O}\mathbf{y}. \tag{5.95}$$

Thus, we can develop a practical scheme that solves the eigenvectors of $\mathbf{T}$ first and then those of $\mathbf{A}$ by multiplying $\mathbf{y}$ with $\mathbf{O}$, which is a byproduct of any tridiagonalization scheme, such as the Householder scheme.

As we discussed earlier, the recursion

$$\sqrt{\mathcal{N}_k}(\mathbf{T} - \mu\mathbf{I})\mathbf{y}^{(k)} = \mathbf{y}^{(k-1)} \tag{5.96}$$

will lead to both the eigenvector $\mathbf{y}_j$ and the eigenvalue λ_j that is closest to μ. Equation (5.96) can be solved with an elimination scheme. Note that the elimination scheme is now extremely simple because there are no more than three nonzero elements in each row. In every step of the above recursion, we need to normalize the resulting vector. We can use the LU decomposition in solving a linear equation set with a tridiagonal coefficient matrix, as given in detail in Section 2.4.

If the eigenvalue corresponds to more than one eigenvector, we can obtain the other vectors simply by changing the value of μ or the initial guess of $\mathbf{y}^{(0)}$. When all the vectors $\mathbf{x}_k$ corresponding to the same eigenvalue are found, we can transform them into an orthogonal set $\mathbf{z}_k$ with

$$\mathbf{z}_k = \mathbf{x}_k - \sum_{j=1}^{k-1}\langle\mathbf{z}_j|\mathbf{x}_k\rangle\mathbf{z}_j = \mathbf{x}_k - \sum_{j=1}^{k-1}\left(\mathbf{z}_j^{\mathrm{T}}\mathbf{x}_k\right)\mathbf{z}_j, \tag{5.97}$$

which is known as the Gram–Schmidt orthogonalization procedure.

If the tridiagonal matrix is obtained by means of the Householder scheme, we can also obtain the eigenvectors of the original matrix at the same time using

$$\begin{aligned}\mathbf{x} &= \mathbf{O}_1\mathbf{O}_2\cdots\mathbf{O}_{n-2}\mathbf{y}\\ &= \left(\mathbf{I} - \frac{1}{\eta_1}\mathbf{w}_1\mathbf{w}_1^{\mathrm{T}}\right)\cdots\left(\mathbf{I} - \frac{1}{\eta_{n-2}}\mathbf{w}_{n-2}\mathbf{w}_{n-2}^{\mathrm{T}}\right)\mathbf{y},\end{aligned} \tag{5.98}$$

which can be carried out in a straightforward fashion if we realize that

$$\left(\mathbf{I} - \frac{1}{\eta_k}\mathbf{w}_k\mathbf{w}_k^{\mathrm{T}}\right)\mathbf{y} = \mathbf{y} - \beta_k\mathbf{w}_k, \tag{5.99}$$

with $\beta_k = \mathbf{w}_k^{\mathrm{T}}\mathbf{y}/\eta_k$. Note that the first k elements in $\mathbf{w}_k$ are all zero.

For a general nondefective matrix, we can also follow a similar scheme to obtain the eigenvectors of the matrix. First we transform the matrix into an upper Hessenberg matrix. A matrix with all the elements below (above) the first off-diagonal elements equal to zero is called an upper (lower) Hessenberg matrix. This transformation can always be achieved for a nondefective matrix **A** with

$$\mathbf{H} = \mathbf{U}^\dagger \mathbf{A} \mathbf{U} \tag{5.100}$$

under the unitary matrix **U**. Here **H** is in the form of a Hessenberg matrix. There are several methods that can be used to reduce a matrix to a Hessenberg matrix; they are given in Wilkinson (1965). A typical scheme is similar to the Householder scheme for reducing a symmetric matrix to a tridiagonal matrix. We can then solve the eigenvalue problem of the Hessenberg matrix. The eigenvalue problem of an upper (lower) Hessenberg matrix is considerably simpler than that of a general matrix under a typical algorithm, such as the QR algorithm. The so-called QR algorithm splits a nonsingular matrix **A** into a product of a unitary matrix **Q** and an upper-triangular matrix **R** as

$$\mathbf{A} = \mathbf{Q}\mathbf{R}. \tag{5.101}$$

We can construct a series of similarity transformations by alternating the order of **Q** and **R** to reduce the original matrix to an upper-triangular matrix that has the eigenvalues of the original matrix on the diagonal. Assume that $\mathbf{A}^{(0)} = \mathbf{A}$ and

$$\mathbf{A}^{(k)} = \mathbf{Q}_k \mathbf{R}_k; \tag{5.102}$$

then we have

$$\mathbf{A}^{(k+1)} = \mathbf{R}_k \mathbf{Q}_k = \mathbf{Q}_k^\dagger \mathbf{A}^{(k)} \mathbf{Q}_k = \mathbf{Q}_k^\dagger \cdots \mathbf{Q}_0^\dagger \mathbf{A} \mathbf{Q}_0 \cdots \mathbf{Q}_k, \tag{5.103}$$

with $k = 0, 1, \ldots$. This algorithm works best if **A** is a Hessenberg matrix. Taking account of stability and computing speed, a Householder type of scheme to transform a nondefective matrix into a Hessenberg matrix seems to be an excellent choice (Wilkinson, 1965).

When the matrix has been transformed into the Hessenberg form, we can use the inverse iteration method to obtain the eigenvectors of the Hessenberg matrix **H** with

$$\sqrt{\mathcal{N}_k}(\mathbf{H} - \mu \mathbf{I})\mathbf{y}^{(k)} = \mathbf{y}^{(k-1)} \tag{5.104}$$

and then the eigenvectors of the original matrix **A** from

$$\mathbf{x} = \mathbf{U}\mathbf{y}. \tag{5.105}$$

Of course, the normalization of the vector at each iteration is assumed with the normalization constant $\mathcal{N}_k$, with a definition similar to that of $\mathcal{N}_k$ in Eq. (5.86), to ensure the convergence.

Many numerical packages are available for dealing with linear algebra and matrix problems, including JMSL, a collection of mathematical, statistical, and charting classes in Java by Visual Numerics Inc. We should understand the basics and then learn how to apply the routines from numerical packages in research. A good survey of the existing packages can be found in Heath (2002).

5.6 The Faddeev–Leverrier method

A very interesting method developed for matrix inversion and eigenvalue problem is the *Faddeev–Leverrier method*. The scheme was first discovered by Leverrier in the middle of the nineteenth century and later modified by Faddeev (Faddeev and Faddeeva, 1963). Here we give a brief discussion of the method. The characteristic polynomial of the matrix is given by

$$p_n(\lambda) = |\mathbf{A} - \lambda\mathbf{I}| = \sum_{k=0}^{n} c_k\lambda^k, \tag{5.106}$$

where $c_n = (-1)^n$. Then we can introduce a set of supplementary matrices $\mathbf{S}_k$ with

$$p_n(\lambda)(\lambda\mathbf{I} - \mathbf{A})^{-1} = \sum_{k=0}^{n-1} \lambda^{n-k-1}\mathbf{S}_k. \tag{5.107}$$

If we multiply the above equation by $(\lambda\mathbf{I} - \mathbf{A})$, we have

$$\sum_{k=0}^{n} c_k\lambda^k\mathbf{I} = \mathbf{S}_0\lambda^n + \sum_{k=1}^{n-1}(\mathbf{S}_k - \mathbf{A}\mathbf{S}_{k-1})\lambda^{n-k} - \mathbf{A}\mathbf{S}_{n-1}. \tag{5.108}$$

Comparing the coefficients from the same order of λ^l for $l = 0, 1, \ldots, n$ on the both sides of the equation, we obtain the recursion for c_{n-k} and $\mathbf{S}_k$,

$$c_{n-k} = -\frac{1}{k}\,\mathrm{Tr}\,\mathbf{A}\mathbf{S}_{k-1}, \tag{5.109}$$

$$\mathbf{S}_k = \mathbf{A}\mathbf{S}_{k-1} + c_{n-k}\mathbf{I}, \tag{5.110}$$

for $k = 1, 2, \ldots, n$. The recursion is started with $\mathbf{S}_0 = \mathbf{I}$ and is ended at c_0. From Eq. (5.107) with $\lambda = 0$, we can easily show that

$$\mathbf{A}\mathbf{S}_{n-1} + c_0\mathbf{I} = \mathbf{0}, \tag{5.111}$$

which can be used to obtain the inverse

$$\mathbf{A}^{-1} = -\frac{1}{c_0}\mathbf{S}_{n-1}. \tag{5.112}$$

The following program is an example of generating $\mathbf{S}_k$ and c_k for a given matrix $\mathbf{A}$ with an evaluation of $\mathbf{A}^{-1}$ from $\mathbf{S}_{n-1}$ and c_0 as an illustration.

```java
// An example of using the Faddeev-Leverrier method to
// obtain the inversion of a matrix.

import java.lang.*;
public class Faddeev {
  public static void main(String argv[]) {
    double a[][]= {{ 1,  3,  2},
                   {-2,  3, -1},
                   {-3, -1,  2}};
    int n = a.length;
    double c[] = new double[n];
    double d[][] = new double[n][n];
    double s[][][] = (double[][][]) fl(a, c);
    for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
        d[i][j] = -s[n-1][i][j]/c[0];
    for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
        System.out.println(d[i][j]);
  }

// Method to complete the Faddeev-Leverrier recursion.

  public static double[][][] fl(double a[][],
    double c[]) {
    int n = c.length;
    double s[][][] = new double[n][n][n];
    for (int i=0; i<n; ++i) s[0][i][i] = 1;
    for (int k=1; k<n; ++k) {
      s[k] = mm(a,s[k-1]);
      c[n-k] = -tr(s[k])/k;
      for (int i=0; i<n; ++i)
      s[k][i][i] += c[n-k];
    }
      c[0] = -tr(mm(a,s[n-1]));
    return s;
  }

// Method to calculate the trace of a matrix.

  public static double tr(double a[][]) {
    int n = a.length;
    double sum = 0;
    for (int i=0; i<n; ++i) sum += a[i][i];
    return sum;
  }

// Method to evaluate the product of two matrices.

  public static double[][] mm(double a[][],
    double b[][]) {
    int n = a.length;
    double c[][] = new double[n][n];
    for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
        for (int k=0; k<n; ++k)
          c[i][j] += a[i][k]*b[k][j];
    return c;
  }
}
```

After running the above program, we obtain the inverse of the matrix. Note that each method in the example program above is kept general enough to deal with any real matrix.

Because the Faddeev–Leverrier recursion also generates all the coefficients c_k for the characteristic polynomial $p_n(\lambda)$, we can use a root search method to obtain all the eigenvalues from $p_n(\lambda) = 0$. This is the same as the situation for the Householder method discussed in the preceding section. We will further explore the zeros of a general real polynomial, including complex ones, in the next section.

After we have found all the eigenvalues λ_k, we can also obtain the corresponding eigenvectors with the availability of the supplementary matrices $\mathbf{S}_k$. If we define a new matrix

$$\mathbf{X}(\lambda_k) = \sum_{l=1}^{n} \mathbf{S}_{l-1}\lambda_k^{n-l}, \tag{5.113}$$

then the columns of $\mathbf{X}(\lambda_k)$ are the eigenvectors $\mathbf{x}_k$. This can be shown from the limit of the matrix $\mathbf{X}(\lambda)$ with $\lambda = \lambda_k + \delta$. Here $\delta \to 0$ is introduced to make $\mathbf{X}$ a nonsingular matrix. From Eq. (5.107), we can easily show that

$$\mathbf{X}(\lambda) = |\lambda\mathbf{I} - \mathbf{A}|(\lambda\mathbf{I} - \mathbf{A})^{-1}, \tag{5.114}$$

which leads each column of the matrix to the eigenvector $\mathbf{x}_k$ as $\delta \to 0$. The advantage of this scheme over the iterative scheme discussed in Section 5.5 is that we do not need to perform iterations as soon as we have the supplementary matrices $\mathbf{S}_k$. Note that we have not specified anything about the matrix $\mathbf{A}$ except that it is nonsingular. Therefore, the Faddeev–Leverrier method can be used for general matrices as well as symmetric matrices. However, the scheme can be time consuming if the dimension of the matrix is large.

5.7 Complex zeros of a polynomial

In scientific computing, we deal with problems mainly involving real symmetric or Hermitian matrices, which always have real eigenvalues. We rarely encounter a general complex matrix problem. If we do, we can still split the matrix into its real and imaginary parts, and then the problem becomes a general problem with two coupled real matrix equations. So occasionally we need to work with a general real matrix that may have complex eigenvalues and eigenvectors. If we can find the complex eigenvalues, we can also find the corresponding eigenvectors either by the inverse iteration method or the Faddeev–Leverrier method. Then how can we find all the eigenvalues of a general real matrix?

In principle, the eigenvalues of an $n \times n$ real matrix $\mathbf{A}$ are given by the zeros of the polynomial

$$p_n(\lambda) = |\mathbf{A} - \lambda\mathbf{I}| = \sum_{k=0}^{n} c_k\lambda^k, \tag{5.115}$$

The coefficients c_k are real, because $\mathbf{A}$ is real, with $c_n = (-1)^n$; the polynomial has a total of n zeros. We have discussed how to obtain all the coefficients c_k with the Faddeev–Leverrier method in the preceding section.

Here we explore some major properties of such a polynomial, especially its zeros, which can be complex even though all c_k are real. Note that if a complex value $z_0 = x_0 + iy_0$ for $y_0 \neq 0$ is a zero of the polynomial, its complex conjugate $z_0^* = x_0 - iy_0$ is also a zero. For a polynomial of degree n, that is, $c_n \neq 0$, there is a total of n zeros. This means that a polynomial with an odd n must have at least one real zero.

Locations of the zeros

In principle, we can rewrite the polynomial as a product

$$p_n(z) = \sum_{k=0}^{n} c_k z^k = c_n \prod_{k=1}^{n} (z - z_k), \tag{5.116}$$

where z_k is the kth zero of the polynomial, or the kth root of the equation $p_n(z) = 0$. If we take $z = 0$ in the above equation, we reach

$$\prod_{k=1}^{n} z_k = (-1)^n \frac{c_0}{c_n}. \tag{5.117}$$

We can force $|c_n| = |c_0| = 1$ by first dividing the polynomial by c_0 and then using z to denote $|c_n/c_0|^{1/n} z$. Then we have, from the above equation, some of the zeros residing inside the unit circle on the complex z plane, centered at the origin of $z = x + iy$. The rest of zeros are outside the unit circle. Thus we can use, for example, the bisection method to find the corresponding real and imaginary parts (or the amplitudes and phases) of the zeros inside the unit circle easily.

For the zeros outside the unit circle, we can further change the variable z to $1/z$ and then multiply the polynomial by z^n. The new polynomial has the coefficients c_k changed to c_{n-k}, for $k = 0, 1, \ldots, n$, and has zeros that are the inverses of the zeros of the original polynomial. Thus we can find the zeros of the original polynomial outside the unit circle by searching for the zeros of the new polynomial inside the unit circle.

The evaluation of a polynomial is necessary for any search scheme that looks for zeros within the unit circle. We can evaluate the polynomial efficiently by realizing that

$$\begin{aligned} p_n(z) &= u_n(x, y) + i v_n(x, y) \\ &= c_0 + z\{c_1 + z[c_2 + \cdots + (c_{n-1} + z c_n) \cdots]\}, \end{aligned} \tag{5.118}$$

where u_n and v_n are the real and imaginary parts of p_n, respectively. Then we can construct the recursion

$$u_k = c_{n-k} + x u_{k-1} - y v_{k-1}, \tag{5.119}$$

$$v_k = x v_{k-1} + y u_{k-1}, \tag{5.120}$$

starting with $u_0 = c_n$ and $v_0 = 0$. The following method implements such a recursive evaluation of p_n.

```
// Method to evaluate the polynomial with given c_k and
// z=x+iy.

  public static double[] polynomial2(double c[],
    double x, double y) {
    double p[] = new double[2];
    int n = c.length-1;
    double u = c[n];
    double v = 0;
    for (int i=0; i<n; ++i) {
      double t = x*v+y*u;
      u = c[n-i-1]+x*u-y*v;
      v = t;
    }
    p[0] = u;
    p[1] = v;
    return p;
  }
```

In order to find a zero of $p_n(z)$, we effectively have to solve two equations, $u_n(x, y) = 0$ and $v_n(x, y) = 0$, simultaneously. This can be done, for example, with the multivariable Newton method discussed in Section 5.4.

A related issue is to evaluate the coefficients of a polynomial with its imaginary axis shift by an amount x_0, that is,

$$q_n(z) = p_n(z - x_0) = \sum_{k=0}^{n} d_k z^k, \tag{5.121}$$

where the new coefficients d_n are given by

$$d_k = c_k - x_0 d_{k+1}, \tag{5.122}$$

for $k = n, n - 1, \ldots, 0$. Note that x_0 is real and we need $d_{n+1} = 0$ to start the recursion.

For a polynomial obtained from the secular equation of a real matrix, the zeros are in general bounded by the column or row modulus of the matrix on the z plane. For example, if we use the row modulus, we have

$$|z_k| \leq \|\mathbf{A}\|, \tag{5.123}$$

which provides a circle that circumscribes all the eigenvalues of the matrix. After we obtain the coefficients c_k, for example, with the Faddeev–Leverrier method, we can conduct an exhaustive search of all the eigenvalues within the circle of $|z| < \|\mathbf{A}\|$. This method of searching for all the eigenvalues is primitive and can be slow if n is large. If we want a more efficient scheme for searching for the zeros of the polynomial or the eigenvalues of the corresponding matrix, we must develop some new ideas.

Factoring a polynomial

A better strategy is to divide the original polynomial by a linear function or a quadratic function, and then search for the zeros of the polynomial by turning the coefficients of the remainders to zero. This is more efficient because we can obtain the zero (or a pair of zeros) and reduce the polynomial to its quotient for further search at the same time.

For example, if we divide the polynomial $p_n(z)$ by a divisor that is a first-order polynomial $f_1(z) = \alpha + \beta z$, we have the quotient

$$p_{n-1}(z) = p_n(x)/f_1(x) - r_0 = \sum_{k=0}^{n-1} d_k z^k, \tag{5.124}$$

where d_k is given by

$$d_{k-1} = (c_k + \alpha d_k)/\beta, \tag{5.125}$$

for $k = n, n-1, \ldots, 1$. Note that we take $d_n = 0$ to start the recursion, and the remainder is given by $r_0 = c_0 - \alpha d_0$. We can easily implement this factoring scheme to locate real zeros. Note that we can use the sign of the remainder to narrow down the region. Consider two points x_1 and $x_2 > x_1$ on the real axis and use $f_1(z) = z - x_1$ to obtain the remainder $r_0(x_1)$ and $f_1(z) = z - x_2$ to obtain the remainder $r_0(x_2)$. If we have $r_0(x_1)r_0(x_2) < 0$, we know that there is at least one real zero $z_0 = x_0$ in the region $[x_1, x_2]$. A combination of the bisection method and the above factoring scheme can locate a real zero of $p_n(z)$ quickly. This process can be continued with a search for a zero in the quotient, the quotient of the quotient, ... , until we have exhausted all the real zeros of the polynomial $p_n(z)$.

To divide the polynomial $p_n(z)$ by $f_2(z) = \alpha + \beta z + \gamma z^2$, we have the quotient

$$p_{n-2}(z) = p_n(x)/f_2(x) - (r_0 + r_1 z) = \sum_{k=0}^{n-2} e_k z^k, \tag{5.126}$$

where e_k is given by

$$e_{k-2} = (c_k - \alpha e_k - \beta e_{k-1})/\gamma, \tag{5.127}$$

for $k = n, n-1, \ldots, 1$. Note that we take $e_n = e_{n-1} = 0$ to start the recursion, and the remainder coefficients are given by $r_1 = c_1 - \alpha e_1 - \beta e_0$ and $r_0 = c_0 - \alpha e_0$.

The above factoring scheme can be applied in the search for a pair of complex zeros in the polynomial $p_n(z)$ by simultaneously solving $r_0(\alpha, \beta) = 0$ and $r_1(\alpha, \beta) = 0$ if we set $\gamma = 1$ in $f_2(z)$. One can use the discrete Newton method introduced in Section 5.4 to accomplish the root search. After we locate a root with parameters $\alpha = \alpha_0$ and $\beta = \beta_0$, we can relate them back to the pair of zeros of $p_n(z)$ as $\big(-\beta_0 \pm i\sqrt{4\alpha_0 - \beta_0^2}\big)/2$.

We can also work out the analytical expressions for the partial derivatives $\partial r_i/\partial\alpha$ and $\partial r_i/\partial\beta$ for $i = 0, 1$, and then apply the Newton method to solve the two equations efficiently. This scheme is called the *Bairstow method*. For a detailed discussion of the Bairstow method, see Wilkinson (1965).

The Routh–Hurwitz test

An interesting scheme for testing the real parts of the zeros of a polynomial $p_n(z)$ is called the *Routh–Hurwitz test*. The scheme uses the coefficients c_k to construct a sequence whose signs determine how many zeros are on the right-hand side of the imaginary axis. Here is how to construct the sequence.

We can build an $(n + 1) \times m$ matrix with

$$B_{ij} = \frac{1}{B_{i+11}} \begin{vmatrix} B_{i+11} & B_{i+1j+1} \\ B_{i+21} & B_{i+2j+1} \end{vmatrix}, \tag{5.128}$$

for $i = n - 1, n - 2, \ldots, 1$ and $j = 1, 2, \ldots, m$, where

$$B_{n+1j} = c_{n-2(j-1)}, \tag{5.129}$$

$$B_{nj} = c_{n-2j+1}, \tag{5.130}$$

with $m = 1 + n/2$ if n is even and $m = (n + 1)/2$ if n is odd. Note that if c_{-1} or any B_{ij} outside the range of the matrix shows up, it is treated as 0. Then we count how many sign agreements that we have between B_{i1} and B_{i+11} for $i = 1, 2, \ldots, n$, which is the number of zeros on the right-hand side of the imaginary axis.

There are a couple of problems in the above recursion. First, if $B_{i+11} = 0$, the recursion cannot go on. We can cure this by multiplying the polynomial by $z - x_0$ with $x_0 < 0$. This does not change the zeros in the original polynomial but merely adds another real zero to the polynomial on the left-hand side of the imaginary axis.

The second problem is when a row of the matrix turns out to be zero. This terminates the recursion prematurely. This problem arises from the elements from the previous row (one line lower on the matrix); they happen to be the coefficients of an exact divisor with a zero remainder. We can cure this by replacing the zeros with the coefficients from the first-order derivative of the exact divisor. The highest order of the exact divisor is determined by the index of the row that is zero. For example, if $B_{kj} = 0$ for all j, the divisor is $d(x) = B_{k+11}x^k + B_{k+12}x^{k-2} + B_{k+13}x^{k-4} + \cdots$.

The Routh–Hurwitz test provides an efficient and powerful method of searching for zeros and of factoring the polynomial if we combine it with the shift operation discussed earlier and the bisection search along the imaginary axis.

5.8 Electronic structures of atoms

As we have pointed out, there are many applications of matrix operations in physics. We have given a few examples in Section 5.1 and a concrete case study on multicharge clusters in Section 5.4. In this section, we demonstrate how to apply the matrix eigenvalue schemes in the calculation of the electronic structures of many-electron atomic systems within the framework of the Hartree–Fock approximation.

The Schrödinger equation for a multielectron atom is given by

$$\mathcal{H}\Psi_k(\mathbf{R}) = \mathcal{E}_k\Psi_k(\mathbf{R}) \tag{5.131}$$

for $k = 0, 1, \ldots,$ where $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ is a $3N$-dimensional position vector for all N electrons in the system, $\mathcal{H}$ is the Hamiltonian of the N electrons in the system, given by

$$\mathcal{H} = -\frac{\hbar^2}{2m_\mathrm{e}}\sum_{i=1}^{N}\nabla_i^2 - \frac{Ze^2}{4\pi\epsilon_0}\sum_{i=1}^{N}\frac{1}{r_i} + \frac{e^2}{4\pi\epsilon_0}\sum_{i>j}^{N}\frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}, \tag{5.132}$$

and $\mathcal{E}_k$ is the kth eigenvalue of the the Hamiltonian. Here Z is the atomic number, ϵ_0 is the electric permittivity of vacuum, m_e is the mass of a free electron, and e is the fundamental charge. In general, it is not possible to obtain an exact solution of the multielectron Schrödinger equation; approximations must be made in order to solve the underlying eigenvalue problem. To simplify the expressions, we use atomic units, that is, $m_\mathrm{e} = e = 4\pi\epsilon_0 = \mu_0/4\pi = \hbar = 1$, where μ_0 is the magnetic permeability of vacuum. Under these choices of units, lengths are given in the Bohr radius, $a_0 = 4\pi\epsilon_0\hbar^2/m_\mathrm{e}e^2 = 0.529\,177\,249(24)\times 10^{-10}$ m, and energies are given in the hartree, $e^2/4\pi\epsilon_0 a_0 = 27.211\,396\,1(81)$ eV.

The Hartree–Fock approximation assumes that the ground state of a system of fermions (electrons in this case) can be viewed as occupying the lowest set of certain single-particle states after considering the Pauli exclusion principle. Then the ground state is approximated by the *Hartree–Fock ansatz*, which can be cast into a determinant

$$\Psi_\mathrm{HF}(\mathbf{R}) = \frac{1}{\sqrt{N!}}\begin{vmatrix} \phi_1(\mathbf{r}_1) & \phi_1(\mathbf{r}_2) & \cdots & \phi_1(\mathbf{r}_N) \\ \phi_2(\mathbf{r}_1) & \phi_2(\mathbf{r}_2) & \cdots & \phi_2(\mathbf{r}_N) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_N(\mathbf{r}_1) & \phi_N(\mathbf{r}_2) & \cdots & \phi_N(\mathbf{r}_N) \end{vmatrix}; \tag{5.133}$$

this is also known as the *Slater determinant*. The index used above includes both the spatial and spin indices. In most cases, it is more convenient to write the single particle states as $\phi_{i\sigma}(\mathbf{r})$ with the spin index σ ($\uparrow$ or $\downarrow$) separated from the spatial index i.

Because $\mathcal{E}_0$ is the ground-state energy of the system, we must have

$$\mathcal{E}_\mathrm{HF} = \frac{\langle\Psi_\mathrm{HF}|\mathcal{H}|\Psi_\mathrm{HF}\rangle}{\langle\Psi_\mathrm{HF}|\Psi_\mathrm{HF}\rangle} \geq \mathcal{E}_0. \tag{5.134}$$

To optimize (minimize) $\mathcal{E}_\mathrm{HF}$, we perform the functional variation with respect to $\phi_{i\sigma}^\dagger(\mathbf{r})$; then we have

$$\begin{aligned}&\left[-\frac{1}{2}\nabla^2 - \frac{Z}{r} + V_\mathrm{H}(\mathbf{r})\right]\phi_{i\sigma}(\mathbf{r}) - \int V_{\mathrm{x}\sigma}(\mathbf{r}', \mathbf{r})\phi_{i\sigma}(\mathbf{r}')\,d\mathbf{r}' \\ &\quad = \varepsilon_i\phi_{i\sigma}(\mathbf{r}),\end{aligned} \tag{5.135}$$

which is known as the Hartree–Fock equation (see Exercise 5.15). Here $V_\mathrm{H}(\mathbf{r})$ is the Hartree potential given by

$$V_\mathrm{H}(\mathbf{r}) = \int\frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|}\,d\mathbf{r}', \tag{5.136}$$

where $\rho(\mathbf{r}) = \rho_\uparrow(\mathbf{r}) + \rho_\downarrow(\mathbf{r})$ is the total density of the electrons at $\mathbf{r}$. The exchange interaction $V_{x\sigma}(\mathbf{r}, \mathbf{r}')$ is given by

$$V_{x\sigma}(\mathbf{r}', \mathbf{r}) = \frac{\rho_\sigma(\mathbf{r}, \mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|}, \tag{5.137}$$

where $\rho_\sigma(\mathbf{r}, \mathbf{r}')$ is the density matrix of the electrons, given by

$$\rho_\sigma(\mathbf{r}, \mathbf{r}') = \sum_{i=1}^{N_\sigma} \phi_{i\sigma}^\dagger(\mathbf{r})\phi_{i\sigma}(\mathbf{r}'). \tag{5.138}$$

Here N_σ is the total number of states occupied by the electrons with spin σ. The density of the electrons is the diagonal of the density matrix, that is, $\rho_\sigma(\mathbf{r}) = \rho_\sigma(\mathbf{r}, \mathbf{r})$. The eigenvalue ε_i above is a multiplier introduced during the optimization (minimization) of $\mathcal{E}_{\mathrm{HF}}$.

The Hartree potential can also be obtained from the solution of the Poisson equation

$$\nabla^2 V_{\mathrm{H}}(\mathbf{r}) = -4\pi\rho(\mathbf{r}). \tag{5.139}$$

The single-particle wavefunctions in the atomic systems can be assumed to have the form

$$\phi_{i\sigma}(\mathbf{r}) = \frac{1}{r} u_{nl\sigma}(r) Y_{lm}(\theta, \phi), \tag{5.140}$$

where $Y_{lm}(\theta, \phi)$ are the spherical harmonics with θ and ϕ being the polar and azimuthal angles, respectively. Quantum numbers n, l, and m correspond to the energy, angular momentum, and z component of the angular momentum.

We can make the further assumption that the electron density is spherically symmetric and divide the space $[0, r_c]$ into discrete points with an evenly spaced interval h. Here r_c is the cut-off in the radial direction, typically a few Bohr radii. The spherical approximation of the density of electrons is valid only if the formation of the magnetic moment is not considered. If we use a numerical expression for the second-order derivative, for example, the three-point formula, the Hartree–Fock equation for a given l is converted into a matrix equation

$$\mathbf{Hu} = \varepsilon\mathbf{u}, \tag{5.141}$$

with the diagonal elements of $\mathbf{H}$ given by

$$H_{ii} = \frac{1}{h^2} + \frac{l(l+1)}{2r_i^2} - \frac{Ze^2}{r_i} + V_{\mathrm{H}}(r_i), \tag{5.142}$$

and the corresponding off-diagonal elements given by

$$H_{ij} = -\frac{\delta_{ij\pm 1}}{2h^2} + hV_{x\sigma}^{(l)}(r_i, r_j), \tag{5.143}$$

where $V_{x\sigma}^{(l)}(r_i, r_j)$ is the lth component of $V_{x\sigma}(\mathbf{r}_i, \mathbf{r}_j)$ expanded in spherical harmonics. We have used $\mathbf{H}$ for the matrix form of $\mathcal{H}$ and $\mathbf{u}$ for the discrete form of the wavefunction $u_{nl\sigma}(r)$. The angular momentum index is suppressed in **H** and **u** for convenience. We can easily apply the numerical schemes introduced in Section 5.5 to solve this matrix eigenvalue problem. The energy levels for different n are obtained for a fixed l. We can, of course, evaluate the density matrix and the Hartree–Fock ground-state energy with the method described here.

5.9 The Lanczos algorithm and the many-body problem

One of the most powerful methods in large-scale matrix computing is the *Lanczos method*, which is an iterative scheme suitable for large, especially sparse (most elements are zero) matrices. The advantage of the Lanczos method is extremely noticeable when we only need a few eigenvalues and eigenvectors, or when the system is extremely large and sparse. Here we just sketch a very basic Lanczos algorithm. More elaborate discussions on various Lanczos methods can be found in several specialized books, for example, Wilkinson (1965), Cullum and Willoughby (1985), and Hackbusch (1994).

Assuming that the matrix **H** is an $n \times n$ real symmetric matrix, we can tridiagonalize an $m \times m$ subset of **H** with

$$\mathbf{O}^{\mathrm{T}}\mathbf{H}\mathbf{O} = \tilde{\mathbf{H}}, \tag{5.144}$$

where **O** is an $n \times m$ matrix with its kth column given by

$$\mathbf{v}_k = \frac{\mathbf{u}_k}{\sqrt{\mathcal{N}_k}}, \tag{5.145}$$

for $k = 1, 2, \ldots, m$, where $\mathcal{N}_k = \mathbf{u}_k^{\mathrm{T}}\mathbf{u}_k$ is the normalization constant and the vectors $\mathbf{u}_k$ are generated recursively from an arbitrary vector $\mathbf{u}_1$ as

$$\mathbf{u}_{k+1} = \mathbf{H}\mathbf{v}_k - \alpha_k \mathbf{v}_k - \beta_k \mathbf{v}_{k-1}, \tag{5.146}$$

with $\beta_k = \tilde{H}_{k-1k} = \mathbf{v}_{k-1}^{\mathrm{T}}\mathbf{H}\mathbf{v}_k$ and $\alpha_k = \tilde{H}_{kk} = \mathbf{v}_k^{\mathrm{T}}\mathbf{H}\mathbf{v}_k$. The recursion is started at $\beta_1 = 0$ with $\mathbf{u}_1$ being a selected vector. Note that $\mathbf{u}_1$ can be a normalized vector with each element generated from a uniform random-number generator, for example. In principle, the vectors $\mathbf{v}_k$, for $k = 1, 2, \ldots, m$, form an orthonormal set, but in practice we still need to carry out the Gram–Schmidt orthogonalization procedure, at every step of the recursion, to remove the effects of rounding errors. We can show that the eigenvalues of the tridiagonal submatrix $\tilde{\mathbf{H}}$ are the approximations of the ones of **H** with the largest magnitudes. We can use the standard methods discussed earlier to diagonalize $\tilde{\mathbf{H}}$ to obtain its eigenvectors and eigenvalues from $\tilde{\mathbf{H}}\tilde{\mathbf{x}}_k = \lambda_k \tilde{\mathbf{x}}_k$. The eigenvectors $\tilde{\mathbf{x}}_k$ can be used to obtain the approximate eigenvectors of **H** with $\mathbf{x}_k \simeq \mathbf{O}\tilde{\mathbf{x}}_k$.

The approximation is improved if we construct a new initial state $\mathbf{u}_1$ from the eigenvectors $\tilde{\mathbf{x}}_k$ with $k = 1, 2, \ldots, m$, for example,

$$\mathbf{u}_1 = \sum_{k=1}^{m} c_k \tilde{\mathbf{x}}_k, \tag{5.147}$$

and then the recursion is repeated again and again. We can show that this iterative scheme will eventually lead to the m eigenvectors of $\mathbf{H}$, corresponding to the eigenvalues with the largest magnitudes. In practice, the selection of the coefficients c_k is rather important in order to have a fast and accurate algorithm. Later in this section we will introduce one of the choices made by Dagotto and Moreo (1985) in the study of the ground state of a quantum many-body system.

We can work out the eigenvalue problem for a specified region of the spectrum of $\mathbf{H}$ with the introduction of the matrix

$$\mathbf{G} = (\mathbf{H} - \mu\mathbf{I})^{-1}. \tag{5.148}$$

We can solve $\mathbf{G}$ with the Lanczos algorithm to obtain the eigenvectors with eigenvalues near μ. Note that

$$\mathbf{G}\mathbf{x}_k = \frac{1}{\lambda_k - \mu}\mathbf{x}_k \tag{5.149}$$

if $\mathbf{H}\mathbf{x}_k = \lambda_k\mathbf{x}_k$. This is useful if one wants to know about the spectrum of a particular region.

At the beginning of this chapter, we used a many-body Hamiltonian (the Hubbard model) in Eq. (5.17) to describe the electronic behavior of H_3^+. It is generally believed that the Hubbard model and its variants can describe the majority of highly correlated quantum systems, for example, transition metals, rare earth compounds, conducting polymers, and oxide superconducting materials. There are good reviews of the Hubbard model in Rasetti (1991). Usually, we want to know the properties of the ground state and the low-lying excited states. For example, if we want to know the ground state and the first excited state of a cluster of N sites with $N_0 < N$ electrons, we can solve the problem using the Lanczos method by taking $m \simeq 10$ and iterating the result a few times. Note that the number of many-body states increases exponentially with both N_0 and N. The iteration converges to the ground state and the low-lying excited state only if they have the largest eigenvalue magnitudes. The ground state and low-lying excited state energies carry the largest magnitudes if the chemical potential of the system is set to be zero. We also have to come up with a construction for the next guess of $\mathbf{u}_1$. We can, for example, use

$$\mathbf{u}_1^{(l+1)} = \sum_{k=1}^{5} \mathbf{v}_k^{(l)}. \tag{5.150}$$

A special choice of the iteration scheme for the ground-state properties is given by Dagotto and Moreo (1985). The $(l+1)$th iteration of $\mathbf{v}_1$ is taken as

$$\mathbf{v}_1^{(l+1)} = \frac{1}{\sqrt{1+a^2}}\left(\mathbf{v}_1^{(l)} + a\mathbf{v}_2^{(l)}\right), \tag{5.151}$$

where a is determined from the minimization of the expectation value of $\mathbf{H}$ under $\mathbf{v}_1^{(l+1)}$, which gives

$$a = b - \sqrt{1+b^2}, \tag{5.152}$$

where b is expressed in terms of the expectation values of the lth iteration as

$$b = \frac{d_3 - 3d_1d_2 + 2d_1^3}{2(d_2 - d_1^2)^{3/2}} \tag{5.153}$$

with $d_1 = \mathbf{v}_1^T\mathbf{H}\mathbf{v}_1$, $d_2 = \mathbf{v}_1^T\mathbf{H}^2\mathbf{v}_1$, and $d_3 = \mathbf{v}_1^T\mathbf{H}^3\mathbf{v}_1$. The second vector

$$\mathbf{v}_2 = \frac{1}{\sqrt{d_2 - d_1^2}} (\mathbf{H}\mathbf{v}_1 - d_1\mathbf{v}_1) \tag{5.154}$$

is also normalized under such a choice of $\mathbf{v}_1$. The advantage of this algorithm is that we only need to store three vectors $\mathbf{v}_1$, $\mathbf{H}\mathbf{v}_1$, and $\mathbf{H}^2\mathbf{v}_1$ during each iteration. The eigenvalue with the largest magnitude is also given iteratively from

$$\lambda_1 = \mathbf{v}_1^T\mathbf{H}\mathbf{v}_1 + \frac{a}{\sqrt{d_2 - d_1^2}}. \tag{5.155}$$

This algorithm is very efficient for calculating the ground-state properties of many-body quantum systems. For more discussions on the method and its applications, see Dagotto (1994).

5.10 Random matrices

The distributions of the energy levels of many physical systems have some universal features determined by the fundamental symmetry of the Hamiltonian. This type of feature is usually qualitative. For example, when disorders are introduced into metallic systems, the resistivities increase and the systems become insulators if the disorders are strong enough. For each metal, the degree of disorder needed to become an insulator is different, but the general behavior of metallic systems to become insulators under strong disorders is the same. If we want to represent a disordered material with a matrix Hamiltonian, the elements of the matrix have to be randomly selected. The general feature of a physical system is obtained with an ensemble of random matrices satisfying the physical constraints of the system.

Even though the elements of the matrices are random, at least three types of fundamental symmetries can exist in the ensembles of the random matrices for a given physical system. For example, if the system has time-reversal symmetry plus rotational invariance for the odd-half-integer spin case, the ensemble is real symmetric, or *orthogonal*.

If rotational invariance is not present in the odd-half-integer spin case, the ensemble is quaternion real, or *symplectic*. The general case without time-reversal symmetry is described by general Hermitian matrices, or a *unitary* ensemble.

The structure of the matrix is the other relevant factor that determines the detailed properties of a specific system. Traditionally, the focus of the general properties was on the orthogonal ensemble with real symmetric matrices and a Gaussian distribution for the matrix elements. Efforts made in the last 30 years are described in Mehta (1991). In this section, we will not be able to cover many aspects of random matrices and will only provide a very brief introduction. Interested readers should consult Mehta (1991), Brody *et al.* (1981), and Bohigas (1991).

We can easily generate a random matrix if the symmetry and the structure of the matrix are specified. The Gaussian orthogonal ensemble of $n \times n$ matrices is specified with a distribution $\mathcal{W}_n(\mathbf{H})$ that is invariant under an orthogonal transformation with

$$\mathcal{W}_n(\mathbf{H}')\,d\mathbf{H}' = \mathcal{W}_n(\mathbf{H})\,d\mathbf{H}, \tag{5.156}$$

where

$$\mathbf{H}' = \mathbf{O}^{\mathrm{T}}\mathbf{H}\mathbf{O}, \tag{5.157}$$

with $\mathbf{O}$ being an orthogonal matrix. The above condition also implies that $\mathcal{W}_n(\mathbf{H})$ is invariant under the orthogonal transformation because $d\mathbf{H}$ is invariant. This restricts the distribution to be

$$\mathcal{W}_n(\mathbf{H}) = e^{-\mathrm{Tr}\,\mathbf{H}^2/4\sigma^2}, \tag{5.158}$$

where the trace is given by

$$\mathrm{Tr}\,\mathbf{H}^2 = \sum_{i=1}^{n} H_{ii}^2 + \sum_{i\neq j}^{n} H_{ij}^2. \tag{5.159}$$

The average of the elements and the variance satisfy $\langle H_{ij}\rangle = 0$ and $\langle H_{ij}^2\rangle = (1 + \delta_{ij})\sigma^2$ for the Gaussian orthogonal ensemble. The variance for the off-diagonal elements is only half that of for the diagonal elements because $H_{ij} = H_{ji}$ in symmetric matrices.

We can generate a random-matrix ensemble numerically and diagonalize each matrix to obtain the distribution of the eigenvalues and eigenvectors. For example, the Gaussian orthogonal random matrix can be obtained with the method given below. The Gaussian random-number generator used is from the Java language.

```
// Method to generate a random matrix for the Gaussian
// orthogonal ensemble with sigma being the standard
// deviation of the off-diagonal elements.

import java.util.Random;
  public static double[][] rm(int n, double sigma) {
    double a[][] = new double[n][n];
    double sigmad = Math.sqrt(2)*sigma;
    Random r = new Random();

    for (int i=0; i<n; ++i)
      a[i][i] = sigmad*r.nextGaussian();

    for (int i=0; i<n; ++i) {
      for (int j=0; j<i; ++j) {
        a[i][j] = sigma*r.nextGaussian();
        a[j][i] = a[i][j];
      }
    }
    return a;
  }
```

This matrix can then be diagonalized by any method discussed earlier in this chapter. In order to obtain the statistical information, the matrix needs to be generated and diagonalized many times before a smooth and correct distribution can be reached. Based on the distributions of the eigenvalues and the eigenvectors of the random-matrix ensemble, we can obtain the correlation functions of the eigenvalues and other important statistical information. Statistical methods and correlation functions are discussed in more detail in Chapters 8 and 10. We can also use the Gaussian random-number generator introduced in Chapter 2 instead the one from Java in the above method.

The distribution density of the eigenvalues in the Gaussian orthogonal ensemble can be obtained analytically, and it is given by a semicircle function

$$\rho(\lambda) = \begin{cases} \dfrac{1}{2\pi}\sqrt{4-\lambda^2} & \text{if } |\lambda| < 2, \\ 0 & \text{elsewhere,} \end{cases} \tag{5.160}$$

which was first derived by Wigner in the 1950s. Here $\rho(\lambda)$ is the normalized density at the eigenvalue λ measured in $\sigma\sqrt{n}$. The numerical simulations carried out so far seem to suggest that the semicircle distribution is true for any other ensemble as long as the elements satisfy $\langle H_{ij}\rangle = 0$ and $\langle H_{ij}^2\rangle = \sigma^2$ for $i < j$. We will leave this as an exercise for the reader.

There has been a lot of activity in the field of random-matrix theory and its applications. Here we mention just a couple of examples. Hackenbroich and Weidenmüller (1995) have shown that a general distribution of the form

$$\mathcal{W}_n(\mathbf{H}) = \frac{1}{\mathcal{Z}} e^{-n\,\mathrm{Tr}\,\mathbf{V}(\mathbf{H})}, \tag{5.161}$$

where $\mathcal{Z}$ is a normalization constant, would lead to the same correlation functions among the eigenvalues as those of a Gaussian orthogonal ensemble, in the limit of $n \to \infty$. This is true for any ensemble, orthogonal, symplectic, or unitary. Here $\mathbf{V}(\mathbf{H})$ is a function of $\mathbf{H}$ with the restriction that its eigenvalues are confined within a finite interval with a smooth distribution and that $V(\lambda)$ grows at least linearly as $\lambda \to \infty$. The Gaussian ensemble is a special case with $\mathbf{V}(\mathbf{H}) \propto \mathbf{H}^2$.

Akulin, Bréchignac, and Sarfati (1995) have used the random-matrix method in the study of the electronic, structural, and thermal properties of metallic clusters. They have introduced a random interaction V, where $\langle V\rangle = 0$ and $\langle V^2\rangle$ is a function characterized by the electron–electron, electron–ion, and/or electron–phonon interactions, and the shape fluctuations of the clusters. Using this theory, they have predicted deformation transformation in the cluster when the temperature is lowered. This effect is predicted only for open shell clusters and is quite similar to the Jahn–Teller effect, which creates a finite distortion in the lattice in ionic solids due to the electron–phonon interaction.

Exercises

5.1 Find the currents in the unbalanced Wheatstone bridge (Fig. 5.1). Assume that $v_0 = 1.5$ V, $r_1 = r_2 = 100\ \Omega$, $r_3 = 150\ \Omega$, $r_x = 120\ \Omega$, $r_a = 1000\ \Omega$, and $r_s = 10\ \Omega$.

5.2 A typical problem in physics is that physical quantities can be calculated for a series of finite systems, but ultimately, we would like to know the results for the infinite system. Hulthén (1938) studied the one-dimensional spin-$\frac{1}{2}$ Heisenberg model with the Hamiltonian

$$\mathcal{H} = \sum_{i=1}^{n-1} \mathbf{s}_i \cdot \mathbf{s}_{i+1},$$

where $\mathbf{s}_i$ is the spin at the ith site and n is the total number of sites in the system. From the eigenequation

$$\mathcal{H}\Psi_k = \mathcal{E}_k \Psi_k,$$

Hulthén obtained the ground-state energy per site, $\varepsilon_n = \mathcal{E}_0/n$, for a series of finite systems with $\varepsilon_2 = -2.0000$, $\varepsilon_4 = -1.5000$, $\varepsilon_6 = -1.4343$, $\varepsilon_8 = -1.4128$, and $\varepsilon_{10} = -1.4031$. Now assume that the ground-state energy per site is given by

$$\varepsilon_n = \varepsilon_\infty + \frac{c_1}{n} + \frac{c_2}{n^2} + \cdots + \frac{c_l}{n^l} + \cdots .$$

Truncate the above series at $l = 4$ and find ε_∞ by solving the linear equation set numerically.

5.3 Consider the least-squares approximation of a discrete function $f(x_i)$ for $i = 0, 1, \ldots, n$ with the polynomial

$$p_m(x) = \sum_{k=0}^{m} c_k x^k.$$

Write a subprogram that evaluates all the c_k from

$$\frac{\partial \chi^2}{\partial c_l} = 0,$$

for $l = 0, 1, \ldots, m$, where

$$\chi^2 = \sum_{i=0}^{n} [p_m(x_i) - f(x_i)]^2.$$

5.4 Develop a subprogram to achieve the LU decomposition of an $n \times n$ banded matrix with l subdiagonals and m superdiagonals with either the Crout or the Doolittle factorization. Simplify the subprogram for the cases of $l = m$ and a symmetric matrix.

5.5 Apply the secant method to obtain the stable geometric structure of clusters of ions $(\mathrm{Na}^+)_n(\mathrm{Cl}^-)_m$, where n and m are small positive integers. Use the empirical interaction potential given in Eq. (5.64) for the ions.

5.6 Write a subprogram to implement the BFGS optimization scheme. Test it by searching for the stable geometric structure of $(\mathrm{NaCl})_5$. Use the empirical interaction potential given in Eq. (5.64) for the ions.

5.7 Write a subprogram that utilizes the Householder scheme to tridiagonalize a real symmetric matrix.

5.8 Write a subprogram that uses the properties of determinant polynomials of a tridiagonal matrix and a root-search method to solve its first few eigenvalues.

5.9 Discretize the one-dimensional Schrödinger equation

$$-\frac{\hbar^2}{2m}\frac{d^2\phi(x)}{dx^2} + V(x)\phi(x) = \varepsilon\phi(x)$$

by applying the three-point formula to the second-order derivative above. Assuming that the potential $V(x)$ is

$$V(x) = \frac{\hbar^2}{2m}\alpha^2\lambda(\lambda - 1)\left[\frac{1}{2} - \frac{1}{\cosh^2(\alpha x)}\right],$$

solve the corresponding eigenvalue problem by the inverse iteration method to obtain the eigenvalues and eigenvectors for the four lowest states. Compare the numerical result obtained here with the analytical result given in Eq. (4.98).

5.10 Find all the 15×15 elements of the Hamiltonian for H_3^+ and solve the corresponding eigenvalues and eigenvectors numerically. Compare the numerical results with the analytic results by reducing the matrix to block-diagonal form with the largest block being a 2×2 matrix.

5.11 It is of special interest in far infrared spectroscopy to know the vibrational spectrum of a molecule. Find all the the vibrational modes of Na_2Cl_2. Use the empirical interaction potential for the ions that is given in Eq. (5.64).

5.12 Implement the Faddeev–Leverrier method in a program to obtain the inverse, eigenvalues, and eigenvectors of a general real matrix.

5.13 Divide the polynomial $p_n(z) = \sum_{k=0}^{n} c_k z^k$ by $f_2(z) = z^2 + \beta z + \alpha$ twice and obtain the recursions for the coefficients in the quotients and remainders in each step. Find analytical expressions for $\partial r_i(\alpha, \beta)/\partial\alpha$ and $\partial r_i(\alpha, \beta)/\partial\beta$, where r_0 and r_1 are the coefficients of the remainder $r = r_1 z + r_0$ after the first division. Derive the Bairstow method that utilizes the Newton method to factor the polynomial $p_n(z)$ and to obtain a pair of its zeros. Implement the scheme in a program and test it with $p_6(z) = (z^2 + z + 1)(z^2 + 2z + 2)(z^2 + 3z + 3)$.

5.14 Combine the Routh–Hurwitz test and bisection method in a program to locate all the zeros of the polynomial $p_n(z) = \sum_{k=0} c_k z^k$. Test it with $p_6(z) = (z^2 + z + 1)(z^2 + 2z - 4)(z^2 + 3z + 3)$.

5.15 Derive the Hartree–Fock equation for the atomic systems given in Section 5.8. Show that the matrix form of the Hartree–Fock equation does represent the original equation if the electron density is spherically symmetric, and find the expression for $V_{x\sigma}^{(l)}(r_i, r_j)$ in the matrix equation.

5.16 Write a program to generate and diagonalize ensembles of real symmetric matrices with the following distributions:

$$\mathcal{W}_n(H_{ij}) = \begin{cases} \frac{1}{2} & \text{if } |H_{ij}| < 1, \\ 0 & \text{otherwise,} \end{cases}$$
$$\mathcal{W}_n(H_{ij}) = \frac{1}{2}[\delta(H_{ij} - 1) + \delta(H_{ij} + 1)],$$
$$\mathcal{W}_n(H_{ij}) = \frac{1}{\sqrt{2\pi}} e^{-H_{ij}^2/2},$$

for $i \leq j$. Compare the density of eigenvalues with that of the Wigner semicircle.

5.17 Find the optimized configurations of $N < 20$ charges confined on the surface of a unit sphere. Discuss whether the configurations found are the global minima of the electrostatic potential energies of the systems.

5.18 Find the optimized configurations of $N < 20$ particles, such as argon atoms, interacting with each other through the Lennard–Jones potential

$$V_{ij} = 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right],$$

where both ε and σ are model parameters. Find the N dependence of the number of local minima that are close to the global minimum of the total interaction energy of the system.

Chapter 6
Spectral analysis

Nature often behaves differently to how our intuition predicts it should. For example, when we observe a periodic motion, immediately we can figure out the period involved, but not the detailed structure of the data within each period. It was Fourier who first pointed out that an arbitrary periodic function $f(t)$, with a period T, can be decomposed into a summation of simple harmonic terms, which are periodic functions of frequencies that are multiples of the fundamental frequency $1/T$ of the function $f(t)$. Each coefficient in the summation is given by an integral of the product of the function and the complex conjugate of that harmonic term.

Assuming that we have a time-dependent function $f(t)$ with a period T, that is, $f(t+T) = f(t)$, the Fourier theorem can be cast as a summation

$$f(t) = \sum_{j=-\infty}^{\infty} g_j e^{-ij\omega t}, \tag{6.1}$$

which is commonly known as the Fourier series. Here $\omega = 2\pi/T$ is the fundamental angular frequency and g_j are the Fourier coefficients, which are given by

$$g_j = \frac{1}{T}\int_0^T f(t) e^{ij\omega t}\, dt. \tag{6.2}$$

The Fourier theorem can be derived from the properties of the exponential functions

$$\phi_j(t) = \frac{1}{\sqrt{T}} e^{-ij\omega t}, \tag{6.3}$$

which form an orthonormal basis set in the region of one period of $f(t)$: namely,

$$\int_{t_0}^{t_0+T} \phi_j^*(t)\phi_k(t)\, dt = \langle j|k\rangle = \delta_{jk}, \tag{6.4}$$

where t_0 is an arbitrary starting point, $\phi_j^*(t)$ is the complex conjugate of $\phi_j(t)$, and δ_{jk} is the Kronecker δ function.

The Fourier coefficients g_k of Eq. (6.2) are then obtained by multiplying Eq. (6.1) by $e^{-ik\omega t}$ and then integrating both sides of the equation over one period of $f(t)$. We can always take $t_0 = 0$ for convenience, because it is arbitrary.

6.1 Fourier analysis and orthogonal functions

We can generalize the Fourier theorem to a nonperiodic function defined in a region of $x \in [a, b]$ if we have a complete basis set of orthonormal functions $\phi_k(x)$ with

$$\int_a^b \phi_j^*(x)\phi_k(x)\, dx = \langle j|k\rangle = \delta_{jk}. \tag{6.5}$$

For any arbitrary function $f(x)$ defined in the region of $x \in [a, b]$, we can always write

$$f(x) = \sum_j g_j\, \phi_j(x) \tag{6.6}$$

if the function is square integrable, defined by

$$\int_a^b |f(x)|^2\, dx < \infty. \tag{6.7}$$

The summation in the series is over all the possible states in the complete set, and the coefficients g_j are given by

$$g_j = \int_a^b \phi_j^*(x) f(x)\, dx = \langle j|f\rangle. \tag{6.8}$$

The continuous Fourier transform is obtained if we restrict the series to the region of $t \in [-T/2, T/2]$ and then extend the period T to infinity. We need to redefine $j\omega \to \omega$ and $\sum_j \to (1/\sqrt{2\pi}) \int d\omega$. Then the sum becomes an integral

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{-i\omega t}\, d\omega, \tag{6.9}$$

which is commonly known as the Fourier integral. The Fourier coefficient function is given by

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t}\, dt. \tag{6.10}$$

Equations (6.9) and (6.10) define an integral transform and its inverse, which are commonly known as the Fourier transform and the inverse Fourier transform. We can show that the two equations above are consistent: that is, we can obtain the second equation by multiplying the first equation by $e^{i\omega t}$ and then integrating it. We need to use the Dirac δ function during this process. The Dirac δ function is defined by

$$\delta(x - x') = \begin{cases} \infty & \text{if } x = x', \\ 0 & \text{elsewhere,} \end{cases} \tag{6.11}$$

and

$$\int_{-\infty}^{\infty} \delta(x - x')\, dx = \lim_{\epsilon \to +0} \int_{x'-\epsilon}^{x'+\epsilon} \delta(x - x')\, dx = 1. \tag{6.12}$$

So the Dirac δ function $\delta(\omega)$ can also be interpreted as the Fourier transform of a constant function $f(t) = 1/\sqrt{2\pi}$, which we can easily show by carrying out the transform integration.

The Fourier transform can also be applied to other types of variables or to higher dimensions. For example, the Fourier transform of a function $f(\mathbf{r})$ in three dimensions is given by

$$f(\mathbf{r}) = \frac{1}{(2\pi)^{3/2}} \int g(\mathbf{q}) e^{i\mathbf{q}\cdot\mathbf{r}}\, d\mathbf{q}, \tag{6.13}$$

with the Fourier coefficient

$$g(\mathbf{q}) = \frac{1}{(2\pi)^{3/2}} \int f(\mathbf{q}) e^{-i\mathbf{q}\cdot\mathbf{r}}\, d\mathbf{r}. \tag{6.14}$$

Note that both of the above integrals are three-dimensional. The space defined by $\mathbf{q}$ is usually called the *momentum space*.

6.2 Discrete Fourier transform

The Fourier transform of a specific function is usually necessary in the analysis of experimental data, because it is often difficult to establish a clear physical picture just from the raw data taken from an experiment. Numerical procedures for the Fourier transform are inevitable in physics and other scientific fields. Usually we have to deal with a large number of data points, and the speed of the algorithm for the Fourier transform becomes a very important issue. In this section we will discuss a straightforward scheme for the one-dimensional case to illustrate some basic aspects of the DFT (discrete Fourier transform). In the next section, we will outline a fast algorithm for the discrete Fourier transform, commonly known as the FTT (fast Fourier transform).

As we pointed out earlier, the one-dimensional Fourier transform is defined by Eq. (6.9) and Eq. (6.10); as always, we need to convert the continuous variables into discrete ones before we develop a numerical procedure.

Let us consider $f(x)$ as a space-dependent physical quantity obtained from experimental measurements. If the measurements are conducted between $x = 0$ and $x = L$, $f(x)$ is nonzero only for $x \in [0, L]$. To simplify our problem, we can assume that the data are taken at evenly spaced points with each interval $h = L/(N-1)$, where N is the total number of data points. We assume that the data repeat periodically outside the region of $x \in [0, L]$, which is equivalent to imposing the periodic boundary condition on the finite system. The corresponding wavenumber in the momentum space is then discrete too, with an interval $\kappa = 2\pi/L$.

The discrete Fourier transform of such a data set can then be expressed in terms of a summation

$$f_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} g_j e^{i2\pi jk/N}, \tag{6.15}$$

with the Fourier coefficients given by

$$g_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} f_k e^{-i2\pi jk/N}, \tag{6.16}$$

where we have used our convention that $f_k = f(t = k\tau)$ and $g_j = g(\omega = j\nu)$. We can show that the above two summations are consistent, meaning that the inverse Fourier transform of the Fourier coefficients will give the exact values of $f(t)$ at $t = 0, \tau, \ldots, (N-1)\tau$. However, this inverse Fourier transform does not ensure smoothness in the recovered data function.

Note that the exponential functions in the series form a discrete basis set of orthogonal functions: that is,

$$\begin{aligned} \langle\phi_j|\phi_m\rangle &= \sum_{k=0}^{N-1} \frac{1}{\sqrt{N}} e^{-i2\pi kj/N} \frac{1}{\sqrt{N}} e^{i2\pi km/N} \\ &= \frac{1}{N} \sum_{k=0}^{N-1} e^{i2\pi k(m-j)/N} = \delta_{jm}. \end{aligned} \tag{6.17}$$

Before we introduce the fast Fourier transform, let us examine how we can implement the discrete Fourier transform of Eq. (6.16) in a straightforward manner. We can separate the real (Re) and imaginary (Im) parts of the coefficients,

$$\operatorname{Re} g_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \left(\cos\frac{2\pi jk}{N} \operatorname{Re} f_k + \sin\frac{2\pi jk}{N} \operatorname{Im} f_k \right), \tag{6.18}$$

$$\operatorname{Im} g_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \left(\cos\frac{2\pi jk}{N} \operatorname{Im} f_k - \sin\frac{2\pi jk}{N} \operatorname{Re} f_k \right), \tag{6.19}$$

for convenience. Note that it is not necessary to separate them in practice even though most people would do so. Separating the real and imaginary parts is convenient because then we only need to deal with real numbers.

The following program is an implementation of Eqs. (6.18) and (6.19) with the real and imaginary parts of the coefficients separated.

```
// An example of performing the discrete Fourier transform
// with function f(x)=x(1-x) with x in [0,1].  The
// inverse transform is also performed for comparison.

import java.lang.*;
public class Fourier {
  static final int n = 128, m = 8;
  public static void main(String argv[]) {
  double x[] = new double[n];
  double fr[] = new double[n];
  double fi[] = new double[n];
  double gr[] = new double[n];
  double gi[] = new double[n];
  double h = 1.0/(n-1);
  double f0 = 1/Math.sqrt(n);

  // Assign the data and perform the transform
    for (int i=0; i<n; ++i) {
      x[i] = h*i;
      fr[i] = x[i]*(1-x[i]);
      fi[i] = 0;
    }
    dft(fr, fi, gr, gi);

  // Perform the inverse Fourier transform
    for (int i=0; i<n; ++i) {
      gr[i] =  f0*gr[i];
      gi[i] = -f0*gi[i];
      fr[i] = fi[i] = 0;
    }
    dft(gr, gi, fr, fi);
    for (int i=0; i<n; ++i) {
      fr[i] =  f0*fr[i];
      fi[i] = -f0*fi[i];
    }

  // Output the result in every m data steps
    for (int i=0; i<n; i+=m)
      System.out.println(x[i] + " " + fr[i] + " " fi[i]);
  }

// Method to perform the discrete Foruier transform.  Here
// fr[] and fi[] are the real and imaginary parts of the
// data with the correponding outputs gr[] and gi[].

  public static void dft(double fr[], double fi[],
    double gr[], double gi[]) {
    int n = fr.length;
    double x = 2*Math.PI/n;
    for (int i=0; i<n; ++i) {
      for (int j=0; j<n; ++j) {
        double q = x*j*i;
        gr[i] += fr[j]*Math.cos(q)+fi[j]*Math.sin(q);
        gi[i] += fi[j]*Math.cos(q)-fr[j]*Math.sin(q);
      }
    }
  }
}
```

The above program takes an amount of computing time that is proportional to N^2. We have used $f(x) = x(1 - x)$ as the data function to test the method in the region of $x \in [0, 1]$. This program demonstrates how we can perform the discrete Fourier transform and its inverse, that is, how to obtain g_j from f_k, and to obtain f_k from g_j, in exactly the same manner. Note that we do not have the factor $1/\sqrt{N}$ in the method, which is a common practice in designing a Fourier transform subprogram. It is obvious that the inverse transform is nearly identical to the transform except for a sign.

As we have pointed out, the inverse Fourier transform provides exactly the same values as the original data at the data points. The result of the inverse discrete Fourier transform from the above program and the original data are shown in Fig. 6.1. Some inaccuracy can still appear at these data points, because of the

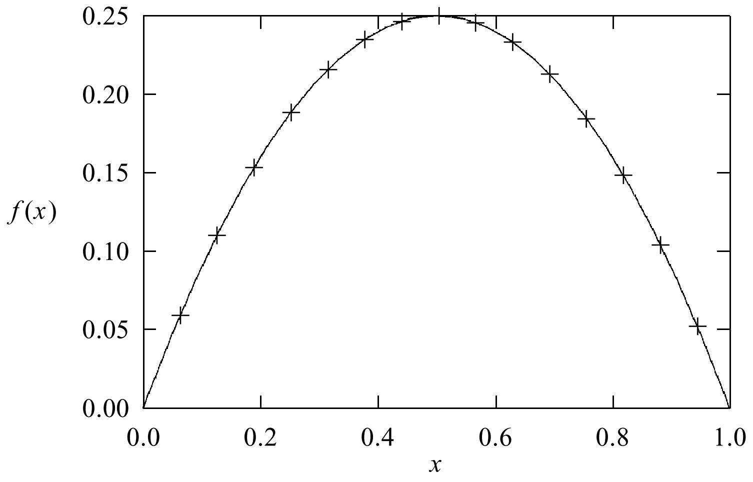


Fig. 6.1 The function $f(x) = x(1 - x)$ (line) and the corresponding values from the inverse transform of its Fourier coefficients (+) calculated with the example program.

rounding error incurred during the computing. A more important issue here is the accuracy of the recovered data function at places other than the data points. When the Fourier transform or its inverse is performed, we can have only a discrete set of data points. So an interpolation is inevitable if we want to know any value that is not at the data points. We can increase the number of points to reduce the errors in the interpolation of the data, but we also have to watch the growth of the rounding errors with the number of points used.

6.3 Fast Fourier transform

As we can clearly see, the straightforward discrete Fourier transform algorithm introduced in the preceding section is very inefficient, because the computing time needed is proportional to N^2. In order to solve this problem, many people have come up with the idea that is now known as the FFT (*fast Fourier transform*). The key element of the fast Fourier transform is to rearrange the terms in the series and to have the summation performed in a hierarchical manner. For example, we can perform a series of pairwise additions to accomplish the summation if the number of data points is of the power of 2: that is, $N = 2^M$, where M is an integer.

The idea behind the fast Fourier transform had been considered a long time ago, even before the first computer was built. As mentioned in Chapter 1, Gauss developed a version of the fast Fourier transform and published his work in neoclassical Latin (Gauss, 1886; Goldstine, 1977, pp. 249–53). However, no one really took notice of Gauss's idea or connected it to modern computation. The fast Fourier transform algorithm was formally discovered and put into practice by Cooley and Tukey (1965). Here we will give a brief outline of their idea.

The simplest fast Fourier transform algorithm is accomplished with the observation that we can separate the odd and even terms in the discrete Fourier transform as

$$\begin{aligned} g_j &= \sum_{k=0}^{N/2-1} f_{2k}\, e^{-i2\pi j(2k)/N} + \sum_{k=0}^{N/2-1} f_{2k+1} e^{-i2\pi j(2k+1)/N}, \\ &= x_j + y_j\, e^{-i2\pi j/N}, \end{aligned} \tag{6.20}$$

where

$$x_j = \sum_{k=0}^{N/2-1} f_{2k}\, e^{-i2\pi jk/(N/2)} \tag{6.21}$$

and

$$y_j = \sum_{k=0}^{N/2-1} f_{2k+1} e^{-i2\pi jk/(N/2)}. \tag{6.22}$$

Here we have ignored the factor $1/\sqrt{N}$, which can always be added back into the program that calls the fast Fourier transform subprogram. What we have done is to rewrite the Fourier transform with a summation of N terms as two summations, each of $N/2$ terms. This process can be repeated further and further until eventually we have only two terms in each summation if $N = 2^M$, where M is an integer. There is one more symmetry between g_k for $k < N/2$ and g_k for $k \geq N/2$: that is,

$$g_j = x_j + w^j y_j, \tag{6.23}$$

$$g_{j+N/2} = x_j - w^j y_j, \tag{6.24}$$

where $w = e^{-i2\pi/N}$ and $j = 0, 1, \ldots, N/2 - 1$. This is quite important in practice, because now we need to perform the transform only up to $j = N/2 - 1$, and the Fourier coefficients with higher j are obtained with the above equation at the same time.

There are two important ingredients in the fast Fourier transform algorithm. After the summation is decomposed M times, we need to add individual data points in pairs. However, due to the sorting of the odd and even terms in every level of decomposition, the points in each pair at the first level of additions can be very far apart in the original data string. However, Cooley and Tukey (1965) found that if we record the data string index with a binary number, a *bit-reversed order* will put each pair of data points next to each other for the summations at the first level. Let us take a set of 16 data points $f_0, f_1, \ldots, f_{15}$ as an example. If we record them with a binary index, we have 0000, 0001, 0010, 0011, ..., 1111, for all the data points. Bit-reversed order is achieved if we reverse the order of each binary string. For example, the bit-reversed order of 1000 is 0001. So the order of the data after the bit reversal is $f_0, f_8, f_4, f_{12}, \ldots, f_3, f_{11}, f_7, f_{15}$. Then the first level of additions is performed between f_0 and f_8, f_4 and f_{12}, ... , f_3 and f_{11}, and f_7 and f_{15}. Equations (6.23) and (6.24) can be applied repeatedly to sum the pairs 2^{l-1} spaces apart in the bit-reversed data stream. Here l indicates the level of additions; for example, the first set of additions corresponds to $l = 1$. At each level of additions, two data points are created from each pair. Note that w^j is associated with the second term in the equation.

With the fast Fourier transform algorithm, the computing time needed for a large set of data points is tremendously reduced. We can see this by examining the calculation steps needed in the transform. Assume that $N = 2^M$, so that after bit reversal, we need to perform M levels of additions and $N/2^l$ additions at the lth level. A careful analysis shows that the total computing time in the fast Fourier transform algorithm is proportional to $N \log_2 N$ instead of to N^2, as is the case for the straightforward discrete Fourier transform. Similar algorithms can be devised for N of the power of 4, 8, and so forth.

Many versions and variations of the fast Fourier transform programs are available in Fortran (Burrus and Parks, 1985) and in other programming languages. One of the earliest Fortran programs of the fast Fourier transform was written by Cooley, Lewis, and Welch (1969). Now many computers come with a fast Fourier transform library, which is usually written in machine language and tailored specifically for the architecture of the system.

Most routines for the fast Fourier transform are written with complex variables. However, it is easier to deal with just real variables. We can always separate the real and imaginary parts in Eqs. (6.20)–(6.24), as discussed earlier. The following method is an example of this. Note also that Java does not have a complex data type at the moment and we need to create a class for complex variables and their operations if we want implement the algorithm without separating the real and imaginary parts of the data. For other languages, it is more concise for programming to have both input and result in a complex form. If the data are real, we can simply remove the lines related to the imaginary part of the input data.

```java
// Method to perform the fast Foruier transform.  Here
// fr[] and fi[] are the real and imaginary parts of
// both the input and output data.

  public static void fft(double fr[], double fi[],
    int m) {
    int n = fr.length;
    int nh = n/2;
    int np = (int) Math.pow(2, m);

  // Stop the program if the indices do not match
    if (np != n) {
      System.out.println("Index mismtch detected");
      System.exit(1);
    }

  // Rearrange the data to the bit-reversed order
    int k = 1;
    for (int i=0; i<n-1; ++i) {
      if (i < k-1) {
        double f1 = fr[k-1];
```

```
          double f2 = fi[k-1];
          fr[k-1] = fr[i];
          fi[k-1] = fi[i];
          fr[i] = f1;
          fi[i] = f2;
        }
        int j = nh;
        while (j < k) {
          k -= j;
          j /= 2;
        }
        k += j;
      }
    // Sum up the reordered data at all levels
      k = 1;
      for (int i=0; i<m; ++i) {
        double w = 0;
        int j = k;
        k = 2*j;
        for (int p=0; p<j; ++p) {
          double u =  Math.cos(w);
          double v = -Math.sin(w);
          w += Math.PI/j;
          for (int q=p; q<n; q+=k) {
            int r = q+j;
            double f1 = fr[r]*u-fi[r]*v;
            double f2 = fr[r]*v+fi[r]*u;
            fr[r] = fr[q]-f1;
            fr[q] += f1;
            fi[r] = fi[q]-f2;
            fi[q] += f2;
          }
        }
      }
    }
```

As we can see from the above method, rearranging the data points into bit-reversed order is nontrivial. However, it can still be understood if we examine the part of the program iteration by iteration with a small N. To be convinced that the above subroutine is a correct implementation of the $N = 2^M$ case, the reader can take $N = 16$ as an example and then work out the terms by hand.

A very important issue here is how much time can use of the fast Fourier transform save. On a scalar machine, it is quite significant. However, on a vector machine, the saving is somewhat restricted. A vector processor can perform the inner loop in the discrete Fourier transform in parallel within each clock cycle, and this makes the computing time for the straightforward discrete Fourier transform proportional to N^α, with α somewhere between 1 and 2. In real problems, α may be a little bit higher than 2 for a scalar machine. However, the advantage of vectorization in the fast Fourier transform is not as significant as in the straightforward discrete Fourier transform. So in general, we may need to examine the problem under study, and the fast Fourier transform is certainly an important tool, because the available computing resources are always limited.

6.4 Power spectrum of a driven pendulum

In Chapter 4 we discussed the fact that a driven pendulum with damping can exhibit either periodic or chaotic behavior. One way to analyze the dynamics of a nonlinear system is to study its power spectrum.

The power spectrum of a dynamical variable is defined as the square of the modulus of the Fourier coefficient function,

$$S(\omega) = |g(\omega)|^2, \tag{6.25}$$

where $g(\omega)$ is given by

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} y(t) e^{i\omega t}\, dt, \tag{6.26}$$

with $y(t)$ being the time-dependent dynamic variable. We discussed in the preceding section how to achieve the fast Fourier transform numerically. With the availability of such a scheme, the evaluation of the power spectrum of a time-dependent variable becomes straightforward.

The driven pendulum with damping is described by

$$\frac{dy_1}{dt} = y_2, \tag{6.27}$$

$$\frac{dy_2}{dt} = -q y_2 - \sin y_1 + b \cos \omega_0 t, \tag{6.28}$$

where $y_1(t) = \theta(t)$ is the angle between the pendulum and the vertical, $y_2(t) = d\theta(t)/dt$ is its corresponding angular velocity, q is a measure of the damping, and b and ω_0 are the amplitude and angular frequency of the external driving force. We have already shown in Chapter 4 how to solve this problem numerically with the fourth-order Runge–Kutta algorithm. The power spectra of the time-dependent angle and the time-dependent angular velocity can be obtained easily by performing a discrete Fourier transform with the fast Fourier transform algorithm. We show the numerical results in Fig. 6.2(a) and (b), which correspond to the power spectra of the time-dependent angle $\theta(t)$ in periodic and chaotic regions, respectively.

We have taken 65 536 data points with 192 points in one period of the driving force. The initial condition, $y_1(0) = 0$ and $y_2(0) = 2$, is used in generating the data. In Fig. 6.2, we have shown the first 2048 points of the power spectra. As we can see from the figure, the power spectrum for periodic motion has separated sharp peaks with δ-function-like distributions. The broadening of the major peak at the angular frequency of the driving force, ω_0, is due to rounding errors. We can reduce the broadening by increasing the number of points used. The power spectrum for the chaotic case is quite irregular, but still has noticeable peaks at the same positions as the periodic case. This is because the contribution at the angular frequency of the driving force is still high, even though the system is chaotic. The peaks at multiples of ω_0 are due to relatively large contributions at the higher frequencies with $\omega = n\omega_0$, where n is an integer, in the Fourier coefficient. If we have more data points, we can also study the fractal structure of the power

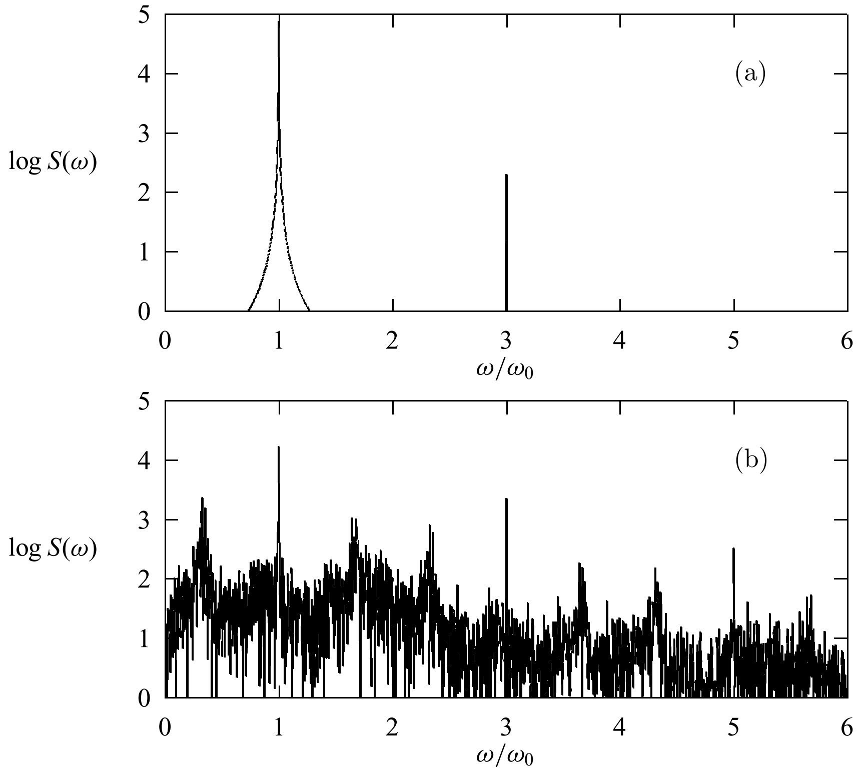


Fig. 6.2 Power spectra of the time-dependent angle of a driven pendulum with damping, with $\omega_0 = 2/3$, $q = 0.5$, for (a) periodic behavior at $b = 0.9$ and (b) chaotic behavior at $b = 1.15$.

spectrum. This could be achieved by examining the data at different scales of the frequency; we would observe similar patterns emerging from different scales with a unique fractal dimensionality.

6.5 Fourier transform in higher dimensions

The Fourier transform can be obtained in a very straightforward manner in higher dimensions if we realize that we can transform each coordinate as if it were a one-dimensional problem, with all other coordinate indices held constant.

Let us take the two-dimensional case as an example. Assume that the data are from a rectangular domain with N_1 mesh points in one direction and N_2 in the other. So the total number of points is $N = N_1 N_2$. The discrete Fourier transform is then given by

$$
\begin{aligned}
g_{jk} &= \frac{1}{\sqrt{N}} \sum_{l=0}^{N_1-1} \sum_{m=0}^{N_2-1} f_{lm} e^{-i2\pi(jl/N_1 + km/N_2)} \\
&= \frac{1}{\sqrt{N_1}} \sum_{l=0}^{N_1-1} e^{-i2\pi jl/N_1} \frac{1}{\sqrt{N_2}} \sum_{m=0}^{N_2-1} f_{lm} e^{-i2\pi km/N_2}. \qquad (6.29)
\end{aligned}
$$

Thus, we can obtain the transform first for all the terms under index m with a fixed l and then for all the terms under index l with a fixed k. The procedure can be seen easily from the method given below.

```
// Method to carry out the fast Fourier transform for a
// 2-dimenisonal array.  Here fr[][] and fi[][] are real
// and imaginary parts in both the input and output.

  public static void fft2d(double fr[][],
    double fi[][], int m1, int m2) {
    int n1 = fr.length;
    int n2 = fr[0].length;
    double hr[] = new double[n2];
    double hi[] = new double[n2];
    double pr[] = new double[n1];
    double pi[] = new double[n1];

 // Perform fft on the 2nd index
    for (int i=0; i<n1; ++i) {
      for (int j=0; j<n2; ++j) {
        hr[j] = fr[i][j];
        hi[j] = fi[i][j];
      }
      fft(hr, hi, m2);
      for (int j=0; j<n2; ++j) {
        fr[i][j] = hr[j];
        fi[i][j] = hi[j];
      }
    }

 // Perform fft on the 1st index
    for (int j=0; j<n2; ++j) {
      for (int i=0; i<n1; ++i) {
        pr[i] = fr[i][j];
        pi[i] = fi[i][j];
      }
      fft(pr, pi, m1);
      for (int i=0; i<n1; ++i) {
        fr[i][j] = pr[i];
        fi[i][j] = pi[i];
      }
    }
  }

  public static void fft(double fr[], double fi[],
    int m) {...}
```

We have used the one-dimensional fast Fourier transform twice, once for each of the two indices. This procedure works well if the boundary of the data is rectangular.

6.6 Wavelet analysis

Wavelet analysis was first introduced by Haar (1910) but not recognized as a powerful mathematical tool until the 1980s. It was Morlet *et al.* (1982a; 1982b) who first used the wavelet approach in seismic data analysis. The wavelet transform contains spectral information at different scales and different locations of the data stream, for example, the intensity of a signal around a specific frequency and a specific time. This is in contrast to Fourier analysis, in which a specific transform coefficient contains information about a specific scale or frequency from the entire data space without referring to its relevance to the location in the original data stream. The wavelet method is extremely powerful in the analysis of short time signals, transient data, or complex patterns. The development and applications of wavelet analysis since the 1980s have shown that many more applications will emerge in the future. Here we give just a brief introduction to the subject and point out its potential applications in computational physics. More details on the method and some applications can be found in several monographs (Daubechies, 1992; Chui, 1992; Meyer, 1993; Young, 1993; Holschneider, 1999; Percival and Walden, 2000) and collections (Combes, Grossmann, and Tchamitchian, 1990; Chui, Montefusco, and Puccio, 1994; Foufoula-Georgiou and Kumar, 1994; Silverman and Vassilicos, 1999; van den Berg, 1999).

Windowed Fourier transform

It is desirable in many applications that the local structure in a set of data can be amplified and analyzed. This is because we may not be able to obtain the data throughout the entire space or on a specific scale of particular interest. Sometimes we also want to filter out the noise around the boundaries of the data. A *windowed Fourier transform* can be formulated to select the information from the data at a specific location. We can define

$$g(\omega, \tau) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) w(t-\tau) e^{i\omega t}\, dt \tag{6.30}$$

as the windowed Fourier transform of the function $f(t)$ under the window function $w(t-\tau)$. The window function $w(t-\tau)$ is used here to extract information about $f(t)$ in the neighborhood of $t = \tau$. The window function $w(t)$ is commonly chosen to be a real, even function, and satisfies

$$\int_{-\infty}^{\infty} w^2(t)\, dt = 1. \tag{6.31}$$

Typical window functions include the triangular window function

$$w(t) = \begin{cases} \frac{1}{\sqrt{\mathcal{N}}}\left(1 - \frac{|t|}{\sigma}\right) & \text{if } |t| < \sigma, \\ 0 & \text{elsewhere,} \end{cases} \tag{6.32}$$

and the Gaussian window function

$$w(t) = \frac{1}{\sqrt{\mathcal{N}}} e^{-t^2/2\sigma^2}, \tag{6.33}$$

where $\mathcal{N}$ is the normalization constant and σ is a measure of the window width. As soon as σ is selected, $\mathcal{N}$ can be evaluated with Eq. (6.31). From the definition, we can also recover the data from their Fourier coefficients through the inverse transform

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega, \tau) w(\tau - t) e^{-i\omega t} \, d\omega \, d\tau \tag{6.34}$$

as in the conventional Fourier transform. The inner product, that is, the integral over the square of the data amplitude, is equal to the integral over the square of its transform coefficient amplitude: that is,

$$\int_{-\infty}^{\infty} |f(t)|^2 dt = \int_{-\infty}^{\infty} |g(\omega, \tau)|^2 \, d\omega \, d\tau. \tag{6.35}$$

The advantage of the windowed Fourier transform is that it tunes the data with the window function so that we can obtain the local structure of the data or suppress unwanted effects in the data string. However, the transform treats the whole data space uniformly and would not be able to distinguish detailed structures of the data at different scales.

Continuous wavelet transform

The windowed Fourier transform can provide information at a given location in time, but it fails to provide the data with a specific scale at the selected location. We can obtain information about a set of data locally and also at different scales through wavelet analysis.

The continuous wavelet transform of a function $f(t)$ is defined through the integral

$$g(\lambda, \tau) = \int_{-\infty}^{\infty} f(t) u_{\lambda\tau}^{*}(t) \, dt, \tag{6.36}$$

where $u_{\lambda\tau}^{*}(t)$ is the complex conjugate of the *wavelet*

$$u_{\lambda\tau}(t) = \frac{1}{\sqrt{|\lambda|}} u\left(\frac{t - \tau}{\lambda}\right), \tag{6.37}$$

with $\lambda \neq 0$ being the *dilate* and τ being the *translate* of the wavelet transform. The parameters λ and τ are usually chosen to be real, and they select, respectively, the scale and the location of the data stream during the transformation. The function $u(t)$ is the generator of all the wavelets $u_{\lambda\tau}(t)$ and is called the *mother wavelet* or just the *wavelet*. Note that $u_{10}(t) = u(t)$ as expected for the case without any dilation or translation in picking up the data.

There are some constraints on the selection of a meaningful wavelet $u(t)$. For example, in order to have the inverse transform defined, we must have

$$\mathcal{Z} = \int_{-\infty}^{\infty} \frac{1}{|\omega|} |z(\omega)|^2 \, d\omega < \infty, \tag{6.38}$$

where $z(\omega)$ is the Fourier transform of $u(t)$. The above constraint is called the *admissibility condition* of the wavelet (Daubechies, 1992), which is equivalent to

$$z(0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} u(t)\, dt = 0, \tag{6.39}$$

if $u(t)$ is square integrable and decays as $t \to \pm\infty$. This is why $u(t)$ is called a wavelet, meaning a small wave, in comparison with a typical plane wave, which satisfies the above condition but is not square integrable. The wavelet $u(t)$ is usually normalized with

$$\langle u|u\rangle = \int_{-\infty}^{\infty} |u(t)|^2 dt = 1 \tag{6.40}$$

for convenience.

Let us examine a simple wavelet

$$u(t) = \Theta(t) - 2\Theta\left(t - \frac{1}{2}\right) + \Theta(t-1), \tag{6.41}$$

which is called the Haar wavelet, with the step function

$$\Theta(t) = \begin{cases} 1 & \text{if } t > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{6.42}$$

The wavelet transform coefficients $g(\lambda, \tau)$ are obtained through Eq. (6.36) with the wavelet given by

$$u_{\lambda\tau}(t) = \frac{1}{\sqrt{|\lambda|}} \left[\Theta(t-\tau) - 2\Theta\left(t - \tau - \frac{\lambda}{2}\right) + \Theta(t-\tau-\lambda)\right]. \tag{6.43}$$

The difficulty here is to have an analytic form of $g(\lambda, \tau)$, even for a simple form of $f(t)$. In general the integration must be carried out numerically. Let us consider a simple function $f(t) = t(1-t)$ for $t \in [0, 1]$, zero otherwise. The following program shows how to obtain the continuous wavelet transform of the function numerically.

```
// An example of performing the continuous wavelet
// transform with function f(t)=t(1-t) for 0<t<1.

import java.lang.*;
public class Wavelet{
  static final int nt = 21, nv = 11;
  public static void main(String argv[]) {
  double t[] = new double[nt];
  double f[] = new double[nt];
  double fi[] = new double[nt];
  double v[] = new double[nv];
  double g[][] = new double[nv][nv];
  double dt = 1.0/(nt-1), dv = 1.0/(nv-1);
  double rescale = 100;

  // Assign the data, dilate, and translate
    for (int i=0; i<nt; ++i) {
      t[i] = dt*i;
      f[i] = t[i]*(1-t[i]);
    }
    for (int i=0; i<nv; ++i) v[i] = dv*(i+1);

  // Perform the transform
    for (int i=0; i<nv; ++i){
      double sa = Math.sqrt(Math.abs(v[i]));
      for (int j=0; j<nv; ++j){
        for (int k=0; k<nt; ++k){
          fi[k]   = f[k]*(theta(t[k]-v[j])
                  - 2*theta(t[k]-v[j]-v[i]/2)
                  + theta(t[k]-v[j]-v[i]))/sa;
        }
        g[i][j] = simpson(fi,dt);
      }
    }

  // Output the coefficients obtained
    for (int i=0; i<nv; ++i){
      for (int j=0; j<nv; ++j){
        double g2 = rescale*g[i][j]*g[i][j];
        System.out.println(v[i] + " " + v[j] + " " + g2);
      }
    }
  }

// Method to create a step funciton.

  public static double theta(double x) {
    if (x>0) return 1.0;
    else return 0.0;
  }

  public static double simpson(double y[], double h) {...}
}
```

In Fig. 6.3 we show the surface and contour plots of $|g(\lambda, \tau)|^2$, generated with the above program. The contour plot is called the *scalogram* of $f(t)$, and can be interpreted geometrically in analyzing the information contained in the data (Grossmann, Kronland-Martinet, and Morlet, 1989).

From the definition of the continuous wavelet transform and the constraints on the selection of the wavelet $u(t)$, we can show that the data function can be recovered from the inverse wavelet transform

$$f(t) = \frac{1}{\mathcal{Z}} \int_{-\infty}^{\infty} \frac{1}{\lambda^2} g(\lambda, \tau) u_{\lambda\tau}(t)\, d\lambda\, d\tau. \tag{6.44}$$

We can also show that the wavelet transform satisfies an identity similar to that in the Fourier transform or the windowed Fourier transform with

$$\int_{-\infty}^{\infty} |f(t)|^2 dt = \frac{1}{\mathcal{Z}} \int_{-\infty}^{\infty} \frac{1}{\lambda^2} |g(\lambda, \tau)|^2 d\lambda\, d\tau. \tag{6.45}$$

Analytically, the continuous wavelet transform is easier to deal with than the discrete wavelet transform. However, most data obtained are discrete in nature.

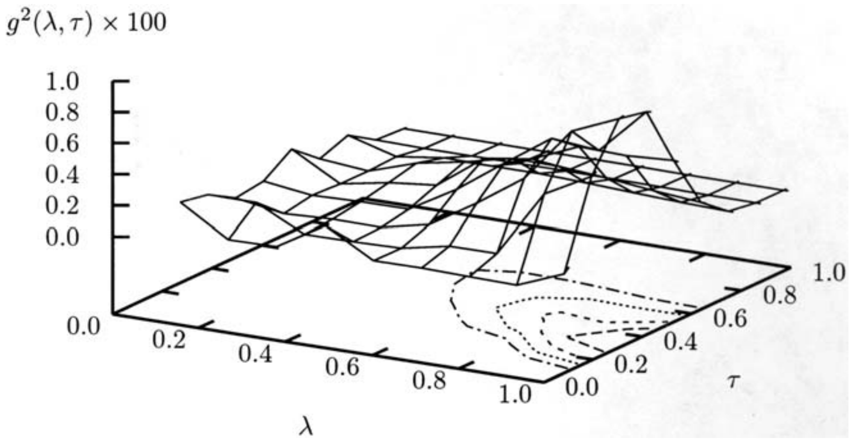


Fig. 6.3 Surface and contour plots of the scalogram of the function $f(t) = t(1 - t)$ for $t \in [0, 1]$ and zero otherwise.

More importantly, it is much easier to implement the data analysis numerically if the transform is defined with discrete variables.

6.7 Discrete wavelet transform

From the continuous wavelet transform, we can obtain detailed information on the data at different locations and scales. The drawback is that the information received is redundant because we effectively decompose a one-dimensional data stream into a two-dimensional data stream. To remove this redundancy, we can take the wavelet at certain selected scales (dyadic scales at $\lambda_j = 2^j$ for integers j) and certain locations (k points apart on the scale of λ_j). The transform can then be achieved level by level with the multiresolution analysis of Mallat (1989) under a pyramid scheme.

In the spirit of carrying out the Fourier analysis in terms of the Fourier series, we expand the time sequence as

$$f(t) = \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} c_{jk} u_{jk}(t), \tag{6.46}$$

where the basis set

$$u_{jk}(t) = 2^{j/2} u(2^j t - k) \tag{6.47}$$

is generated from dilating and translating the wavelet $u(t)$. The transform is obtained from

$$c_{jk} = \langle u_{jk} | f \rangle = \int_{-\infty}^{\infty} f(t) u_{jk}(t)\, dt, \tag{6.48}$$

following the general argument that $u_{jk}(t)$ form a complete, orthonormal basis set. The expansion in Eq. (6.46) is also known as *synthesis* because we can reconstruct the time sequence $f(t)$ if all the coefficients c_{jk} are given. Consequently, the transformation of Eq. (6.48) is called *analysis*. To simplify our problem, we have taken the basis set to be orthonormal even though completeness is the only necessary condition.

We have also made the assumption that the basis set is real and orthonormal for simplicity, even though completeness is the only necessary condition needed. An example of an orthonormal wavelet is the Haar wavelet discussed in the preceding section. We can follow the approach that we have used in the preceding section for the continuous wavelet transform to obtain all the integrals in the transform coefficients. However, the hierarchical structure of the discrete wavelet transform allows us to use a much more efficient method to obtain the transform without worrying about all the integrals involved.

Multiresolution analysis

We can first define a set of linear vector spaces W_j, with each span over the function space covered by u_{jk} for $-\infty < k < \infty$, allowing us to decompose $f(t)$ into components residing in individual W_j with

$$f(t) = \sum_{j=-\infty}^{\infty} d_j(t), \tag{6.49}$$

where $d_j(t)$ is called the *detail* of $f(t)$ in W_j and is given by

$$d_j(t) = \sum_{k=-\infty}^{\infty} c_{jk} u_{jk}(t). \tag{6.50}$$

Then we can introduce another set of linear vector spaces V_j, with each the addition of its nested subspace $V_{j-1} \subset V_j$ and W_{j-1}. We can visualize this by examining three-dimensional Euclidean space. If V_j were taken as the entire Euclidean space and V_{j-1} the xy plane, W_{j-1} would be the z axis. Note specifically that $u(t-k)$ form the complete basis set for the space W_0. A symbolic sketch of the spaces W_j and V_j is given in Fig. 6.4. Note that there is no overlap between any two different W_j spaces, whereas V_j are nested within V_k for $j < k$.

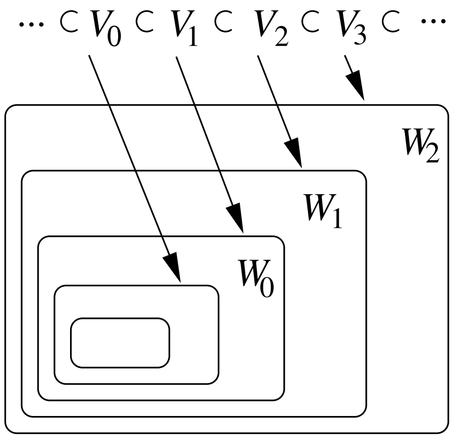
Then the projection of $f(t)$ into the space V_j can be written as

$$a_j(t) = \sum_{k=-\infty}^{j-1} \sum_{l=-\infty}^{\infty} c_{kl} u_{kl}(t), \tag{6.51}$$

with the limits $a_{\infty}(t) = f(t)$ and $a_{-\infty}(t) = 0$. The function $a_j(t)$ is called the *approximation* of $f(t)$ in V_j. The spaces W_j and V_j, and the approximations $a_j(t)$ and details $d_j(t)$ are simply related by

$$V_{j+1} = V_j \oplus W_j, \tag{6.52}$$

$$a_{j+1}(t) = a_j(t) + d_j(t). \tag{6.53}$$



$\cdots \subset V_0 \subset V_1 \subset V_2 \subset V_3 \subset \cdots$
W_2
W_1
W_0

Fig. 6.4 A symbolic representation of the spaces W_j, which are orthogonal with each other, and the spaces V_j, which are nested within V_k for $j < k$.

The above relations define the multiresolution of the space and function. The nested spaces V_j possess certain unique properties. For example, $V_{-\infty}$ contains only one element, the zero vector, which is shared among all the spaces V_j. This means that the projection of any function $f(t)$ into $V_{-\infty}$ is zero. Furthermore, if $f(t)$ is a function in space V_j, $f(2t)$ is automatically a function in space V_{j+1}, and vice versa. This is the consequence of $u_{j+1k}(t) = \sqrt{2}u_{jk}(2t)$. The space V_∞ becomes equivalent to the space $L^2(R)$, which holds all the square-integrable functions $f(t)$.

Let us consider a simple example of decomposing a function $f(t)$ given in a certain space V_j a few times. The space hierarchy is given as

$$\begin{aligned} V_j &= V_{j-1} \oplus W_{j-1} \\ &= V_{j-2} \oplus W_{j-2} \oplus W_{j-1} \\ &= V_{j-3} \oplus W_{j-3} \oplus W_{j-2} \oplus W_{j-1}, \end{aligned} \tag{6.54}$$

which allows us to expand the function as

$$\begin{aligned} f(t) &= A_1(t) + D_1(t) \\ &= A_2(t) + D_2(t) + D_1(t) \\ &= A_3(t) + D_3(t) + D_2(t) + D_1(t), \end{aligned} \tag{6.55}$$

where $A_1(t)$ is the first-level approximation or the projection of $f(t)$ into V_{j-1}, $A_2(t)$ is the second-level approximation or the projection into V_{j-2}, $D_1(t)$ is the first-level detail in W_{j-1}, $D_2(t)$ is the second-level detail in W_{j-2}, and so forth. This is just a special case of $f(t) = a_j(t)$ with $A_k(t) = a_{j-k}(t)$ and $D_k(t) = d_{j-k}(t)$. The essence of the multiresolution analysis is therefore to decompose a data sequence into difference levels of approximations and details.

Scaling function

The multiresolution analysis of $f(t)$ can be realized in a concise form if there exist a set of functions $v_{jk}(t)$ that form a complete, orthogonal basis set in space V_j with

$$\langle v_{jk}(t)|v_{jl}(t)\rangle = \int_{-\infty}^{\infty} v_{jk}(t)v_{jl}(t)\,dt = \delta_{kl}. \tag{6.56}$$

Here $v_{jk}(t)$ are called scaling functions that satisfy

$$v_{jk}(t) = 2^{j/2}v(2^j t - k), \tag{6.57}$$

where the scaling function $v(t)$ is sometimes also referred to as the *father scaling function* or *father wavelet*. Note that $v_{jk}(t)$ with different j cannot be made orthogonal to each other because the spaces V_j are nested rather than orthogonal, unlike the spaces W_j. More importantly, V_0 and W_0 are both the subspaces of V_1 because $V_0 \oplus W_0 = V_1$. We can therefore expand a basis function in V_0, $v(t) = v_{00}(t)$, and a basis function in W_0, $u(t) = u_{00}(t)$, in terms of the basis set in V_1 as

$$v(t) = \sum_{k=-\infty}^{\infty} h(k)v_{1k}(t) = \sum_{k=-\infty}^{\infty} h(k)\sqrt{2}v(2t-k), \tag{6.58}$$

$$u(t) = \sum_{k=-\infty}^{\infty} g(k)v_{1k}(t) = \sum_{k=-\infty}^{\infty} g(k)\sqrt{2}v(2t-k), \tag{6.59}$$

where $h(k)$ and $g(k)$ are referred to as filters, which we will discuss in more detail below. The two-scale relations given in Eqs. (6.58) and (6.59) are instrumental to the development of an efficient algorithm in achieving the discrete wavelet transform.

There are certain properties that we can derive from the orthogonal conditions of the scaling functions and wavelets. For example, from the integration over time on the two-scale relation for the scaling function, we obtain

$$\sum_{k=-\infty}^{\infty} h(k) = \sqrt{2}. \tag{6.60}$$

Furthermore, because $\langle v(t)|v(t-l)\rangle = \delta_{0l}$, we also have

$$\sum_{k=-\infty}^{\infty} h(k)h(k+2l) = \delta_{0l}, \tag{6.61}$$

after applying the two-scale equation for the scaling function. Furthermore, we can show, from the Fourier transform of the scaling functions, that

$$\int_{-\infty}^{\infty} v(t)\,dt = 1. \tag{6.62}$$

Using the admissibility condition

$$\int_{-\infty}^{\infty} u(t)\,dt = 0 \tag{6.63}$$

and $\langle v(t)|u(t-l)\rangle = \delta_{0l}$, a consequence of W_0 being orthogonal to V_0, we can also obtain

$$g(k) = (-1)^k h(r-k), \tag{6.64}$$

where r is an arbitrary, odd integer. However, if the number of nonzero $h(k)$ is finite and equal to n, we must have $r = n - 1$.

In order to have all $h(k)$ for $k = 0, 1, \ldots, n-1$ determined, we need a total of n independent equations. Equations (6.60) and (6.61) provide a total of $n/2 + 1$ equations. So we are still free to impose more conditions on $h(k)$. If we want to recover polynomial data up to the $(n/2 - 1)$th order, the moments under the filters must satisfy (Strang, 1989)

$$\sum_{k=0}^{n-1} (-1)^k k^l h(k) = 0, \tag{6.65}$$

for $l = 0, 1, \ldots, n/2 - 1$, which supplies another $n/2$ equations. Note that the case of $l = 0$ in either Eq. (6.61) or Eq. (6.65) can be derived from Eq. (6.60) and other given relations. So Eqs. (6.60), (6.61), and (6.65) together provide n independent equations for n coefficients $h(k)$, and therefore they can be uniquely determined for a given n. For small n we can solve $h(k)$ easily with this procedure.

Now let us use a simple example to illustrate the points made above. Consider here the case of the Haar scaling function and wavelet for $n = 2$. The Haar scaling function is a box between 0 and 1, namely, $v(t) = \Theta(t) - \Theta(t-1)$, where $\Theta(t)$ is the step function. Then we have

$$v(t) = v(2t) + v(2t-1), \tag{6.66}$$

which gives $h(0) = h(1) = 1/\sqrt{2}$ and $h(k) = 0$ for $k \neq 0, 1$. We also have $g(0) = 1/\sqrt{2}$, $g(1) = -1/\sqrt{2}$, and $g(k) = 0$ if $k \neq 0, 1$, which comes from $g(k) = (-1)^k h(n-1-k) = (-1)^k h(1-k)$, with $n = 2$. Considering now $n = 4$, we have

$$\begin{aligned} h(0) &= \frac{1+\sqrt{3}}{4\sqrt{2}};\ h(1) = \frac{3+\sqrt{3}}{4\sqrt{2}};\\ h(2) &= \frac{3-\sqrt{3}}{4\sqrt{2}};\ h(3) = \frac{1-\sqrt{3}}{4\sqrt{2}}, \end{aligned} \tag{6.67}$$

which is commonly known as the D4 wavelet, named after Daubechies (1988). For even larger n, however, we may have to solve the n coupled equations for $h(k)$ numerically.

A better scheme can be devised (Daubechies, 1988) through the zeros z_k of a polynomial

$$p(z) = \sum_{k=0}^{n/2-1} \frac{(n/2-1+k)!}{k!(n/2-1)!} \frac{(z-1)^{2k}}{(-z)^k} \tag{6.68}$$

inside the unit circle. The coefficients $h(k)$ are given by $h(k) = b_k/\sqrt{\mathcal{N}}$, where b_k is from the expansion

$$(z+1)^{n/2} \prod_{k=1}^{n/2-1} (z - z_k) = \sum_{k=0}^{n-1} b_k z^{n-k-1} \tag{6.69}$$

and the normalization constant $\mathcal{N} = \sum_{k=0}^{n-1} b_k^2$.

In general, we can express the basis functions for the spaces V_{j-1} and W_{j-1} in terms of those for the space V_j as

$$v_{j-1\,k}(t) = \sum_{l=-\infty}^{\infty} h(l) v_{j\,2k+l}(t), \tag{6.70}$$

$$u_{j-1\,k}(t) = \sum_{l=-\infty}^{\infty} g(l) v_{j\,2k+l}(t), \tag{6.71}$$

following Eqs. (6.47), (6.57), (6.58), and (6.59). This provides a means for us to decompose a given set of data level by level, starting from any chosen resolution.

Filter bank and pyramid algorithm

Now let us turn to the analysis of the data in the spaces V_j and W_j. If we start by approximating the data function $f(t)$ by $A_0(t) = a_j(t)$ with a reasonably large j, we can perform the entire wavelet analysis level by level. First we decompose $A_0(t)$ once to have

$$A_0(t) = \sum_{k=-\infty}^{\infty} a^{(0)}(k) v_{jk}(t) = A_1(t) + D_1(t), \tag{6.72}$$

where

$$A_1(t) = \sum_{k=-\infty}^{\infty} a^{(1)}(k) v_{j-1k}(t), \tag{6.73}$$

$$D_1(t) = \sum_{k=-\infty}^{\infty} d^{(1)}(k) u_{j-1k}(t). \tag{6.74}$$

This is the consequence of $V_j = V_{j-1} \oplus W_{j-1}$, and $v_{jk}(t)$ for all integers k form the basis set for V_j and $u_{jk}(t)$ for all integers k form the basis set for W_j. The coefficients in the above expression are obtained from

$$a^{(1)}(k) = \langle v_{j-1\,k} | A_1 \rangle = \langle v_{j-1\,k} | A_0 \rangle = \sum_{l=-\infty}^{\infty} h(l-2k) a^{(0)}(l), \tag{6.75}$$

$$d^{(1)}(k) = \langle u_{j-1\,k} | D_1 \rangle = \langle u_{j-1\,k} | A_0 \rangle = \sum_{l=-\infty}^{\infty} g(l-2k) a^{(0)}(l). \tag{6.76}$$

We can then decompose $A_1(t)$ into $A_2(t)$ and $D_2(t)$ to obtain the next level analysis. The transform is used to find $a^{(n)}(k)$ and $d^{(n)}(k)$, or $A_n(t)$ and $D_n(t)$, provided the filters $h(k)$ and $g(k)$ are given. Now if we know $a^{(0)}(k)$ and the filters,

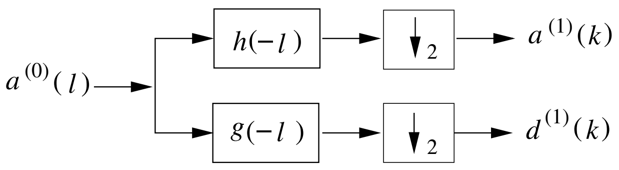


Fig. 6.5 The filtering and downsampling processes of the first-level decomposition in the discrete wavelet transform.

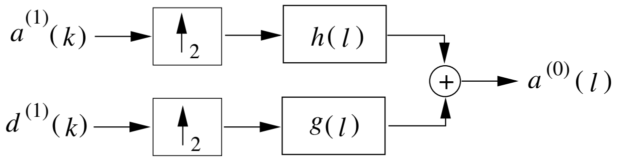


Fig. 6.6 The filtering and upsampling processes of the one-level reconstruction in the discrete wavelet transform.

we can obtain the transform, that is, calculate $a^{(n)}(k)$ and $d^{(n)}(k)$ for a given integer n, easily. As soon as we have the scaling function and its corresponding filters specified, we also have the wavelet from the second two-scale relation and the coefficients $a^{(0)}(k) = \langle v(t-k)|f(t)\rangle$. A digital filter is defined by the convolution

$$a * b(t) = \sum_{k=-\infty}^{\infty} a(k)b(t-k) = \sum_{k=-\infty}^{\infty} a(k-t)b(t), \tag{6.77}$$

where $a(t)$ is called the filter and the time sequence $b(t)$ is the data function that is filtered by $a(t)$. From Eqs. (6.75) and (6.76), we see that the filters associated with the wavelet analysis are time-reversed and skip every other data point during the filtering, which is called *downsampling*. We therefore can represent each level of analysis by two operations, the filtering and downsampling, as shown in Fig. 6.5. The downsampling is equivalent to converting function $x(k)$ into $x(2k)$. The analysis can be continued with the same pair of filters to the next level with a decomposition of $A_1(t)$ into $A_2(t)$ and $D_2(t)$, and so forth, forming a multistage filter bank pyramid. We can design a pyramid algorithm to carry out n levels of decompositions to obtain $\{a^{(n)}, d^{(n)}, \ldots, d^{(1)}\}$, for $n \geq 1$.

The inverse transform (synthesis) can be obtained with the same pair of filters without reversing the time. From the expansion of Eq. (6.72), we have

$$\begin{aligned} a^{(0)}(k) &= \langle v_{jk}(t)|A_0(t)\rangle \\ &= \sum_{l=-\infty}^{\infty} a^{(1)}(l)h(k-2l) + \sum_{l=-\infty}^{\infty} d^{(1)}(l)g(k-2l), \end{aligned} \tag{6.78}$$

which can be interpreted as a combination of an *upsampling* and a filtering operation. The upsampling is equivalent to converting function $x(k)$ into $x(k/2)$ for even k and into zero for odd k. Graphically, we can express this one level of synthesis as that shown in Fig. 6.6. Of course, the process can be continued to the second level of synthesis, the third level of synthesis, and so forth, until we have the time sequence completely reconstructed.

Like the discrete Fourier transform, the discrete wavelet transform can also be used in higher-dimensional spaces. Interested readers can find discussions of the two-dimensional wavelet transform in Newland (1993). There are many other scaling functions and wavelets available; interested readers should consult the references cited at the beginning of the preceding section. For a particular problem, one may work better than others.

6.8 Special functions

The solutions of a special set of differential equations can be expressed in terms of polynomials. In some cases, each polynomial has an infinite number of terms. For example, the Schrödinger equation for a particle in a central potential $V(r)$ is

$$-\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{r}) + V(r)\Psi(\mathbf{r}) = \varepsilon\Psi(\mathbf{r}), \tag{6.79}$$

with

$$\nabla^2 = \frac{1}{r^2}\frac{\partial}{\partial r}r^2\frac{\partial}{\partial r} + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\sin\theta\frac{\partial}{\partial\theta} + \frac{1}{r^2\sin^2\theta}\frac{\partial^2}{\partial\phi^2} \tag{6.80}$$

in spherical coordinates. We can assume that the solution $\Psi(\mathbf{r})$ is of the form

$$\Psi(r,\theta,\phi) = R(r)Y(\theta,\phi), \tag{6.81}$$

and then the equation becomes

$$\begin{aligned}\frac{1}{R}\frac{d}{dr}r^2\frac{dR}{dr} + \frac{2mr^2}{\hbar^2}(\varepsilon - V) &= -\frac{1}{Y}\left(\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\sin\theta\frac{\partial Y}{\partial\theta} + \frac{1}{\sin^2\theta}\frac{\partial^2 Y}{\partial\phi^2}\right)\\ &= \lambda,\end{aligned} \tag{6.82}$$

where λ is an introduced parameter to be determined by solving the eigenvalue problem of $Y(\theta,\phi)$. We can further assume that $Y(\theta,\phi) = \Theta(\theta)\Phi(\phi)$ and then the equation for $\Theta(\theta)$ becomes

$$\frac{1}{\sin\theta}\frac{d}{d\theta}\sin\theta\frac{d\Theta}{d\theta} + \left(\lambda - \frac{m^2}{\sin^2\theta}\right)\Theta = 0, \tag{6.83}$$

with the corresponding equation for $\Phi(\phi)$ given as

$$\Phi''(\phi) = -m^2\Phi(\phi), \tag{6.84}$$

where m is another introduced parameter and has to be an integer, because $\Phi(\phi)$ needs to be a periodic function of ϕ, that is, $\Phi(\phi + 2\pi) = \Phi(\phi)$. Note that $\Phi(\phi) = Ae^{im\phi} + Be^{-im\phi}$ is the solution of the differential equation for $\Phi(\phi)$.

In order for the solution of $\Theta(\theta)$ to be finite everywhere with $\theta \in [0,\pi]$, we must have $\lambda = l(l+1)$, where l is a positive integer. Such solutions $\Theta(\theta) = P_l^m(\cos\theta)$ are called associated Legendre polynomials and can be written as

$$P_l^m(x) = (1-x^2)^{m/2}\frac{d^m}{dx^m}P_l(x), \tag{6.85}$$

where $P_l(x)$ are the Legendre polynomials that satisfy the recursion

$$(l+1)P_{l+1}(x) = (2l+1)xP_l(x) - lP_{l-1}(x), \tag{6.86}$$

starting with $P_0(x) = 1$ and $P_1(x) = x$. We can then obtain all $P_l(x)$ from Eq. (6.86) with $l = 2, 3, 4, \ldots$. We can easily show that $P_l(x)$ is indeed the solution of the equation

$$\frac{d}{dx}(1-x^2)\frac{d}{dx}P_l(x) + l(l+1)P_l(x) = 0 \tag{6.87}$$

and is a polynomial of order l in the region of $x \in [-1, 1]$, as discussed in Section 5.8 in the calculation of the electronic structures of atoms.

Legendre polynomials are useful in almost every subfield of physics and engineering where partial differential equations involving spherical coordinates need to be solved. For example, the electrostatic potential of a charge distribution $\rho(\mathbf{r})$ can be written as

$$\Phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\int \frac{\rho(\mathbf{r}')\,d\mathbf{r}'}{|\mathbf{r}-\mathbf{r}'|}, \tag{6.88}$$

where $1/|\mathbf{r}-\mathbf{r}'|$ can be expanded with the application of Legendre polynomials as

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|} = \sum_{l=0}^{\infty}\frac{r_<^l}{r_>^{l+1}}P_l(\cos\theta), \tag{6.89}$$

where $r_<$ ($r_>$) is the smaller (greater) value of r and r' and θ is the angle between $\mathbf{r}$ and $\mathbf{r}'$. So if we can generate $P_l(\cos\theta)$, we can evaluate the electrostatic potential for any given charge distribution $\rho(\mathbf{r})$ term by term.

The Legendre polynomials can be produced from the recursion given in Eq. (6.86). Below is a simple implementation of the recursion to create the Legendre polynomials for a given x and a maximum l.

```
// Method to create the Legendre polynomials p_l(x) for
// a given x and maximum l.

  public static double[] p (double x, int lmax) {
    double pl[] = new double[lmax+1];
    pl[0] = 1;
    if (lmax==0) return pl;
    else {
      pl[1] = x;
      if (lmax==1) return pl;
      else {
        for (int l=1; l<lmax; ++l)
          pl[l+1] = ((2*l+1)*x*pl[l]
                    -(l+1)*pl[l-1])/(l+1);
        return pl;
      }
    }
  }
```

Legendre polynomials $P_l(x)$ form a complete set of orthogonal basis functions in the region of $x \in [-1, 1]$, which can be used to achieve the least-squares approximations, as discussed in Chapter 2, or to accomplish a generalized Fourier transform. If we choose

$$U_l(x) = \sqrt{\frac{2l+1}{2}}\, P_l(x), \tag{6.90}$$

we can easily show that $U_l(x)$ satisfy

$$\int_{-1}^{1} U_l(x) U_{l'}(x)\, dx = \delta_{ll'}. \tag{6.91}$$

A function $f(x)$ in the region of $x \in [-1, 1]$ can be written as

$$f(x) = \sum_{l=0}^{\infty} a_l U_l(x) \tag{6.92}$$

with

$$a_l = \int_{-1}^{1} f(x) U_l(x)\, dx. \tag{6.93}$$

This is a generalized Fourier transform of the function $f(t)$. In fact, almost every aspect associated with the Fourier transform can be generalized to other orthogonal functions that form a complete basis set. There is a whole class of orthogonal polynomials that are similar to Legendre polynomials and can be applied to similar problems in the same fashion. Detailed discussions on these orthogonal polynomials can be found in Hochstrasser (1965).

In the case of cylindrical coordinates, the equation governing the Laplace operator in the radial direction is the so-called Bessel equation

$$\frac{d^2 J(x)}{dx^2} + \frac{1}{x}\frac{dJ(x)}{dx} + \left(1 - \frac{\nu^2}{x^2}\right) J(x) = 0, \tag{6.94}$$

where ν is a parameter and the solution of the equation is called the Bessel function of order ν. Here ν can be an integer or a fraction. The recursion for the Bessel functions is

$$J_{\nu\pm 1}(x) = \frac{2\nu}{x} J_\nu(x) - J_{\nu\mp 1}(x), \tag{6.95}$$

which can be used in a similar fashion to Legendre polynomials to generate $J_{\nu\pm 1}(x)$ from $J_\nu(x)$ and $J_{\nu\mp 1}(x)$.

Bessel functions can be further divided into two types, depending on their asymptotic behavior. We call the ones with finite values as $x \to 0$ functions of the first kind and denote them by $J_\nu(x)$; the ones that diverge as $x \to 0$ are called functions of the second kind and are denoted by $Y_\nu(x)$. We will consider only the case where ν is an integer. Both $J_\nu(x)$ and $Y_\nu(x)$ can be applied to a generalized Fourier transform because they form orthogonal basis sets under certain given conditions: for example,

$$\int_0^a J_\nu(\kappa_{\nu k}\rho) J_\nu(\kappa_{\nu l}\rho) \rho\, d\rho = \frac{a^2}{2} J_{\nu+1}^2(x_{\nu k}) \delta_{kl}, \tag{6.96}$$

where $\kappa_{\nu k}$ and $x_{\nu k} = \kappa_{\nu k} a$ are from the kth zero (root) of $J_\nu(x) = 0$. Then for a function $f(\rho)$ defined in the region $\rho \in [0, a]$, we have

$$f(\rho) = \sum_{k=1}^{\infty} A_{\nu k} J_\nu(\kappa_{\nu k}\rho), \tag{6.97}$$

with

$$A_{\nu k} = \frac{2}{a^2 J_{\nu+1}^2(x_{\nu k})} \int_0^a f(\rho) J_\nu(\kappa_{\nu k}\rho)\rho \, d\rho. \tag{6.98}$$

In practice, there are two more problems in generating Bessel functions numerically. Bessel functions have an infinite number of terms in the series representation, so it is difficult to initiate the recursion numerically, and the Bessel function of the second kind, $Y_\nu(x)$, increases exponentially with ν when $\nu > x$.

These problems can be resolved if we carry out the recursion forward for $Y_\nu(x)$ and backward for $J_\nu(x)$. We can use several properties of the functions to initiate the recursion. For $J_\nu(x)$, we can set the first two points as $J_N(x) = 0$ and $J_{N-1}(x) = 1$ in the backward recursion. The functions generated can be rescaled afterward with

$$R_N = J_0(x) + 2J_2(x) + \cdots + 2J_N(x), \tag{6.99}$$

because $R_\infty = 1$ for the actual Bessel functions and

$$\lim_{\nu\to\infty} J_\nu(x) = 0. \tag{6.100}$$

For $Y_\nu(x)$, we can use the values obtained for $J_0(x), J_1(x), \ldots, J_N(x)$ to initiate the first two points,

$$Y_0(x) = \frac{2}{\pi}\left(\ln\frac{x}{2} + \gamma\right) J_0(x) - \frac{4}{\pi}\sum_{k=1}^{\infty}(-1)^k \frac{J_{2k}(x)}{k} \tag{6.101}$$

and

$$Y_1(x) = \frac{1}{J_0(x)}\left[J_1(x)Y_0(x) - \frac{2}{\pi x}\right], \tag{6.102}$$

in the forward recursion. Here γ is the Euler constant,

$$\gamma = \lim_{N\to\infty}\left(\sum_{k=1}^{N}\frac{1}{k} - \ln N\right) = 0.577\,215\,664\,9\ldots. \tag{6.103}$$

The summation in Eq. (6.101) is truncated at $2k = N$, which should not introduce too much error into the functions, because $J_\nu(x)$ exponentially decreases with ν for $\nu > x$. The following method is an implementation of the above schemes for generating Bessel functions of the first and second kinds.

```
// Method to create Bessel functions for a given x, index n,
// and index buffer nb.

  public static double[][] b(double x, int n, int nb) {
    int nmax = n+nb;
    double g = 0.5772156649;
    double y[][] = new double[2][n+1];
    double z[] = new double[nmax+1];

// Generate the Bessel function of 1st kind J_n(x)
    z[nmax-1] = 1;
    double s = 0;
    for (int i=nmax-1; i>0; --i) {
      z[i-1] = 2*i*z[i]/x-z[i+1];
      if (i%2 == 0) s += 2*z[i];
    }
    s += z[0];
    for (int i=0; i<=n; ++i) y[0][i] = z[i]/s;

// Generate the Bessel function of 2nd kind Y_n(x)
    double t = 0;
    int sign = -1;
    for (int i=1; i<nmax/2; ++i) {
      t += sign*z[2*i]/i;
      sign *= -1;
    }
    t *= -4/(Math.PI*s);
    y[1][0] = 2*(Math.log(x/2)+g)*y[0][0]/Math.PI+t;
    y[1][1] = (y[0][1]*y[1][0]-2/(Math.PI*x))/y[0][0];
    for (int i=1; i<n; ++i)
      y[1][i+1] = 2*i*y[1][i]/x-y[1][i-1];

    return y;
  }
```

If we only need to generate $J_\nu(x)$, the lines associated with $Y_\nu(x)$ can be deleted from the above method. Note that the variable x needs to be smaller than the maximum ν in order for the value of $J_\nu(x)$ to be accurate.

6.9 Gaussian quadratures

We can use special functions to construct numerical integration quadratures that automatically minimize the possible errors due to the deviation of approximate functions from the data, the same concept used in Chapter 3 to obtain the approximation of a function by orthogonal polynomials. In fact, all special functions form basis sets for certain vector spaces.

An integral defined in the region $[a, b]$ can be written as

$$I = \int_a^b w(x) f(x)\, dx, \tag{6.104}$$

where $w(x)$ is the weight of the integral, which has exactly the same meaning as the weight of the orthogonal functions defined in Chapter 3. Examples of $w(x)$ will be given in this section.

Now we divide the region $[a, b]$ into n points ($n - 1$ slices) and approximate the integral as

$$I \simeq \sum_{k=1}^{n} w_k f(x_k), \tag{6.105}$$

with x_k and w_k determined according to two criteria: the simplicity of the expression and the accuracy of the approximation.

The expression in Eq. (6.105) is quite general and includes the simplest quadratures introduced in Chapter 3 with $w(x) = 1$. For example, if we take

$$x_k = a + \frac{k-1}{n-1}(b-a) \tag{6.106}$$

for $k = 1, 2, \ldots, n$ and

$$w_k = \frac{1}{n(1 + \delta_{k1} + \delta_{kn})}, \tag{6.107}$$

we recover the trapezoid rule. The Simpson rule is recovered if we take a more complicated w_k with

$$w_k = \begin{cases} 1/3n & \text{for } k = 1 \text{ or } n, \\ 1/n & \text{for } k = \text{even numbers}, \\ 2/3n & \text{for } k = \text{odd numbers}, \end{cases} \tag{6.108}$$

but still the same set of x_k with $w(x) = 1$.

The so-called Gaussian quadrature is constructed from a set of orthogonal polynomials $\phi_l(x)$ with

$$\int_a^b \phi_l(x) w(x) \phi_k(x)\, dx = \langle \phi_l | \phi_k \rangle = \mathcal{N}_l \delta_{lk}, \tag{6.109}$$

where the definition of each quantity is exactly the same as in Section 3.2. We can show that to choose x_k to be the kth root of $\phi_n(x) = 0$ and to choose

$$w_k = \frac{-a_n \mathcal{N}_n}{\phi_n'(x_k)\phi_{n+1}(x_k)}, \tag{6.110}$$

with $n = 1, 2, \ldots, N$, we ensure that the error in the quadrature is given by

$$\Delta I = \frac{\mathcal{N}_n}{A_n^2 (2n)!} f^{(2n)}(x_0), \tag{6.111}$$

where x_0 is a value of $x \in [a, b]$, A_n is the coefficient of the x^n term in $\phi_n(x)$, and $a_n = A_{n+1}/A_n$. We can use any kind of orthogonal polynomials for this purpose. For example, with the Legendre polynomials, we have $a = -1, b = 1, w(x) = 1$, $a_n = (2n+1)/(n+1)$, and $\mathcal{N}_n = 2/(2n+1)$.

Another set of very useful orthogonal polynomials is the Chebyshev polynomials, defined in the region $[-1, 1]$. For example, the recursion relation for the Chebyshev polynomials of the first kind is

$$T_{k+1}(x) = 2x\, T_k(x) - T_{k-1}(x), \tag{6.112}$$

starting with $T_0(x) = 1$ and $T_1(x) = x$. We can easily show that

$$\int_{-1}^{1} T_k(x) w(x) T_l(x)\, dx = \delta_{kl}, \tag{6.113}$$

with $w(x) = 1/\sqrt{1-x^2}$. If we write as a series an integral with the same weight,

$$\int_{-1}^{1} \frac{f(x)\,dx}{\sqrt{1-x^2}} \simeq \sum_{k=1}^{n} w_k f(x_k), \tag{6.114}$$

we have

$$x_k = \cos\frac{(2k-1)\pi}{2n} \tag{6.115}$$

and $w_k = \pi/n$, which are extremely easy to use. Note that we can translate a given integration region $[a, b]$ to another, for example, $[0, 1]$ or $[-1, 1]$, through a linear coordinate transformation.

Exercises

6.1 Run the discrete Fourier transform program and the fast Fourier transform program on a computer and establish the time dependence on the number of points for both of them. Would it be different if the processor were a vector processor, that is, if a segment of the inner loop were performed in one clock cycle?

6.2 Develop a method that implements the fast Fourier transform for N data points with $N = 4^M$, where M is an integer. Is this algorithm faster that the one with $N = 2^M$ given in Section 6.3 when applied to the same data stream?

6.3 Analyze the power spectrum of the Duffing model defined in Exercise 4.6. What is the most significant difference between the Duffing model and the driven pendulum with damping?

6.4 Develop a method that carries out the continuous wavelet transform with the Haar wavelet. Apply the method to the angular data in the driven pendulum with damping. Illustrate the wavelet coefficients in a surface plot. What can we learn from the wavelet coefficients at different parameter regions?

6.5 Show that the real-value Morlet wavelet

$$u(t) = e^{-t^2} \cos\left(\pi\sqrt{\frac{2}{\ln 2}}\,t\right)$$

satisfies the admissibility condition roughly. Apply this wavelet in a continuous wavelet transform to the time-dependent function $f(t) = 1$ for $t \in [0, 1]$, zero otherwise. Plot the scalogram of $f(t)$ and discuss the features shown.

6.6 Show that the wavelet

$$u(t) = e^{-t^2}(1 - 2t^2)$$

satisfies the admissibility condition. Apply this wavelet in a continuous wavelet transform to a sequence $f_i = f(t_i)$ from a uniform random-number generator with the assumption that t_i are uniformly spaced in the region [0, 1]. Does the scalogram show any significant structures?

6.7 Solve the equation set outlined in Section 6.7 for the scaling filters for $n = 4, 8, 16$. Is the result for the $n = 4$ the same as the D4 wavelet?

6.8 Find the zeros of the polynomial given in Eq. (6.68) inside the unit circle, and then use Eq. (6.69) to obtain the scaling filters for $n = 4, 8, 16$. Is the result for the $n = 4$ case the same as the D4 wavelet?

6.9 Develop a method that performs the pyramid algorithm in the discrete wavelet transform under the D4 wavelet. Apply the method to the displacement data in the Duffing model. What can we learn from both the wavelet and scaling coefficients at different parameter regions?

6.10 Based on the inverse pyramid algorithm, write a method that recovers the original data from the wavelet and scaling coefficients. Test the method with a simple data stream generated from a random-number generator.

6.11 Write a method that generates the Chebyshev polynomials of the first kind discussed in Section 6.9.

6.12 The Laguerre polynomials form an orthogonal basis set in the region $[0, \infty]$ and satisfy the following recursion

$$(n+1)L_{n+1}(x) = (2n+1-x)L_n(x) - nL_{n-1}(x),$$

starting with $L_0(x) = 1$ and $L_1(x) = 1 - x$. (a) Write a subroutine that generates the Laguerre polynomials for a given x and maximum n. (b) Show that the weight of the polynomials $w(x) = e^{-x}$, and that the normalization factor $\mathcal{N}_n = \int_0^\infty w(x)L_n^2(x)dx = 1$. (c) If we want to construct the Gaussian quadrature for the integral

$$I = \int_0^\infty e^{-x} f(x)\,dx = \sum_{n=1}^{N} w_n f(x_n) + \Delta I,$$

where x_n is the nth root of $L_N(x) = 0$, show that w_n is given by

$$w_n = \frac{(N!)^2 x_n}{(N+1)^2 L_{N+1}^2(x_n)}.$$

This quadrature is extremely useful in statistical physics, where the integrals are typically weighted with an exponential function.

6.13 The Hermite polynomials are another set of important orthogonal polynomials in the region $[-\infty, \infty]$ and satisfy the following recursion

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x),$$

starting with $H_0(x) = 1$ and $H_1(x) = 2x$. (a) Write a subroutine that generates the Hermite polynomials for a given x and maximum n. (b) Show that the weight of the polynomials $w(x) = e^{-x^2}$ and that the normalization factor $\mathcal{N}_n = \int_{-\infty}^{\infty} w(x)H_n^2(x)dx = \sqrt{\pi}2^n n!$. (c) Now if we want to construct the Gaussian quadrature for the integral

$$I = \int_{\infty}^{\infty} e^{-x^2} f(x)\,dx = \sum_{n=1}^{N} w_n f(x_n) + \Delta I,$$

where x_n is the nth root of $H_N(x) = 0$, show that w_n is given by

$$w_n = \frac{2^{N-1} N! \sqrt{\pi}}{N^2 H_{N-1}^2(x_n)}.$$

Many integrals involving the Gaussian distribution can take advantage of this quadrature.

6.14 The integral expressions for the Bessel functions of the first and second kinds of order zero are given by

$$J_0(x) = \frac{2}{\pi} \int_0^{\infty} \sin(x \cosh t)\,dt,$$

$$Y_0(x) = -\frac{2}{\pi} \int_0^{\infty} \cos(x \cosh t)\,dt,$$

with $x > 0$. Calculate these two expressions numerically and compare them with the values generated with the subroutine in Section 6.8.

6.15 Demonstrate numerically that for a small angle θ but large l,

$$P_l(\cos\theta) \simeq J_0(l\theta).$$

6.16 In quantum scattering, when the incident particle has very high kinetic energy, the cross section

$$\sigma = 2\pi \int_0^{\pi} \sin\theta |f(\theta)|^2\,d\theta$$

has a very simple form with

$$f(\theta) \simeq -\frac{2m}{\hbar^2} \int_0^{\infty} r V(x) \sin(qr)\,dr,$$

where $q = |\mathbf{k}_f - \mathbf{k}_i| = 2k \sin(\theta/2)$ and $\mathbf{k}_i$ and $\mathbf{k}_f$ are initial and final momenta of the particle with the same magnitude k. (a) Show that the above $f(\theta)$ is the dominant term as $k \to \infty$. (b) If the particle is an electron and the scattering potential is an ionic potential

$$V(r) = \frac{1}{4\pi\epsilon_0} \frac{Ze^2}{r} e^{-r/r_0},$$

write a program to evaluate the cross section of the scattering with the Gaussian quadrature from the Laguerre polynomials for the integrals. Here Z, e, and r_0 are the number of protons, the proton charge, and the screening length. Use $Z = 2$ and $r_0 = a_0/4$, where a_0 is the Bohr radius, as a testing example.

6.17 Another approximation in quantum scattering, called the eikonal approximation, is valid for high-energy and small-angle scattering. The scattering cross section can be written as

$$\sigma = 8\pi \int_0^\infty b \sin^2[\alpha(b)]\, db,$$

with

$$\alpha(b) = -\frac{m}{2\hbar^2 k} \int_{-\infty}^{\infty} V(b, x)\, dx,$$

where b is the impact parameter and x is the coordinate of the particle along the impact direction with the scattering center at $x = 0$. (a) Show that the eikonal approximation is valid for high-energy and small-angle scattering. (b) If the scattering potential is

$$V(r) = -V_0 e^{-(r/r_0)^2},$$

calculate the cross section with the Gaussian quadratures of Hermite and Laguerre polynomials for the integrals. Here V_0 and r_0 are given parameters.

Chapter 7
Partial differential equations

In Chapter 4, we discussed numerical methods for solving initial-value and boundary-value problems given in the form of differential equations with one independent variable, that is, ordinary differential equations. Many of those methods can be generalized to study differential equations involving more than one independent variable, that is, partial differential equations. Many physics problems are given in the form of a second-order partial differential equation, elliptic, parabolic, or hyperbolic.

7.1 Partial differential equations in physics

In this chapter, we discuss numerical schemes for solving partial differential equations. There are several types of partial differential equations in physics. The Poisson equation

$$\nabla^2 \phi(\mathbf{r}) = -\rho(\mathbf{r})/\epsilon_0 \tag{7.1}$$

for the electrostatic potential $\phi(\mathbf{r})$ at the position $\mathbf{r}$ under the given charge distribution $\rho(\mathbf{r})$ is a typical elliptic equation. The diffusion equation

$$\frac{\partial n(\mathbf{r}, t)}{\partial t} - \boldsymbol{\nabla} \cdot D(\mathbf{r}) \boldsymbol{\nabla} n(\mathbf{r}, t) = S(\mathbf{r}, t) \tag{7.2}$$

for the concentration $n(\mathbf{r}, t)$ at the position $\mathbf{r}$ and time t under the given source $S(\mathbf{r}, t)$ is a typical parabolic equation. Here $D(\mathbf{r})$ is the diffusion coefficient at the position $\mathbf{r}$. The wave equation

$$\frac{1}{c^2} \frac{\partial^2 u(\mathbf{r}, t)}{\partial t^2} - \nabla^2 u(\mathbf{r}, t) = R(\mathbf{r}, t) \tag{7.3}$$

for the generalized displacement $u(\mathbf{r}, t)$ at the position $\mathbf{r}$ and time t under the given source $R(\mathbf{r}, t)$ is a typical hyperbolic equation. The time-dependent Schrödinger equation

$$-\frac{\hbar}{i} \frac{\partial \Psi(\mathbf{r}, t)}{\partial t} = \mathcal{H} \Psi(\mathbf{r}, t) \tag{7.4}$$

for the wavefunction $\Psi(\mathbf{r}, t)$ of a quantum system defined by the Hamiltonian $\mathcal{H}$, can be viewed as a diffusion equation under imaginary time.

All the above equations are linear if the sources and other quantities in the equations are not related to the solutions. There are also nonlinear equations in physics that rely heavily on numerical solutions. For example, the equations for fluid dynamics,

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} + \frac{1}{\rho} \nabla P - \eta \nabla^2 \mathbf{v} = 0, \tag{7.5}$$

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \mathbf{v} = 0, \tag{7.6}$$

$$f(P, \rho) = 0, \tag{7.7}$$

require numerical solutions under most conditions. Here the first equation is the Navier–Stokes equation, in which $\mathbf{v}$ is the velocity, ρ the density, P the pressure, and η the kinetic viscosity of the fluid. The Navier–Stokes equation can be derived from the Newton equation for a small element in the fluid. The second equation is the continuity equation, which is the result of mass conservation. The third equation is the equation of state, which can also involve temperature as an additional variable in many cases. We will cover numerical solutions of the hydrodynamical equations in Chapter 9.

In this chapter, we will first outline an analytic scheme that can simplify numerical tasks drastically in many cases and is commonly known as the separation of variables. The basic idea of the separation of variables is to reduce a partial differential equation to several ordinary differential equations. The separation of variables is the standard method in analytic solutions of most partial differential equations. Even in the numerical study of partial differential equations, due to the limitation of computing resources, the separation of variables can serve the purpose of simplifying a problem to the point where it can be solved with the available computing resources. Later in the chapter, we will introduce several numerical schemes used mainly in solving linear partial differential equations.

7.2 Separation of variables

Before we introduce numerical schemes for partial differential equations, we now discuss an analytic method, that is, the *separation of variables*, for solving partial differential equations. In many cases, using a combination of the separation of variables and a numerical scheme increases the speed of computing and the accuracy of the solution. The combination of analytic and numerical methods becomes critical in cases where the memory and speed of the available computing resources are limited. This section is far from being a complete discussion; interested readers should consult a standard textbook, such as Courant and Hilbert (1989).

Here we will just illustrate how the method works. Each variable (time or one of the spatial coordinates) is isolated from the rest, and the solutions of the resulting ordinary differential equations for all the variables are obtained before they are combined into the general solution of the original partial differential equation. Boundary and initial conditions are then used to determine the unknown parameters left in the solution, which usually appear as the coefficients or eigenvalues.

Consider first the standing waves on a light, uniform string that has both ends ($x = 0$ and $x = L$) fixed. The wave equation is

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = 0, \tag{7.8}$$

where $u(x,t)$ is the displacement of the string at the location x and time t, and $c = \sqrt{T/\rho}$ is the phase speed of the wave, with T being the tension on the string and ρ the linear mass density of the string. The boundary condition for fixed ends is $u(0,t) = u(L,t) = 0$ in this case. Now if we assume that the solution is given by

$$u(x,t) = X(x)\Theta(t), \tag{7.9}$$

where $X(x)$ is a function of x only and $\Theta(t)$ is a function of t only, we have

$$\frac{\Theta''(t)}{\Theta(t)} = c^2 \frac{X''(x)}{X(x)} = -\omega^2, \tag{7.10}$$

after we substitute Eq. (7.9) into Eq. (7.8). Note that the introduced parameter (eigenvalue) ω must be independent of the position from the first ratio and independent of the time from the second ratio. Thus we must have

$$X''(x) = -\frac{\omega^2}{c^2} X(x) = -k^2 X(x), \tag{7.11}$$

where $k = \omega/c$ is determined from the boundary condition $X(0) = 0$ and $X(L) = 0$. We can view either ω or k as being the eigenvalue sought for the eigenvalue problem defined in Eq. (7.11) under the given boundary condition. The two independent, analytical solutions of Eq. (7.11) are $\sin kx$ and $\cos kx$. Then we have

$$X(x) = A \sin kx + B \cos kx. \tag{7.12}$$

From the boundary condition $X(0) = 0$, we must have $B = 0$; from $X(L) = 0$, we must have

$$k_n = \frac{n\pi}{L}, \tag{7.13}$$

with $n = 1, 2, \ldots, \infty$. Using this k_n in the equation for $\Theta(t)$, we obtain

$$\Theta(t) = C \sin \omega_n x + D \cos \omega_n x, \tag{7.14}$$

with

$$\omega_n = ck_n = \frac{n\pi c}{L}. \tag{7.15}$$

Combining the solutions for $X(x)$ and $\Theta(t)$, we obtain the general solution

$$u(x,t) = \sum_{n=1}^{\infty} (a_n \sin \omega_n t + b_n \cos \omega_n t) \sin k_n x, \tag{7.16}$$

where a_n and b_n are two sets of parameters that are determined by the initial displacement $u(x, 0) = u_0(x)$ and the initial velocity $\partial u(x, t)/\partial t|_{t=0} = v_0(x)$. Using $t = 0$ for $u(x, t)$ given in Eq. (7.16) and its partial derivative of time, we have

$$u_0(x) = \sum_{n=1}^{\infty} b_n \sin k_n x, \tag{7.17}$$

$$v_0(x) = \sum_{n=1}^{\infty} a_n \omega_n \sin k_n x. \tag{7.18}$$

Then we have

$$b_l = \frac{2}{L} \int_0^L u_0(x) \sin k_l x \, dx, \tag{7.19}$$

$$a_l = \frac{2}{\omega_l L} \int_0^L v_0(x) \sin k_l x \, dx, \tag{7.20}$$

after multiplying Eqs. (7.17) and (7.18) by $\sin k_l x$ and integrating over $[0, L]$. We have also used the orthogonal properties of $\sin k_n x$. This completes the solution of the problem.

What happens if the string is not light, and/or carries a mass density $\rho(x)$ that is not a constant? The separation of variables still works, but we need to solve the eigenvalue problem numerically. Let us take a closer look at the problem. The equation now becomes

$$\frac{\partial^2 u(x, t)}{\partial t^2} = \frac{T}{\rho(x)} \frac{\partial^2 u(x, t)}{\partial x^2} - g, \tag{7.21}$$

where $g = 9.80$ m/s^2 is the magnitude of the gravitational acceleration. This is an inhomogeneous equation. We can solve it with a general solution of the homogeneous equation $u_\mathrm{h}(x, t)$ (with $g = 0$) and a particular solution $u_\mathrm{p}(x, t)$ of the inhomogeneous equation.

First let us consider the solution of the homogeneous equation. We can still separate t from x by taking $u_\mathrm{h}(x, t) = X(x)\Theta(t)$; then we have

$$X''(x) = -\frac{\omega^2 \rho(x)}{T} X(x), \tag{7.22}$$

which can be solved as an eigenvalue problem with using either the scheme developed in Chapter 4 or that in Chapter 5. We will leave it as an exercise for the reader to find the eigenvalues ω_n for the loaded string.

For the particular solution of the inhomogeneous equation, we can simplify the problem by considering the static case, that is, $u_\mathrm{p}(x, t) = u_\mathrm{p}(x)$. Then we have

$$\frac{d^2 u_\mathrm{p}(x)}{dx^2} = \frac{g\rho(x)}{T}, \tag{7.23}$$

which is a linear equation set if we discretize the second-order derivative involved. This equation determines the shape of the string when it is in equilibrium. We will see the similar example of a loaded bench in Section 7.4.

After we obtain the general solution $u_h(x, t)$ of the homogeneous equation and a particular solution of the inhomogeneous equation, for example, $u_p(x)$ from the above static equation, we have the general solution of the inhomogeneous equation as

$$u(x, t) = u_h(x, t) + u_p(x, t). \tag{7.24}$$

The initial displacement $u_0(x)$ and initial velocity $v_0(x)$ are still needed to determined the remaining parameters in the general solution above.

This procedure of separating variables can also be applied to higher-dimensional systems. Let us consider the diffusion equation in an isotropic, three-dimensional, and infinite space, with a time-dependent source and a zero initial value, as an example. The diffusion equation under a time-dependent source and a constant diffusion coefficient is given by

$$\frac{\partial n(\mathbf{r}, t)}{\partial t} - D\nabla^2 n(\mathbf{r}, t) = S(\mathbf{r}, t). \tag{7.25}$$

The initial value $n(\mathbf{r}, 0) = 0$ will be considered first. Note that $t = 0$ in this case can be viewed as the moment of the introduction of the source. Then we can view the source as an infinite number of impulsive sources, each within a time interval $d\tau$, given by $S(\mathbf{r}, \tau)\delta(t - \tau)d\tau$. This way of treating the source is called the *impulse method*. The solution $n(\mathbf{r}, t)$ is then the superposition of the solution in each time interval $d\tau$, with

$$n(\mathbf{r}, t) = \int_0^t \eta(\mathbf{r}, t; \tau)\, d\tau, \tag{7.26}$$

where $\eta(\mathbf{r}, t; \tau)$ satisfies

$$\frac{\partial \eta(\mathbf{r}, t; \tau)}{\partial t} - D\nabla^2 \eta(\mathbf{r}, t; \tau) = S(\mathbf{r}, \tau)\delta(t - \tau), \tag{7.27}$$

with $\eta(\mathbf{r}, 0; \tau) = 0$. The impulse equation above is equivalent to a homogeneous equation

$$\frac{\partial \eta(\mathbf{r}, t; \tau)}{\partial t} - D\nabla^2 \eta(\mathbf{r}, t; \tau) = 0, \tag{7.28}$$

with the initial condition $\eta(\mathbf{r}, \tau; \tau) = S(\mathbf{r}, \tau)$. Thus we have transformed the original inhomogeneous equation with a zero initial value into an integral of a function that is a solution of a homogeneous equation with a nonzero initial value. We can generally write the solution of the inhomogeneous equation as the sum of a solution of the corresponding homogeneous equation and a particular solution of the inhomogeneous equation with the given initial condition satisfied. So if we find a way to solve the corresponding homogeneous equation with a nonzero initial value, we find the solution of the general problem.

Now let us turn to the separation of variables for a homogeneous diffusion equation with a nonzero initial value. We will assume that the space is an isotropic, three-dimensional, and infinite space, in order to simplify our discussion. Other finite boundary situations can be solved in a similar manner. We will also suppress the parameter τ to simplify our notation. The solution of $\eta(\mathbf{r}, t) = \eta(\mathbf{r}, t; \tau)$ can be assumed to be

$$\eta(\mathbf{r}, t) = X(\mathbf{r})\Theta(t), \tag{7.29}$$

where $X(\mathbf{r})$ is a function of $\mathbf{r}$ only and $\Theta(t)$ is a function of t only. If we substitute this assumed solution into the equation, we have

$$X(\mathbf{r})\Theta'(t) - D\Theta(t)\nabla^2 X(\mathbf{r}) = 0, \tag{7.30}$$

or

$$\frac{\Theta'(t)}{D\Theta(t)} = \frac{\nabla^2 X(\mathbf{r})}{X(\mathbf{r})} = -k^2, \tag{7.31}$$

where $k = |\mathbf{k}|$ is an introduced parameter (eigenvalue) that will be determined later. Now we have two separate equations,

$$\nabla^2 X(\mathbf{r}) + k^2 X(\mathbf{r}) = 0, \tag{7.32}$$

and

$$\Theta'(t) + \omega\Theta(t) = 0, \tag{7.33}$$

which are related by $\omega = Dk^2$. The spatial equation is now a standard eigenvalue problem with k^2 being the eigenvalue to be determined by the specific boundary condition. Because we have assumed that the space is isotropic, three-dimensional, and infinite, the solution is given by plane waves with

$$X(\mathbf{r}) = Ce^{i\mathbf{k}\cdot\mathbf{r}}, \tag{7.34}$$

where C is a general constant. Similarly, the time-dependent equation for $\Theta(t)$ is a standard initial-value problem with the solution

$$\Theta(t) = Be^{-\omega(t-\tau)}, \tag{7.35}$$

where $\omega = Dk^2$ and B is a coefficient to be determined by the initial condition of the problem. If we combine the spatial solution and the time solution, we have

$$\eta(\mathbf{r}, t) = \frac{1}{(2\pi)^{3/2}} \int A_{\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{r} - \omega(t-\tau)}\, d\mathbf{k}, \tag{7.36}$$

and we can obtain the coefficient $A_{\mathbf{k}}$ from the initial value of $\eta(\mathbf{r}, \tau) = S(\mathbf{r}, \tau)$ with the Fourier transform of the above equation as

$$A_{\mathbf{k}} = \frac{1}{(2\pi)^{3/2}} \int S(\mathbf{r}, \tau) e^{-i\mathbf{k}\cdot\mathbf{r}}\, d\mathbf{r}. \tag{7.37}$$

If we substitute the above result for $A_{\mathbf{k}}$ into the integral for $\eta(\mathbf{r}, t)$ and complete the integration over $\mathbf{k}$, we obtain

$$\eta(\mathbf{r}, t) = \frac{1}{\sqrt{4\pi Dt}} \int S(\mathbf{r}', \tau) e^{-(\mathbf{r}-\mathbf{r}')^2/4Dt}\, d\mathbf{r}', \tag{7.38}$$

which can be substituted into the integral expression of $n(\mathbf{r}, t)$. We then reach the final solution of the original problem,

$$n(\mathbf{r}, t) = \int_0^t \frac{d\tau}{\sqrt{4\pi D(t-\tau)}} \int S(\mathbf{r}', \tau) e^{-(\mathbf{r}-\mathbf{r}')^2/4D(t-\tau)}\, d\mathbf{r}'. \tag{7.39}$$

This integral can be evaluated numerically if the form of $S(\mathbf{r}, t)$ is given. We should realize that if the space is not infinite but confined by some boundary, the above closed form of the solution will be in a different but similar form, because the idea of the Fourier transform is quite general in the sense of the spatial eigenstates. In most cases, the spatial eigenstates form an orthogonal basis set which can be used for a general Fourier transform, as discussed in Chapter 6. It is worth pointing out that the above procedure is also similar to the Green's function method, that is, transforming the problem into a Green's function problem and expressing the solution as a convolution integral of Green's function and the source. For more discussions on the Green's function method, see Courant and Hilbert (1989).

For the wave equation with a source that varies over time and space, we can follow more or less the same steps to complete the separation of variables. The general solution for a nonzero initial displacement and/or velocity is a linear combination of the solution $u_{\rm h}(\mathbf{r}, t)$ of the homogeneous equation and a particular solution $u_{\rm p}(\mathbf{r}, t)$ of the inhomogeneous equation. For the homogeneous equation

$$\nabla^2 u_{\rm h}(\mathbf{r}, t) = \frac{1}{c^2}\frac{\partial^2 u_{\rm h}(\mathbf{r}, t)}{\partial t^2}, \tag{7.40}$$

we can assume that $u_{\rm h}(\mathbf{r}, t) = X(\mathbf{r})\Theta(t)$, and then we have $\Theta''(t) = -\omega_{\mathbf{k}}^2\Theta(t)$ and $\nabla^2 X(\mathbf{r}) = -k^2 X(\mathbf{r})$, with $\omega_{\mathbf{k}}^2 = c^2k^2$. If we solve the eigenvalue problem for the spatial part, we obtain the general solution

$$u_{\rm h}(\mathbf{r}, t) = \sum_{\mathbf{k}} \left(A_{\mathbf{k}} e^{-i\omega_{\mathbf{k}} t} + B_{\mathbf{k}} e^{i\omega_{\mathbf{k}} t}\right) u_{\mathbf{k}}(\mathbf{r}), \tag{7.41}$$

where $u_{\mathbf{k}}(\mathbf{r})$ is the eigenstate of the equation for $X(\mathbf{r})$ with the eigenvalue $\mathbf{k}$. The particular solution $u_{\rm p}(\mathbf{r}, t)$ can be obtained by performing a Fourier transform of time on both sides of Eq. (7.3). Then we have

$$\nabla^2 \tilde{u}_{\rm p}(\mathbf{r}, \omega) + \frac{\omega^2}{c^2}\tilde{u}_{\rm p}(\mathbf{r}, \omega) = -\tilde{R}(\mathbf{r}, \omega), \tag{7.42}$$

where $\tilde{u}_{\rm p}(\mathbf{r}, \omega)$ and $\tilde{R}(\mathbf{r}, \omega)$ are the Fourier transforms of $u_{\rm p}(\mathbf{r}, t)$ and $R(\mathbf{r}, t)$, respectively. We can expand $\tilde{u}_{\rm p}(\mathbf{r}, \omega)$ and $\tilde{R}(\mathbf{r}, \omega)$ in terms of the eigenstates $u_{\mathbf{k}}(\mathbf{r})$ as

$$\tilde{u}_{\rm p}(\mathbf{r}, \omega) = \sum_{\mathbf{k}} c_{\mathbf{k}} u_{\mathbf{k}}(\mathbf{r}) \tag{7.43}$$

and

$$\tilde{R}(\mathbf{r}, \omega) = \sum_{\mathbf{k}} d_{\mathbf{k}} u_{\mathbf{k}}(\mathbf{r}), \tag{7.44}$$

which give

$$c_{\mathbf{k}} = \frac{c^2 d_{\mathbf{k}}}{\omega_{\mathbf{k}}^2 - \omega^2} \tag{7.45}$$

from Eq. (7.42). If we put all these together, we obtain the general solution for the inhomogeneous equation with

$$u(\mathbf{r}, t) = u_{\mathrm{h}}(\mathbf{r}, t) + \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \left(\sum_{\mathbf{k}} \frac{d_{\mathbf{k}}}{k^2 - \omega^2/c^2} \right) e^{-i\omega t}\, d\omega, \tag{7.46}$$

where $d_{\mathbf{k}}$ is given by

$$d_{\mathbf{k}} = \int \tilde{R}(\mathbf{r}, \omega) u_{\mathbf{k}}^*(\mathbf{r})\, d\mathbf{r}, \tag{7.47}$$

which is a function of ω. We have assumed that $u_{\mathbf{k}}(\mathbf{r})$ are normalized and form an orthogonal basis set. Here $u_{\mathbf{k}}^*(\mathbf{r})$ is the complex conjugate of $u_{\mathbf{k}}(\mathbf{r})$. The initial condition can now be used to determine $A_{\mathbf{k}}$ and $B_{\mathbf{k}}$.

A useful result of the separation of variables is that we may then need to deal with an equation or equations involving only a single variable, so that all the methods we discussed in Chapter 4 become applicable.

7.3 Discretization of the equation

The essence of all numerical schemes for solving differential equations lies in the discretization of the continuous variables, that is, the spatial coordinates and time. If we use the rectangular coordinate system, we have to discretize $\partial A(\mathbf{r}, t)/\partial r_i$, $\partial^2 A(\mathbf{r}, t)/\partial r_i \partial r_j$, $\partial A(\mathbf{r}, t)/\partial t$, and $\partial^2 A(\mathbf{r}, t)/\partial^2 t$, where r_i or r_j is either x, y, or z. Sometimes we also need to deal with a situation similar to having a spatially dependent diffusion constant, for example, to discretize $\boldsymbol{\nabla} \cdot D(\mathbf{r}) \boldsymbol{\nabla} A(\mathbf{r}, t)$. Here $A(\mathbf{r}, t)$ is a generic function from one of the equations discussed at the beginning of this chapter. We can apply the numerical schemes developed in Chapter 3 for first-order and second-order derivatives for all the partial derivatives. Typically, we use the two-point or three-point formula for the first-order partial derivative, that is,

$$\frac{\partial A(\mathbf{r}, t)}{\partial t} = \frac{A(\mathbf{r}, t_{k+1}) - A(\mathbf{r}, t_k)}{\tau} \tag{7.48}$$

or

$$\frac{\partial A(\mathbf{r}, t)}{\partial t} = \frac{A(\mathbf{r}, t_{k+1}) - A(\mathbf{r}, t_{k-1})}{2\tau}. \tag{7.49}$$

We can also use the three-point formula

$$\frac{\partial^2 A(\mathbf{r}, t)}{\partial^2 t} = \frac{A(\mathbf{r}, t_{k+1}) - 2A(\mathbf{r}, t_k) + A(\mathbf{r}, t_{k-1})}{\tau^2} \tag{7.50}$$

for the second-order derivative. Here $t_k = t$ and $\tau = t_{k+1} - t_k = t_k - t_{k-1}$. The same formulas can be applied to the spatial variables as well, for example,

$$\frac{\partial A(\mathbf{r}, t)}{\partial x} = \frac{A(x_{k+1}, y, z, t) - A(x_{k-1}, y, z, t)}{2h_x}, \tag{7.51}$$

with $h_x = x_{k+1} - x_k = x_k - x_{k-1}$, and

$$\frac{\partial^2 A(\mathbf{r}, t)}{\partial^2 x} = \frac{A(x_{k+1}, y, z, t) - 2A(x_k, y, z, t) + A(x_{k-1}, y, z, t)}{h_x^2}. \tag{7.52}$$

The partial differential equations can then be solved at discrete points.

However, when we need to deal with a discrete mesh that is not uniform or with an inhomogeneity such as $\nabla \cdot D(\mathbf{r})\nabla A(\mathbf{r}, t)$, we may need to introduce some other discretization scheme. Typically, we can construct a functional from the field of the solution in an integral form. The differential equation is recovered from functional variation. This is in the same spirit as deriving the Lagrange equation from the optimization of the action integral. The advantage here is that we can discretize the integral first and then carry out the functional variation at the lattice points. The corresponding difference equation results. Let us take the one-dimensional Poisson equation

$$\frac{d}{dx}\epsilon(x)\frac{d\phi(x)}{dx} = -\rho(x), \tag{7.53}$$

for $x \in [0, L]$, as an illustrative example. Here $\epsilon(x)$ is the electric permittivity that has a spatial dependence.

For simplicity, let us take the homogeneous Dirichlet boundary condition $\phi(0) = \phi(L) = 0$. We can construct a functional

$$U = \int_0^L \left\{ \frac{1}{2}\epsilon(x)\left[\frac{d\phi(x)}{dx}\right]^2 - \rho(x)\phi(x) \right\} dx, \tag{7.54}$$

which leads to the Poisson equation if we take

$$\frac{\delta U}{\delta \phi} = 0. \tag{7.55}$$

If we use the three-point formula for the first-order derivative in the integral and the trapezoid rule to convert the integral into a summation, we have

$$U \simeq \frac{1}{2h}\sum_k \epsilon_{k-1/2}(\phi_k - \phi_{k-1})^2 - h\sum_k \rho_k\phi_k \tag{7.56}$$

as the discrete representation of the functional. We have used $\phi_k = \phi(x_k)$ for notational convenience and have used $\epsilon_{k-1/2} = \epsilon(x_{k-1/2})$ as the value of $\epsilon(x)$ in the middle of the interval $[x_{k-1}, x_k]$. Now if we treat each discrete variable ϕ_k as an independent variable, the functional variation in Eq. (7.55) becomes a partial derivative

$$\frac{\partial U}{\partial \phi_k} = 0. \tag{7.57}$$

We then obtain the desired difference equation

$$\epsilon_{k+1/2}\phi_{k+1} - (\epsilon_{k+1/2} + \epsilon_{k-1/2})\phi_k + \epsilon_{k-1/2}\phi_{k-1} = -h^2\rho_k, \tag{7.58}$$

which is a discrete representation of the original Poisson equation. This scheme is extremely useful when the geometry of the system is not rectangular or when the coefficients in the equation are not constants. For example, we may want to study the Poisson equation in a cylindrically or spherically symmetric case, or diffusion of some contaminated materials underground where we have a spatially dependent diffusion coefficient. The stationary one-dimensional diffusion equation is equivalent to the one-dimensional Poisson equation if we replace $\epsilon(x)$ with $D(x)$ and $\rho(x)$ with $S(x)$ in Eqs. (7.53)–(7.58). The scheme can also be generalized for the time-dependent diffusion equation by replacing the source term $h^2 S_k$ with $h^2\{S_k(t_i) - [n_k(t_{i+1}) - n_k(t_i)]/\tau\}$. For higher spatial dimensions, the integral is a multidimensional integral. The aspects of time dependence and higher dimensions will be discussed in more detail later in this chapter.

Sometimes it is more convenient to deal with the physical quantities only at the lattice points. Equation (7.58) can be modified to such a form, with the quantity midway between two neighboring points replaced by the average of two lattice points, for example, $\epsilon_{k+1/2} \simeq (\epsilon_k + \epsilon_{k+1})/2$. The difference equation given in Eq. (7.58) can then be modified to

$$(\epsilon_{k+1} + \epsilon_k)\phi_{k+1} - 4\epsilon_k\phi_k + (\epsilon_{k-1} + \epsilon_k)\phi_{k-1} = -2h^2\rho_k, \tag{7.59}$$

which does not sacrifice too much accuracy if the permittivity $\epsilon(x)$ is a slowly-varying function of x.

7.4 The matrix method for difference equations

When a partial differential equation is discretized and given in a difference equation form, we can generally solve it with the matrix methods that we introduced in Chapter 5. Recall that we outlined how to solve the Hartree–Fock equation for atomic systems with the matrix method in Section 5.8 as a special example. In general, when we have a differential equation with the form

$$\mathcal{L}u(\mathbf{r}, t) = f(\mathbf{r}, t), \tag{7.60}$$

where $\mathcal{L}$ is a linear differential operator of the spatial and time variables, $u(\mathbf{r}, t)$ is the physical quantity to be solved, and $f(\mathbf{r}, t)$ is the source, we can always discretize the equation and put it into the matrix form

$$\mathbf{Au} = \mathbf{b}, \tag{7.61}$$

where the coefficient matrix $\mathbf{A}$ is from the discretization of the operator $\mathcal{L}$, the column vector $\mathbf{u}$ is from the discrete values of $u(\mathbf{r}, t)$ excluding the boundary and initial points, and $\mathbf{b}$ is the known vector constructed from the discrete values of $f(\mathbf{r}, t)$ and the boundary and initial points of $u(\mathbf{r}, t)$. The time variable is usually separated with the Fourier transform first unless it has to be dealt with at different spatial points. Situations like this will be discussed in Section 7.7.

For example, the difference equations for the one-dimensional Poisson equation obtained in the preceding section can be cast into such a form. For the difference equation given in Eq. (7.59), we have

$$A_{ij} = \begin{cases} -4\epsilon_i & \text{for } i = j, \\ \epsilon_i + \epsilon_{i+1} & \text{for } i = j - 1, \\ \epsilon_i + \epsilon_{i-1} & \text{for } i = j + 1, \\ 0 & \text{otherwise,} \end{cases} \tag{7.62}$$

and

$$b_i = -2h^2 \rho_i. \tag{7.63}$$

This matrix representation of the difference equation resulting from the discretization of a differential equation is very general.

Let us illustrate this method further by studying an interesting example. Consider a situation in which a person is sitting at the middle of a long bench supported at both ends. Assume that the length of the bench is L. If we want to know the displacement of the bench at every point, we can establish the equation for the curvature at different locations from Newton's equation for each tiny slice of the bench. Then we obtain

$$YI \frac{d^2u(x)}{dx^2} = f(x), \tag{7.64}$$

where $u(x)$ is the curvature (that is, the inverse of the effective radius of the curve at x), Y is Young's modulus of the bench, $I = t^3w/3$ is a geometric constant, with t being the thickness and w the width of the bench, and $f(x)$ is the force density on the bench. Because both ends are fixed, the curvatures there are taken to be zero: that is, $u(0) = u(L) = 0$. If we discretize the equation with evenly spaced intervals, that is, $x_0 = 0, x_1 = h, \ldots, x_{n+1} = L$, via the three-point formula for the second-order derivative, we have

$$u_{i+1} - 2u_i + u_{i-1} = \frac{h^2 f_i}{YI} \tag{7.65}$$

for $i = 0, 1, \ldots, n + 1$, which is equivalent to

$$\begin{pmatrix} -2 & 1 & \cdots & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \tag{7.66}$$

with $u_0 = u_{n+1} = 0$, as required by the boundary condition, and $b_i = h^2 f_i / YI$. We can easily solve the problem with the Gaussian elimination scheme or the LU decomposition scheme, as discussed in Section 2.4. For the specific problem discussed here, we have $e_i = c_i = 1$ and $d_i = -2$. Assume that the force distribution on the bench is given by

$$f(x) = \begin{cases} -f_0[e^{-(x-L/2)^2/x_0^2} - e^{-1}] - \rho g & \text{for } |x - L/2| \leq x_0, \\ -\rho g & \text{otherwise,} \end{cases} \tag{7.67}$$

with $f_0 = 200$ N/m and $x_0 = 0.25$ m, and that the bench has a length of 3.0 m, a width of 0.20 m, a thickness of 0.030 m, a linear density of $\rho = 3.0$ kg/m, and a Young's modulus of 1.0×10^9 N/m^2. Here $g = 9.80$ m/s^2 is the magnitude of the gravitational acceleration. Note that we have taken a truncated Gaussian form for the weight of the person distributed over the bench.

The following program uses the LU decomposition method introduced in Section 2.4 to solve the bench problem with the force distribution and parameters given above using the SI units.

```
// A program to solve the problem of a person sitting
// on a bench as described in the text.

import java.lang.*;
public class Bench {
  final static int n = 99, m = 2;
  public static void main(String argv[]) {
  double d[] = new double[n];
  double b[] = new double[n];
  double c[] = new double[n];
  double l = 3, l2 = l/2, h = l/(n+1), h2 = h*h;
  double x0 = 0.25, x2 = x0*x0, e0 = 1/Math.E;
  double rho = 3, g = 9.8, f0 = 200;
  double y = 1e9*Math.pow(0.03,3)*0.2/3;

 // Evaluate the coefficient matrix elements
    for (int i=0; i<n; ++i) {
      d[i] = -2;
      c[i] =  1;
      b[i] = -rho*g;
      double x = h*(i+1)-l2;
      if (Math.abs(x) < x0)
        b[i] -= f0*(Math.exp(-x*x/x2)-e0);
      b[i] *= h2/y;
    }

 // Obtain the solution of the curverture of the bench
    double u[] = tridiagonalLinearEq(d, c, c, b);

 // Output the result in every m time steps
    double x = h;
    double mh = m*h;
    for (int i=0; i<n; i+=m) {
      System.out.println(x + " " + 100*u[i]);
      x += mh;
    }
  }

// Method to solve the tridiagonal linear equation set.

  public static double[] tridiagonalLinearEq(double d[],
    double e[], double c[], double b[]) {...}
}
```

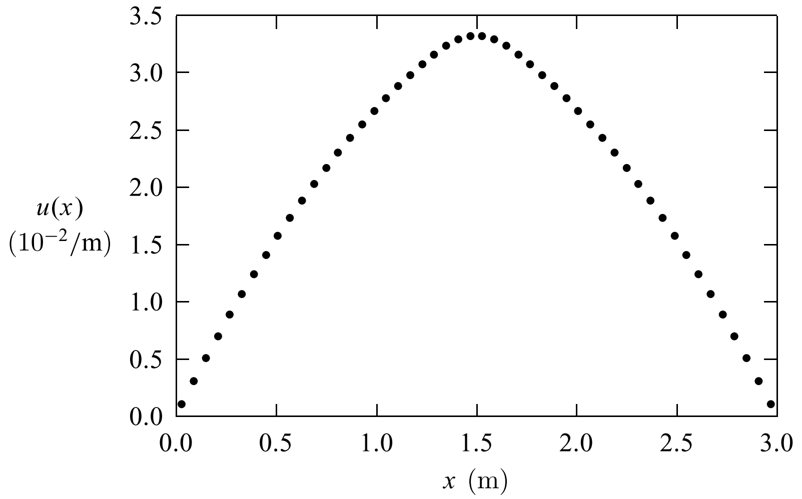


Fig. 7.1 The curvature of the bench, as evaluated with the program given.

The numerical result for the curvature of the bench calculated with the above program is illustrated in Fig. 7.1. Under these realistic parameters, the bench does not deviate from the original position very much. Even at the middle of the bench, the effective radius for the curve is still about 30 m.

A noticeable feature of this problem is that when the equation is discretized into a difference equation, it becomes extremely simple with a symmetric, tridiagonal coefficient matrix. In general, all one-dimensional problems with the same mathematical structure can be solved in the same fashion, such as the one-dimensional Poisson equation or the stationary one-dimensional diffusion equation. We will show later that equations with higher spatial dimensions, or with time dependence, can be solved in a similar manner. We need to note that the boundary condition as well as the coefficient matrix can become more complicated. However, if we split the coefficient matrix among the coordinates correctly, we can still preserve its tridiagonal nature, at least, in each step along each coordinate direction. As long as the coefficient matrix is tridiagonal, we can use the simple LU decomposition outlined in Section 2.4 to obtain the solution of the linear equation set. Sometimes we may also need to solve the linear problem with the full matrix that is no longer tridiagonal. Then the Gaussian elimination or the LU decomposition introduced in Chapter 5 can be used.

7.5 The relaxation method

As we have discussed, a functional can be constructed for the purpose of discretizing a differential equation. The procedure for reaching the differential equation is to optimize the functional. In most cases, the optimization is equivalent to the minimization of the functional. A numerical scheme can therefore be devised to find the numerical solution iteratively, as long as the functional associated with the equation does not increase with each new iteration. The solution of the differential equation is obtained when the functional no longer changes. This numerical scheme, formally known as the *relaxation method*, is extremely powerful for solving elliptic equations, including the Poisson equation, the stationary diffusion equation, and the spatial part of the wave equation after the separation of variables.

Let us first examine the stationary one-dimensional diffusion equation

$$-\frac{d}{dx}\left[D(x)\frac{dn(x)}{dx}\right] = S(x), \tag{7.68}$$

which can be written in a discrete form as

$$n_i = \frac{1}{D_{i+1/2} + D_{i-1/2}}\left(D_{i+1/2}n_{i+1} + D_{i-1/2}n_{i-1} + h^2 S_i\right), \tag{7.69}$$

as discussed in Section 7.3. It is the above equation that gives us the basic idea of the relaxation method: a guessed solution that satisfies the required boundary condition is gradually modified to satisfy the difference equation within the given tolerance. So the key is to establish an updating scheme that can modify the guessed solution gradually toward the correct direction, namely, the direction with the functional minimized. In practice, one uses the following updating scheme

$$n_i^{(k+1)} = (1 - p)n_i^{(k)} + pn_i, \tag{7.70}$$

where $n_i^{(k)}$ is the solution of the kth iteration at the ith lattice point; n_i is given from Eq. (7.69) with the terms on the right-hand side calculated under $n_i^{(k)}$. Here p is an adjustable parameter restricted in the region $p \in [0, 2]$. The reason for such a restriction on p is that the procedure has to ensure the optimization of the functional during the iterations, and this is equivalent to having the solution of Eq. (7.70) approach the true solution of the diffusion equation defined in Eq. (7.68). More discussion on the quantitative aspect of p can be found in Young and Gregory (1988, pp. 1026–39). The points at the boundaries can be updated under the constraints of the boundary condition. Later in this section we will show how to achieve this numerically.

Here let us reexamine the bench problem solved in the preceding section with the LU decomposition. Equation (7.64) is equivalent to Eq. (7.68) with $D(x) = 1$ and $S(x) = -f(x)/(YI)$. The following example program is an implementation of the relaxation scheme for the problem of a person sitting on a bench.

```
// A program to solve the problem of a person sitting
// on a bench with the relaxation scheme.

import java.lang.*;
public class Bench2 {
  final static int n = 100, m = 2;
  public static void main(String argv[]) {
  double u[] = new double[n+1];
  double d[] = new double[n+1];
  double s[] = new double[n+1];
```

```
    double l = 3, l2 = l/2, h = l/n, h2 = h*h;
    double x0 = 0.25, x2 = x0*x0, e0 = 1/Math.E;
    double x = 0, rho = 3, g = 9.8, f0 = 200;
    double y = 1e9*Math.pow(0.03,3)*0.2/3;
    double u0 = 0.032, p = 1.5, del =1e-3;
    int nmax = 100;

  // Evaluate the source in the equation
      for (int i=0; i<=n; ++i) {
        s[i] = rho*g;
        x = h*i-l2;
        if (Math.abs(x) < x0)
          s[i] += f0*(Math.exp(-x*x/x2)-e0);
        s[i] *= h2/y;
      }
      for (int i=1; i<n; ++i) {
        x = Math.PI*h*i/l;
        u[i] = u0*Math.sin(x);
        d[i] = 1;
      }
      d[0] = d[n] = 1;
      relax(u, d, s, p, del, nmax);

  // Output the result in every m time step
      x = 0;
      double mh = m*h;
      for (int i=0; i<n; i+=m) {
        System.out.println(x + " " + 100*u[i]);
        x += mh;
      }
    }

// Method to complete one step of relaxation.

    public static void relax(double u[], double d[],
      double s[], double p, double del, int nmax) {
      int n = u.length-1;
      double q = 1-p, fi = 0;
      double du = 2*del;
      int k = 0;

      while ((du>del) && (k<nmax)) {
        du = 0;
        for (int i=1; i<n; ++i) {
          fi = u[i];
          u[i] = p*u[i]
                +q*((d[i+1]+d[i])*u[i+1]
                +(d[i]+d[i-1])*u[i-1]+2*s[i])/(4*d[i]);
          fi = u[i]-fi;
          du += fi*fi;
        }
        du = Math.sqrt(du/n);
        k++;
      }
      if (k==nmax) System.out.println("Convergence not" +
        " found after " + nmax + " iterations");
    }
}
```

Note that in the above method we have used the updated u_{i-1} on the right-hand side of the relaxation scheme, which is usually more efficient but less stable. We have also used $D_{i-1/2} \simeq (D_i + D_{i-1})/2$, $D_{i+1/2} \simeq (D_i + D_{i+1})/2$, and $D_{i+1/2} + D_{i-1/2} \simeq 2D_i$ in Eq. (7.69) in case the diffusion coefficient is given at the lattice points only. The multiplication of $S(x)$ by h^2 is done outside the method to keep the scheme more efficient. In general, the solution with the LU decomposition is faster and stabler than the solution with the relaxation method for the bench problem. But the situation reverses in the case of a higher-dimensional system.

The points at the boundaries are not updated in the method. For the Dirichlet boundary condition, we can just keep them constant. For the Neumann boundary condition, we can use either a four-point or a six-point formula for the first-order derivative to update the solution at the boundary. For example, if we have

$$\left.\frac{dn(x,t)}{dx}\right|_{x=0} = 0, \tag{7.71}$$

we can update the first point, n_0, by setting

$$n_0 = \frac{1}{3}(4n_1 - n_2), \tag{7.72}$$

which is the result of the four-point formula of the first-order derivative at $x = 0$,

$$\left.\frac{dn(x)}{dx}\right|_{x=0} \simeq \frac{1}{6h}(-2n_{-1} - 3n_0 + 6n_1 - n_2) = 0, \tag{7.73}$$

where $n_{-1} = n_1$ is the result of the zero derivative. A partial derivative is dealt with in the same manner. Higher accuracy can be achieved if we use a formula with more points. We have to use even-numbered point formulas because we want to have n_0 in the expression.

Now let us turn to the two-dimensional case, and we will use the Poisson equation

$$\nabla^2\phi(\mathbf{r}) = -\rho(\mathbf{r})/\epsilon_0 = -s(\mathbf{r}), \tag{7.74}$$

as an illustrative example. Here $s(\mathbf{r})$ is introduced for convenience. Now if we consider the case with a rectangular boundary, we have

$$\frac{\phi_{i+1j} + \phi_{i-1j} - 2\phi_{ij}}{h_x^2} + \frac{\phi_{ij+1} + \phi_{ij-1} - 2\phi_{ij}}{h_y^2} = -s_{ij}, \tag{7.75}$$

where i and j are used for the x and y coordinates and h_x and h_y are the intervals along the x and y directions, respectively. We can rearrange the above equation so that the value of the solution at a specific point is given by the corresponding values at the neighboring points,

$$\phi_{ij} = \frac{1}{2(1+\alpha)}\left[\phi_{i+1j} + \phi_{i-1j} + \alpha(\phi_{ij+1} + \phi_{ij-1}) + h_x^2 s_{ij}\right], \tag{7.76}$$

with $\alpha = (h_x/h_y)^2$. So the solution is obtained if the values of ϕ_{ij} at all the lattice points satisfy the above equation and the boundary condition. The relaxation scheme is based on the fact that the solution of the equation is approached iteratively. We first guess a solution that satisfies the boundary condition. Then we can update or improve the guessed solution with

$$\phi_{ij}^{(k+1)} = (1-p)\phi_{ij}^{(k)} + p\phi_{ij}, \tag{7.77}$$

where p is an adjustable parameter close to 1. Here $\phi_{ij}^{(k)}$ is the result of the kth iteration, and ϕ_{ij} is obtained from Eq. (7.76) with $\phi_{ij}^{(k)}$ used on the right-hand side.

The second part of the above equation can be viewed as the correction to the solution, because it is obtained through the differential equation. The choice of p determines the speed of convergence. If p is selected outside the allowed range by the specific geometry and discretization, the algorithm will be unstable. Usually, the optimized p is somewhere between 0 and 2. In practice, we can find the optimized p easily by just running the program for a few iterations, say, ten, with different values of p. The convergence can be analyzed easily with the result from iterations of each choice of p. Mathematically, one can place an upper limit on p but in practice, this is not really necessary, because it is much easier to test the choice of p numerically.

7.6 Groundwater dynamics

Groundwater dynamics is very rich because of the complexity of the underground structures. For example, a large piece of rock may modify the speed and direction of flow drastically. The dynamics of groundwater is of importance in the construction of any underground structure.

In this section, because we just want to illustrate the power of the method, we will confine ourselves to the relatively simple case of a two-dimensional aquifer with a rectangular geometry of dimensions $L_x \times L_y$. Readers interested in learning more about the subject can find detailed discussions on groundwater modeling in Wang and Anderson (1982), Konikow and Reilly (1999), or Charbeneau (2000).

Steady groundwater flow is described by the so-called Darcy's law

$$\mathbf{q} = -\boldsymbol{\sigma} \cdot \nabla\phi, \tag{7.78}$$

where $\mathbf{q}$ is the *specific discharge* vector (the flux density), which is a measure of the volume of the fluid passing through a unit cross-sectional area perpendicular to the velocity of the flow in a unit of time. The average velocity $\mathbf{v}$ of the flow at a given position and time can be related to the specific discharge at the same position and time by $\mathbf{v} = \mathbf{q}/\beta$, where β is the *porosity*, which is the percentage of the empty space (voids) on a cross section perpendicular to the flow. Here $\boldsymbol{\sigma}$ is the *hydraulic conductivity* tensor of rank 2 (or 3×3 matrix); it has nine elements given by the specified porous medium and carries a unit of velocity. For example, for an isotropic medium, $\boldsymbol{\sigma}$ reduces to a diagonal matrix with three identical elements σ, and Darcy's law is simplified to $\mathbf{q} = -\sigma \boldsymbol{\nabla}\phi$. The scalar field ϕ is referred to as the *head*, which is a measure of relative energy of the water in a standpipe from the datum (zero point). The head at elevation z, measured from the datum, can be related to the hydraulic pressure P and speed of the flow at the same elevation as

$$\phi \simeq \frac{v^2}{2g} + \frac{P}{\rho g} + z \simeq \frac{P}{\rho g} + z, \tag{7.79}$$

where ρ is the density of the fluid, g is the magnitude of the gravitational acceleration, and $v \simeq 0$ is the speed of the flow. Note that Darcy's law here is analogous to Ohm's law for electric conduction or Fourier's law for thermal conduction. For an ideal fluid under the steady flow condition, $\rho g \phi$ is a constant, a consequence of Bernoulli's equation.

If we combine the above equation with the continuity equation resulting from mass conservation (volume conservation if ρ is constant), we have

$$\mu \frac{\partial \phi(\mathbf{r}, t)}{\partial t} - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} \cdot \boldsymbol{\nabla}\phi(\mathbf{r}, t) = f(\mathbf{r}, t), \tag{7.80}$$

where μ is the *specific storage*, a measure of the influence on the rate of head change, and f is the rate of *infiltration*. For an isotropic, steady flow, the above equation reduces to a generalized Poisson equation

$$\nabla^2 \phi(\mathbf{r}) = -f(\mathbf{r})/\sigma, \tag{7.81}$$

which describes the groundwater dynamics reasonably well in most cases. When there is no infiltration, the above equation reduces to its simplest form

$$\nabla^2 \phi(\mathbf{r}) = 0, \tag{7.82}$$

which is the Laplace equation for the head.

Now let us demonstrate how one can solve the groundwater dynamics problem by assuming that we are dealing with a rectangular geometry with nonuniform conductivity and nonzero infiltration. Boundary conditions play a significant role in determining the behavior of groundwater dynamics. We discretize the equation with the scheme discussed in Section 7.3, and the equation becomes

$$\begin{aligned}\phi_{ij} = {} & \frac{1}{4(1+\alpha)\sigma_{ij}} \{(\sigma_{i+1j} + \sigma_{ij})\phi_{i+1j} + (\sigma_{ij} + \sigma_{i-1j})\phi_{i-1j} \\ & + \alpha[(\sigma_{ij+1} + \sigma_{ij})\phi_{ij+1} + (\sigma_{ij} + \sigma_{ij-1})\phi_{ij-1}] + 2h_x^2 f_{ij}\},\end{aligned} \tag{7.83}$$

where $\alpha = (h_x/h_y)^2$. The above difference equation can form the basis for an iterative approach under the relaxation method with

$$\phi_{ij}^{(k+1)} = (1-p)\phi_{ij}^{(k)} + p\phi_{ij}, \tag{7.84}$$

where $\phi_{ij}^{(k)}$ is the value of the kth iteration and ϕ_{ij} is evaluated from Eq. (7.83) with the right-hand side evaluated under $\phi_{ij}^{(k)}$.

We illustrate this procedure by studying an actual example, with $L_x = 1000$ m, $L_y = 500$ m, $\sigma(x, y) = \sigma_0 + ay$ with $\sigma_0 = 1.0$ m/s and $a = -0.04$ s^{-1}, and $f(x, y) = 0$. The boundary condition is $\partial\phi/\partial x = 0$ at $x = 0$ and $x = 1000$ m, $\phi = \phi_0$ at $y = 0$, and $\phi = \phi_0 + b\cos(x/L_x)$ at $y = 500$ m, with $\phi_0 = 200$ m and $b = -20$ m. The four-point formula for the first-order derivative is used to ensure zero partial derivatives for two of the boundaries. The following program is an implementation of the relaxation method to this groundwater dynamics problem.

```
// An example of studying the 2-dimensional groundwater
// dynamics through the relaxation method.

import java.lang.*;
public class Groundwater {
  final static int nx = 100, ny = 50, ni = 5;
  public static void main(String argv[]) {
    double sigma0 = 1, a = -0.04, phi0 = 200, b = -20;
    double lx = 1000, hx = lx/nx, ly = 500, hy =ly/ny;
    double phi[][] = new double[nx+1][ny+1];
    double sigma[][] = new double[nx+1][ny+1];
    double f[][] = new double[nx+1][ny+1];
    double p = 0.5;

 // Set up boundary values and a trial solution
    for (int i=0; i<=nx; ++i) {
      double x = i*hx;
      for (int j=0; j<=ny; ++j) {
        double y = j*hy;
        sigma[i][j]  = sigma0+a*y;
        phi[i][j] = phi0+b*Math.cos(Math.PI*x/lx)*y/ly;
        f[i][j] = 0;
      }
    }
    for (int step=0; step<ni; ++step) {

   // Ensure boundary conditions by 4-point formula
      for (int j=0; j<ny; ++j) {
         phi[0][j] = (4*phi[1][j]-phi[2][j])/3;
         phi[nx][j] = (4*phi[nx-1][j]-phi[nx-2][j])/3;
      }
      relax2d(p, hx, hy, phi, sigma, f);
    }

 // Output the result
    for (int i=0; i<=nx; ++i) {
      double x = i*hx;
      for (int j=0; j<=ny; ++j) {
        double y = j*hy;
        System.out.println(x + " " + y + " "
          + phi[i][j]);
      }
    }
  }

// Method to perform a relaxation step in 2D.

  public static void relax2d(double p, double hx,
```

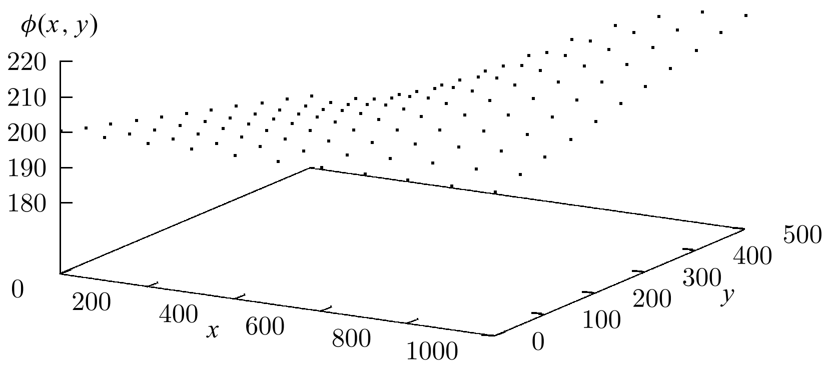


Fig. 7.2 The head of the groundwater obtained from the program given in the text. Here x, y, and $\phi(x, y)$ are all plotted in meters.

```
    double hy, double u[][], double d[][],
    double s[][]) {
    double h2 = hx*hx, a = h2/(hy*hy),
      b = 1/(4*(1+a)), ab = a*b, q = 1-p;
    for (int i=1; i<nx; ++i) {
      for (int j = 1; j <ny; ++j) {
        double xp = b*(d[i+1][j]/d[i][j]+1);
        double xm = b*(d[i-1][j]/d[i][j]+1);
        double yp = ab*(d[i][j+1]/d[i][j]+1);
        double ym = ab*(d[i][j-1]/d[i][j]+1);
        u[i][j] = q*u[i][j]+p*(xp*u[i+1][j]
                  +xm*u[i-1][j]+yp*u[i][j+1]
                  +ym*u[i][j-1]+h2*s[i][j]);
      }
    }
  }
}
```

Note that we have used the four-point formula to update the boundary points at $x = 0$ and $x = 1000$ m to ensure zero partial derivatives there. We can use the six-point formula to improve the accuracy if needed. The output of the above program is shown in Fig. 7.2.

The transient state of groundwater dynamics involves the time variable, and is similar to the situations that we will discuss in the next two sections. Dynamics involving time can be solved with the combination of the discrete scheme discussed in the preceding two sections for the spatial variables and the scheme discussed in Chapter 4 for the initial-value problems. It is interesting that the time evolution involved in the problem is almost identical to the iterations of the relaxation scheme. A stabler scheme requires the use of the tridiagonal-matrix scheme discussed in Section 7.4.

7.7 Initial-value problems

A typical initial-value problem can be either the time-dependent diffusion equation or the time-dependent wave equation. Some initial-value problems are nonlinear equations, such as the equation for a stretched elastic string or the Navier–Stokes equation in fluid dynamics. We can, in most cases, apply the Fourier transform for the time variable of the equation to reduce it to a stationary equation, which can be solved by the relaxation method discussed earlier in this chapter. Then the time dependence can be obtained with an inverse Fourier transform after the solution of the corresponding stationary case is obtained.

For equations with higher-order time derivatives, we can also redefine the derivatives as new variables in order to convert the equations to ones with only first-order time derivatives, as we did in Chapter 4. For example, we can redefine the first-order time derivative in the wave equation, that is, the velocity

$$v(\mathbf{r}, t) = \frac{\partial u(\mathbf{r}, t)}{\partial t} \tag{7.85}$$

as a new variable. Then we have two coupled first-order equations,

$$\frac{\partial u(\mathbf{r}, t)}{\partial t} = v(\mathbf{r}, t), \tag{7.86}$$

$$\frac{1}{c^2}\frac{\partial v(\mathbf{r}, t)}{\partial t} = \nabla^2 u(\mathbf{r}, t) + R(\mathbf{r}, t), \tag{7.87}$$

which describe the same physics as the original second-order wave equation. Note that the above equation set now has a mathematical structure similar to that of a first-order equation such as the diffusion equation

$$\frac{\partial n(\mathbf{r}, t)}{\partial t} = \boldsymbol{\nabla} \cdot D(\mathbf{r})\boldsymbol{\nabla} n(\mathbf{r}, t) + S(\mathbf{r}, t). \tag{7.88}$$

This means we can develop numerical schemes for equations with first-order time derivatives only. In the case of higher-order time derivatives, we will always introduce new variables to reduce the higher-order equation to a first-order equation set. In the next chapter, however, we will introduce some numerical algorithms designed to solve second-order differential equations directly. As one can see, after discretization of the spatial variables, we have practically the same initial-value problem as that discussed in Chapter 4. However, there is one more complication. The specific scheme used to discretize the spatial variables as well as the time variable will certainly affect the stability and accuracy of the solution. Even though it is not the goal here to analyze all aspects of various algorithms, we will still make a comparison among the most popular algorithms with some actual examples in physics and discuss specifically the relevant aspects of the instability under the spatial and time intervals adopted.

In order to analyze the stability of the problem, let us first consider the one-dimensional diffusion equation

$$\frac{\partial n(x, t)}{\partial t} = D\frac{\partial^2 n(x, t)}{\partial^2 x} + S(x, t). \tag{7.89}$$

If we discretize the first-order time derivative by means of the two-point formula with an interval τ and the second-order spatial derivative by means of the three-point formula with an interval h, we obtain a difference equation

$$n_i(t+\tau) = n_i(t) + \gamma[n_{i+1}(t) + n_{i-1}(t) - 2n_i(t)] + \tau S_i(t), \tag{7.90}$$

which is the result of the Euler method, equivalent to that for the initial-value problems introduced in Chapter 4. Here $\gamma = D\tau/h^2$ is a measure of the relative sizes between the space and the time intervals. Note that we have used $n_i(t) = n(x_i, t)$ for notational convenience. So the problem is solved if we know the initial value $n(x, 0)$ and the source $S(x, t)$. However, this algorithm is unstable if γ is significantly larger than 1/2. We can show this by examining the case with $x \in [0, 1]$ and $n(0, t) = n(L, t) = 0$. For detailed discussions, see Young and Gregory (1988, pp. 1078–84).

A better scheme is the Crank–Nicolson method, which modifies the Euler method by using the average of the second-order spatial derivative and the source at t and $t + \tau$ on the right-hand side of the equation, resulting in

$$n_i(t+\tau) = n_i(t) + \frac{1}{2}\{[H_i n_i(t) + \tau S_i(t)] + [H_i n_i(t+\tau) + \tau S_i(t+\tau)]\}, \tag{7.91}$$

where we have used

$$H_i n_i(t) = \gamma[n_{i+1}(t) + n_{i-1}(t) - 2n_i(t)] \tag{7.92}$$

to simplify the notation. The implicit iterative scheme in Eq. (7.91) can be rewritten into the form

$$(2 - H_i)n_i(t+\tau) = (2 + H_i)n_i(t) + \tau[S_i(t) + S_i(t+\tau)], \tag{7.93}$$

which has all the unknown terms at $t + \tau$ on the left. More importantly, Eq. (7.93) is a linear equation set with a tridiagonal coefficient matrix, which can easily be solved as discussed in Section 2.4 for the cubic-spline approximation, or as in Section 7.4 for the problem of a person sitting on a bench. We can also show that the algorithm is stable for any γ and converges as $h \to 0$, and that the error in the solution is on the order of h^2 (Young and Gregory, 1988).

However, the above tridiagonal matrix does not hold if the system is in a higher-dimensional space. There are two ways to deal with this problem in practice. We can discretize the equation in the same manner and then solve the resulting linear equation set with some other methods, such as the Gaussian elimination scheme or a general LU decomposition scheme for a full matrix. A more practical approach is to deal with each spatial coordinate separately. For example, if we are dealing with the two-dimensional diffusion equation, we have

$$H_{ij} n_{ij}(t) = (H_i + H_j)n_{ij}(t), \tag{7.94}$$

with

$$H_i n_{ij}(t) = \gamma_x[n_{i+1j}(t) + n_{i-1j}(t) - 2n_{ij}(t)], \tag{7.95}$$

$$H_j n_{ij}(t) = \gamma_y[n_{ij+1}(t) + n_{ij-1}(t) - 2n_{ij}(t)]. \tag{7.96}$$

Here $\gamma_x = D\tau/h_x^2$ and $\gamma_y = D\tau/h_y^2$. The decomposition of H_{ij} into H_i and H_j can be used to take one half of each time step along the x direction and the other half along the y direction with

$$(2 - H_j)n_{ij}\left(t + \frac{\tau}{2}\right) = (2 + H_j)n_{ij}(t) + \frac{\tau}{2}\left[S_{ij}(t) + S_{ij}\left(t + \frac{\tau}{2}\right)\right], \tag{7.97}$$

and

$$\begin{aligned}(2 - H_i)n_{ij}(t + \tau) = (2 + H_i)n_{ij}&\left(t + \frac{\tau}{2}\right)\\ &+ \frac{\tau}{2}\left[S_{ij}\left(t + \frac{\tau}{2}\right) + S_{ij}(t + \tau)\right],\end{aligned} \tag{7.98}$$

which is a combined implicit scheme developed by Peaceman and Rachford (1955). As we can see, in each of the above two steps, we have a tridiagonal coefficient matrix. We still have to be careful in using the Peaceman–Rachford algorithm. Even though it has the same accuracy as the Crank–Nicolson algorithm, the convergence in the Peaceman–Rachford algorithm can sometimes be very slow in practice. We should compare the Peaceman–Rachford algorithm with the Crank–Nicolson method in test runs before deciding which one to use in a specific problem. We can easily monitor the convergence of the algorithm by changing h_x and h_y. For more discussions on and comparisons of various algorithms, see Young and Gregory (1988).

7.8 Temperature field of a nuclear waste rod

The temperature increase around nuclear waste rods is not only an interesting physics problem but also an important safety question that has to be addressed before building nuclear waste storage facilities. The typical plan for nuclear waste storage is an underground facility with the waste rods arranged in an array. In this section, we will study a very simple case, the temperature around a single rod.

The relation between the thermal current density and the temperature gradient is given by Fourier's law

$$\mathbf{j} = -\boldsymbol{\sigma} \cdot \nabla T, \tag{7.99}$$

where $\boldsymbol{\sigma}$ is the thermal conductivity, which is typically a tensor of rank 2 or a 3×3 matrix. For an isotropic material, $\boldsymbol{\sigma}$ reduces to a diagonal matrix with identical diagonal elements σ. Note that this relation is a typical result from the linear-response theory, which provides all sorts of similar relations between the gradient of a field and the corresponding current density.

If we combine Fourier's law with the energy conservation, we have

$$c\rho\frac{\partial T}{\partial t} - \nabla \cdot \boldsymbol{\sigma} \cdot \nabla T = q(\mathbf{r}, t), \tag{7.100}$$

where c the specific heat and ρ the density of the material, and $q(\mathbf{r}, t)$ is the heat released into the system per unit volume per unit time at the position $\mathbf{r}$ and time t. Assuming that the system is isotropic, then we have

$$\frac{1}{\kappa}\frac{\partial T(\mathbf{r}, t)}{\partial t} - \nabla^2 T(\mathbf{r}, t) = S(\mathbf{r}, t), \tag{7.101}$$

where $\kappa = \sigma/c\rho$ is called the *diffusivity*, a system-dependent parameter, and $S = q/\sigma$ is the effective source.

For the specific geometry of a nuclear waste rod, we can assume that the system is effectively two-dimensional with the temperature field having a circular symmetry around the rod. The above equation then becomes

$$\frac{1}{\kappa}\frac{\partial T(r, t)}{\partial t} - \frac{1}{r}\frac{\partial}{\partial r}r\frac{\partial T(r, t)}{\partial r} = S(r, t), \tag{7.102}$$

with the boundary condition $T(\infty, t) = T_{\mathrm{E}}$, the environment temperature. For the purpose of the numerical evaluation, we assume that

$$S(r, t) = \begin{cases} \dfrac{T_0}{a^2}e^{-t/\tau_0} & \text{for } r \leq a, \\ 0 & \text{elsewhere}, \end{cases} \tag{7.103}$$

where T_0, a, and τ_0 are system-dependent parameters. If we take advantage of the cylindrical symmetry of the problem, the discretized equation along the radial direction is equivalent to a one-dimensional diffusion equation with a spatially dependent diffusion coefficient $D \propto r$, that is,

$$(2 - H_i)T(t + \tau) = (2 + H_i)T(t) + \tau\kappa[S_i(t) + S_i(t + \tau)], \tag{7.104}$$

with

$$H_i T(t) = \frac{\tau\kappa}{r_i h^2}[r_{i+1/2}T_{i+1}(t) + r_{i-1/2}T_{i-1}(t) - 2r_i T_i(t)]. \tag{7.105}$$

The numerical problem is now more complicated than the constant diffusion coefficient case, but the matrix involved is still tridiagonal. We can solve the problem iteratively with the simple method described in Section 2.4 by performing an LU decomposition first and then a forward substitution followed by a backward substitution.

Special care must be given at $r = 0$ and the cut-off radius r_c. Consider that $i = 0$ at $r = 0$ and $i = n + 1$ at $r = r_c$. Because the energy cannot flow into the $r = 0$ region, we must have

$$\left.\frac{\partial T}{\partial r}\right|_{r=0} = 0. \tag{7.106}$$

Then we can use Eq. (7.72) to express the temperature at $r = 0$ in terms of the temperatures at the next two points. One way to fix the temperature at the cut-off radius is by extrapolation. We will leave this as an exercise for the reader.

Now let us see an actual numerical example. Assuming that the system (the rod and its environment) is close to concrete with $c = 789$ J/(kg K),

$\sigma = 1.00$ W/(m K), and $\rho = 2.00 \times 10^3$ kg/m^3, we have $\kappa = \sigma/c\rho = 6.34 \times 10^{-7}$ m^2/s $= 2.00 \times 10^7$ cm^2/100 yr. We will take $a = 25$ cm, $T_0 = 1.00$ K, and $\tau_0 = 100$ years.

To simplify the problem here, we will assume that the temperature at the cut-off radius $r_c = 100$ cm is fixed at 300 K under ventilation. This restriction can be removed by extrapolation. The initial condition is given by $\Delta T(r, 0) = 0$ K or $T(r, 0) = T_E$, and the boundary condition $\Delta T(r_c, t) = 0$. The following program puts everything together for the temperature field change around a nuclear waste rod.

```
// A program to study the time-dependent temperature
// field around a nuclear waste rod in a 2D model.

import java.lang.*;
public class Nuclear {
  final static int nx = 100, n = 5, nt = 1000,
    mt = 1000;
  public static void main(String argv[]) {
    double d[] = new double[nx];
    double e[] = new double[nx];
    double c[] = new double[nx];
    double b[] = new double[nx];
    double p[] = new double[nx];
    double s[][] = new double[nt+1][nx];
    double T[][] = new double[nt+1][nx+1];
    double dt = 1.0/mt, tc = 1, T0 = 1, kappa = 2e7;
    double ra = 25, rb = 100, h = rb/nx, h2 = h*h;
    double s0 = dt*kappa*T0/(ra*ra), g = dt*kappa/h2;

 // Assign the elements in the matrix 2-H_i
    for (int i=0; i<nx; ++i) {
      d[i] = 2*(1+g);
      e[i] = -(1+0.5/(i+1))*g;
      c[i] = -(1-0.5/(i+2))*g;
    }

 // Modify the first equation from T"=0 at r=0
    d[0] -= 2*g/3;
    e[0] += g/6;

 // Assign the source of the radiation heat
    int na = (int) (ra/h);
    for (int i=0; i<=nt; ++i) {
      double t = -dt*i/tc;
      for (int j=0; j<na-1; ++j) {
        s[i][j] = s0*Math.exp(t);
      }
    }

 // Find the temperature field recursively
    for (int i=1; i<=nt; ++i) {

   // Assign the elements in the matrix 2+H_0
      double d0 = 2*(1-g);
      double e0 = (1+0.5)*g;
```

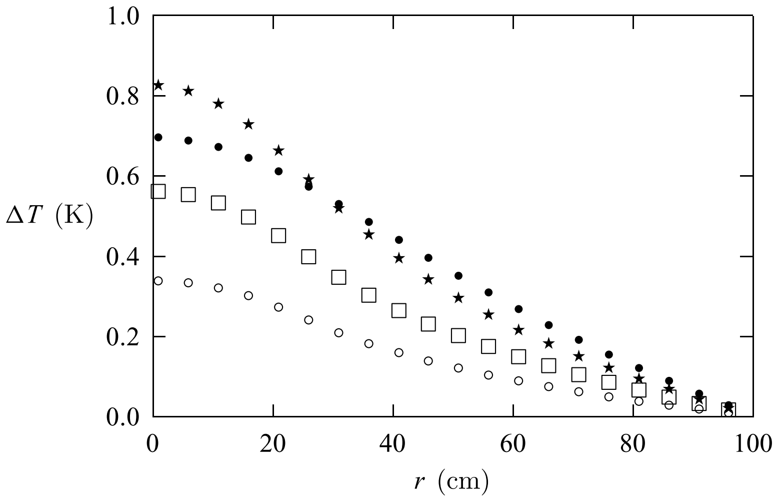


Fig. 7.3 Temperature change around a nuclear waste rod calculated with the program given in the text. Solid dots are for $t = 1$ year, stars are for $t = 10$ years, squares are for $t = 50$ years, and circles are for $t = 100$ years.

```
      double c0 = (1-0.5)*g;

   // Evaluate b[0] under the condition T"=0 at r=0
      b[0] = d0*T[i-1][0]+e0*T[i-1][1]
            +c0*(4*T[i-1][0]-T[i-1][1])/3
            +s[i-1][0]+s[i][0];

   // Find the elements in the array b[i]
      for (int j=1; j<nx; ++j) {

     // Assign the elements in the matrix 2+H_0
        d0 = 2*(1-g);
        e0 = (1+0.5/(j+1))*g;
        c0 = (1-0.5/(j+1))*g;

     // Obtain the elements from the last recursion
        b[j] = d0*T[i-1][j]+e0*T[i-1][j+1]
              +c0*T[i-1][j-1]+s[i-1][j]+s[i][j];
      }

   // Obtain the solution of the temperature field
      p = tridiagonalLinearEq(d, e, c, b);
      for (int j=0; j<nx; ++j) T[i][j] = p[j];
    }

 // Output the result at every n spatial data points
    for (int j=0; j<nx; j+=n) {
      double r = h*(j+1);
      System.out.println(r + " " + T[nt][j]);
    }
  }

// Method to solve the tridiagonal linear equation set.

  public static double[] tridiagonalLinearEq(double d[],
    double e[], double c[], double b[]) {...}
}
```

The parameters used in the above programs are not from actual storage units. However, if we want to study the real situation, we only need to modify the parameters for the actual system and environment. The numerical scheme and program are quite applicable to realistic situations. The output of the program is shown in Fig. 7.3. Note that the maximum change of the temperature (the peak) increases with time initially and then decreases afterward. The largest change of the temperature over time and the rate of the temperature change are critical in designing a safe and practical nuclear waste storage.

Exercises

7.1 Consider the Poisson equation

$$\nabla^2\phi(x, y) = -\rho(x, y)/\epsilon_0$$

from electrostatics on a rectangular geometry with $x \in [0, L_x]$ and $y \in [0, L_y]$. Write a program that solves this equation using the relaxation method. Test your program with: (a) $\rho(x, y) = 0$, $\phi(0, y) = \phi(L_x, y) = \phi(x, 0) = 0, \phi(x, L_y) = 1$ V, $L_x = 1$ m, and $L_y = 1.5$ m; (b) $\rho(x, y)/\epsilon_0 = 1$ V/m^2, $\phi(0, y) = \phi(L_x, y) = \phi(x, 0) = \phi(x, L_y) = 0$, and $L_x = L_y = 1$ m.

7.2 Develop a numerical scheme that solves the Poisson equation

$$\nabla^2\phi(r, \theta) = -\rho(r, \theta)/\epsilon_0$$

in polar coordinates. Assume that the geometry of the boundary is a circular ring with the potential at the inner radius, $\phi(a, \theta)$, and outer radius, $\phi(b, \theta)$, given. Test the scheme with some special choice of the boundary values.

7.3 If the charge distribution in the Poisson equation is spherically symmetric, derive the difference equation for the potential along the radius. Test the algorithm with $\rho(r) = \rho_0 e^{-r/r_0}$ in a program.

7.4 Derive the relaxation scheme for a three-dimensional system with rectangular boundaries. Analyze the choice of p in a program for the Poisson equation with constant potentials at the boundaries.

7.5 Modify the program for the groundwater dynamics problem given in Section 7.6 to study the general case of the transient state, that is, the case where the time derivative of the head is nonzero. Apply the program to study the stationary case given there as well as the evolution of the solution with time if the infiltration $f(\mathbf{r}, t) = f_0 e^{-t/\tau}$ for various f_0 and τ.

7.6 Write a program that solves the wave equation of a finite string with both ends fixed. Assume that the initial displacement and velocity are given. Test the program with some specific choice of the initial condition.

7.7 Solve the time-dependent Schrödinger equation using the Crank–Nicolson method. Consider the one-dimensional case and test it by applying it to the problem of a square well with a Gaussian initial state coming in from the left.

7.8 Obtain the algorithm for solving the three-dimensional wave equation with 1/3 of the time step applied to each coordinate direction. Test the algorithm with the equation under the homogeneous Dirichlet boundary condition. Take the initial condition as $u(\mathbf{r};0) = 0$ and $v(\mathbf{r};0) = \sin(\pi x/L_x)\sin(\pi y/L_y)\sin(\pi z/L_z)$, where L_x, L_y, and L_z are the lengths of the box along the three directions.

7.9 Consider an elastic rope that is fixed at both ends at $x = 0$ and $x = L = 8.00$ m, subject to a tension $T = 5.00 \times 10^3$ N. Two friends are sitting on the rope, each with a mass distribution (mass per unit length) from a truncated Gaussian

$$\rho_i(x) = \frac{m_i}{2a_i} e^{-(x-x_i)^2/a_i^2},$$

for $|x - x_i| \leq a_i$. Assume that $m_1 = 40.0$ kg, $m_2 = 55.0$ kg, $x_1 = 3.50$ m, $x_2 = 4.10$ m, $a_1 = 0.300$ m, and $a_2 = 0.270$ m, and the rope has a density of $\rho_0 = 4.00$ kg/m. (a) Show that the motion of the rope is described by

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{T}{\rho(x)} \frac{\partial^2 u(x,t)}{\partial x^2} - g,$$

where $u(x,t)$ is the displacement of the rope at the position x and time t, $\rho(x) = \rho_0 + \rho_1(x) + \rho_2(x)$, and $g = 9.8\,\text{m/s}^2$ is the magnitude of the gravitational constant. (b) Find the displacement of the rope when it is in equilibrium. Where is the maximum displacement of the rope? (c) Find the first five angular frequencies if the system is in vibration and plot out the corresponding eigenstates.

7.10 Solve the nuclear waste rod problem discussed in Section 7.8 with the extrapolation of the temperature made at the cut-off radius. Compare various extrapolation schemes, linear extrapolation, quadratic extrapolation, the Lagrange extrapolation with multipoints, and the extrapolation based on the cubic spline. Which scheme works the best? Are there any significant differences between the result here and that found in Section 7.8 with a fixed temperature at the cut-off radius and why?

7.11 Simulate the process of burning a hole in the middle of a large silver sheet with a propane torch numerically. Assuming that the torch head can generate a power of about 5000 W within a circle that has a 4 mm radius, estimate how long it will take to create a hole in a sheet that has a thickness of 2 mm.

7.12 The macroscopic state of Bose–Einstein condensate is described by the Gross–Pitaevskii equation

$$i\hbar \frac{\partial \phi(\mathbf{r}, t)}{\partial t} = \left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\text{ext}}(\mathbf{r}) + g|\phi(\mathbf{r}, t)|^2 \right] \phi(\mathbf{r}, t),$$

where $\phi(\mathbf{r}, t)$ is the macroscopic wavefunction, m and g are two system-related parameters, and $V_{\text{ext}}(\mathbf{r})$ is the external potential. Develop a program that solves this equation numerically. Consider various potentials, such as a parabolic or a square well.

Chapter 8
Molecular dynamics simulations

Most physical systems are collections of interacting objects. For example, a drop of water contains more than 10^{21} water molecules, and a galaxy is a collection of millions and millions of stars. In general, there is no analytical solution that can be found for an interacting system with more than two objects. We can solve the problem of a two-body system, such as the Earth–Sun system, analytically, but not a three-body system, such as the Moon–Earth–Sun system. The situation is similar in quantum mechanics, in that one can obtain the energy levels of the hydrogen atom (one electron and one proton) analytically, but not those the helium atom (two electrons and a nucleus). Numerical techniques beyond those we have discussed so far are needed to study a system of a large number of interacting objects, or the so-called many-body system. Of course, there is a distinction between three-body systems such as the Moon–Earth–Sun system and a more complicated system, such as a drop of water. Statistical mechanics has to be applied to the latter.

8.1 General behavior of a classical system

In this chapter, we introduce a class of simulation techniques called *molecular dynamics*, which solves the dynamics of a classical many-body system described by the Hamiltonian

$$\mathcal{H} = E_{\mathrm{K}} + E_{\mathrm{P}} = \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2m_i} + \sum_{i>j=1}^{N} V(\mathbf{r}_{ij}) + \sum_{i=1}^{N} U_{\mathrm{ext}}(\mathbf{r}_i), \tag{8.1}$$

where E_{K} and E_{P} are the kinetic energy and potential energy of the system, respectively, m_i, $\mathbf{r}_i$, and $\mathbf{p}_i$ are the mass, position vector, and momentum of the ith particle, and $V(\mathbf{r}_{ij})$ and $U(\mathbf{r}_i)$ are the corresponding interaction energy and external potential energy. From Hamilton's principle, the position vector and momentum satisfy

$$\dot{\mathbf{r}}_i = \frac{\partial \mathcal{H}}{\partial \mathbf{p}_i} = \frac{\mathbf{p}_i}{m_i}, \tag{8.2}$$

$$\dot{\mathbf{p}}_i = -\frac{\partial \mathcal{H}}{\partial \mathbf{q}_i} = \mathbf{f}_i, \tag{8.3}$$

which are called Hamilton's equations and valid for any given $\mathcal{H}$, including the case of a system that can exchange energy or heat with its environment. Here the force $\mathbf{f}_i$ is given by

$$\mathbf{f}_i = -\nabla_i U_{\text{ext}}(\mathbf{r}_i) - \sum_{j\neq i} \nabla_i V(\mathbf{r}_{ij}). \tag{8.4}$$

The methods for solving Newton's equation discussed in Chapter 1 and Chapter 4 can be used to solve the above equation set. However, those methods are not as practical as the ones about to be discussed, in terms of the speed and accuracy of the computation and given the statistical nature of large systems. In this chapter, we discuss several commonly used molecular dynamics simulation schemes and offer a few examples.

Before we go into the numerical schemes and actual physical systems, we need to discuss several issues. There are several ways to simulate a many-body system. Most simulations are done either through a stochastic process, such as the Monte Carlo simulation, which will be discussed in Chapter 10, or through a deterministic process, such as a molecular dynamics simulation. Some numerical simulations are performed in a hybridized form of both, for example, Langevin dynamics, which is similar to molecular dynamics except for the presence of a random dissipative force, and Brownian dynamics, which is performed under the condition that the acceleration is balanced out by the drifting and random dissipative forces. We will not discuss Langevin or Brownian dynamics in this book, but interested readers can find detailed discussions in Heermann (1986) and Kadanoff (2000).

Another issue is the distribution function of the system. In statistical mechanics, each special environment is dealt with by way of a special ensemble. For example, for an isolated system we use the microcanonical ensemble, which assumes a constant total energy, number of particles, and volume. A system in good contact with a thermal bath is dealt with using the canonical ensemble, which assumes a constant temperature, number of particles, and volume (or pressure). For any given ensemble, the system is described by a probability function $\mathcal{W}(\mathbf{R}, \mathbf{P})$, which is in general a function of phase space, consisting of all coordinates and momenta of the particles $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)$ and $\mathbf{P} = (\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N)$, and other quantities, such as temperature, total particle number of the system, and so forth. For the canonical ensemble, we have

$$\mathcal{W}(\mathbf{R}, \mathbf{P}) = \frac{1}{\mathcal{N}} e^{-\mathcal{H}/k_\mathrm{B}T}, \tag{8.5}$$

where T is the temperature of the system, k_B is the Boltzmann constant, and $\mathcal{N}$ is a normalization constant. For an N-particle quasi-classical system, $\mathcal{N} = N!h^{3N}$, where h is the Planck constant. Note that we can separate the position dependence and momentum dependence in $\mathcal{W}(\mathbf{R}, \mathbf{P})$ if they are not coupled in $\mathcal{H}$. Any average of the momentum-dependent quantity becomes quite simple because of the quadratic behavior of the momentum in $\mathcal{H}$. So we concentrate on the position dependence here. The statistical average of a physical quantity $A(\mathbf{R}, \mathbf{P})$ is then given by

$$\langle A \rangle = \frac{1}{\mathcal{Z}} \int A(\mathbf{R}, \mathbf{P}) \mathcal{W}(\mathbf{R}, \mathbf{P})\, d\mathbf{R}\, d\mathbf{P}, \tag{8.6}$$

where $\mathcal{Z}$ is the partition function of the system from

$$\mathcal{Z} = \int \mathcal{W}(\mathbf{R}, \mathbf{P})\, d\mathbf{R}\, d\mathbf{P}. \tag{8.7}$$

The ensemble average given above is equivalent to the time average

$$\langle A \rangle = \lim_{\tau \to \infty} \frac{1}{\tau} \int_0^\tau A(t)\, dt, \tag{8.8}$$

if the system is *ergodic*: that is, every possible state is accessed with an equal probability. Because molecular dynamics simulations are deterministic in nature, almost all physical quantities are obtained through time averages. Sometimes the average over all the particles is also needed to characterize the system. For example, the average kinetic energy of the system can be obtained from any ensemble average, and the result is given by the partition theorem

$$\langle E_\mathrm{K} \rangle = \left\langle \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2m_i} \right\rangle = \frac{G}{2} k_\mathrm{B} T, \tag{8.9}$$

where G is the total number of degrees of freedom. For a very large system, $G \simeq 3N$, because each particle has three degrees of freedom. In molecular dynamics simulations, the average kinetic energy of the system can be obtained through

$$\langle E_\mathrm{K} \rangle = \frac{1}{M} \sum_{j=1}^{M} E_\mathrm{K}(t_j), \tag{8.10}$$

where M is the total number of data points taken at different time steps and $E_\mathrm{K}(t_j)$ is the kinetic energy of the system at time t_j. If the system is ergodic, the time average is equivalent to the ensemble average. The temperature T of the simulated system is then given by the average kinetic energy with the application of the partition theorem, $T = 2\langle E_\mathrm{K} \rangle / G k_\mathrm{B}$.

8.2 Basic methods for many-body systems

In general, we can define an n-body density function

$$\rho_n(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n) = \frac{1}{\mathcal{Z}} \frac{N!}{(N-n)!} \int \mathcal{W}(\mathbf{R}, \mathbf{P})\, d\mathbf{R}_n\, d\mathbf{P}, \tag{8.11}$$

where $d\mathbf{R}_n = d\mathbf{r}_{n+1}\, d\mathbf{r}_{n+2} \cdots d\mathbf{r}_N$. Note that the particle density $\rho(\mathbf{r}) = \rho_1(\mathbf{r})$ is the special case of $n = 1$. The two-body density function is related to the *pair-distribution function* $g(\mathbf{r}, \mathbf{r}')$ through

$$\rho_2(\mathbf{r}, \mathbf{r}') = \rho(\mathbf{r}) g(\mathbf{r}, \mathbf{r}') \rho(\mathbf{r}'), \tag{8.12}$$

and one can easily show that

$$\rho_2(\mathbf{r}, \mathbf{r}') = \langle \hat{\rho}(\mathbf{r})\hat{\rho}(\mathbf{r}')\rangle - \delta(\mathbf{r} - \mathbf{r}')\rho(\mathbf{r}), \tag{8.13}$$

where the first term is the so-called *density–density correlation function*. Here $\hat{\rho}(\mathbf{r})$ is the density operator, defined as

$$\hat{\rho}(\mathbf{r}) = \sum_{i=1}^{N} \delta(\mathbf{r} - \mathbf{r}_i). \tag{8.14}$$

The density of the system is given by the average of the density operator,

$$\rho(\mathbf{r}) = \langle \hat{\rho}(\mathbf{r})\rangle. \tag{8.15}$$

If the density of the system is nearly a constant, the expression for $g(\mathbf{r}, \mathbf{r}')$ can be reduced to a much simpler form

$$g(\mathbf{r}) = \frac{1}{\rho N}\left\langle \sum_{i\neq j}^{N} \delta(\mathbf{r} - \mathbf{r}_{ij})\right\rangle, \tag{8.16}$$

where ρ is the average density from the points $\mathbf{r}' = 0$ and $\mathbf{r}$. If the angular distribution is not the information needed, we can take the angular average to obtain the *radial distribution function*

$$g(r) = \frac{1}{4\pi}\int g(\mathbf{r}) \sin\theta \, d\theta \, d\phi, \tag{8.17}$$

where θ and ϕ are the polar and azimuthal angles from the spherical coordinate system. The pair-distribution or radial distribution function is related to the *static structure factor* $S(\mathbf{k})$ through the Fourier transform

$$S(\mathbf{k}) - 1 = \rho \int [g(\mathbf{r}) - 1]e^{-i\mathbf{k}\cdot\mathbf{r}} \, d\mathbf{r} \tag{8.18}$$

and its inverse

$$g(\mathbf{r}) - 1 = \frac{1}{(2\pi)^3\rho}\int [S(\mathbf{k}) - 1]e^{i\mathbf{k}\cdot\mathbf{r}} \, d\mathbf{k}. \tag{8.19}$$

The angular average of $S(\mathbf{k})$ is given by

$$S(k) - 1 = 4\pi\rho \int_0^\infty \frac{\sin kr}{kr}[g(r) - 1]r^2 \, dr. \tag{8.20}$$

The structure factor of a system can be measured with the light- or neutron-scattering experiment.

The behavior of the pair-distribution function can provide a lot of information regarding the translational nature of the particles in the system. For example, a solid structure would have a pair-distribution function with sharp peaks at the distances of nearest neighbors, next nearest neighbors, and so forth. If the system is a liquid, the pair-distribution still has some broad peaks at the average distances of nearest neighbors, next nearest neighbors, and so forth, but the feature fades away after several peaks.

If the bond orientational order is important, one can also define an orientational correlation function

$$g_n(\mathbf{r}, \mathbf{r}') = \langle q_n(\mathbf{r}) q_n(\mathbf{r}') \rangle, \tag{8.21}$$

where $q_n(\mathbf{r})$ is a quantity associated with the orientation of a specific bond. Detailed discussions on orientational order can be found in Strandburg (1992).

Here we discuss how one can calculate $\rho(\mathbf{r})$ and $g(r)$ in a numerical simulation. The density at a specific point is given by

$$\rho(\mathbf{r}) \simeq \frac{\langle N(\mathbf{r}, \Delta r) \rangle}{\Omega(\mathbf{r}, \Delta r)}, \tag{8.22}$$

where $\Omega(\mathbf{r}, \Delta r)$ is the volume of a sphere centered at $\mathbf{r}$ with a radius Δr and $N(\mathbf{r}, \Delta r)$ is the number of particles in the volume. Note that we may need to adjust the radius Δr to have a smooth and realistic density distribution $\rho(\mathbf{r})$ for a specific system. The average is taken over the time steps.

Similarly, we can obtain the radial distribution function numerically. We need to measure the radius r from the position of a specific particle $\mathbf{r}_i$, and then the radial distribution function $g(r)$ is the probability of another particle's showing up at a distance r. Numerically, we have

$$g(r) \simeq \frac{\langle \Delta N(r, \Delta r) \rangle}{\rho \Delta\Omega(r, \Delta r)}, \tag{8.23}$$

where $\Delta\Omega(r, \Delta r) \simeq 4\pi r^2 \Delta r$ is the volume element of a spherical shell with radius r and thickness Δr and $\Delta N(r, \Delta r)$ is the number of particles in the shell with the ith particle at the center of the sphere. The average is taken over the time steps as well as over the particles, if necessary.

The dynamics of the system can be measured from the displacement of the particles in the system. We can evaluate the time dependence of the mean-square displacement of all the particles,

$$\Delta^2(t) = \frac{1}{N} \sum_{i=1}^{N} [\mathbf{r}_i(t) - \mathbf{r}_i(0)]^2, \tag{8.24}$$

where $\mathbf{r}_i(t)$ is the position vector of the ith particle at time t. For a solid system, $\Delta^2(t)$ is relatively small and does not grow with time, and the particles are in nondiffusive, or oscillatory, states. For a liquid system, $\Delta^2(t)$ grows linearly with time:

$$\Delta^2(t) = 6Dt + \Delta^2(0), \tag{8.25}$$

where D is the *self-diffusion coefficient* (a measure of the motion of a particle in a medium of identical particles) and $\Delta^2(0)$ is a time-independent constant. The particles are then in diffusive, or propagating, states.

The very first issue in numerical simulations for a bulk system is how to extend a finite simulation box to model the nearly infinite system. A common practice is to use a periodic boundary condition, that is, to approximate an infinite system by piling up identical simulation boxes periodically. A periodic boundary condition removes the conservation of the angular momentum of the simulated system (particles in one simulation box), but still preserves the translational symmetry of the center of mass. So the temperature is related to the average kinetic energy by

$$\langle E_K \rangle = \frac{3}{2}(N-1)k_B T, \tag{8.26}$$

where the factor $(N-1)$ is due to the removal of the rotation around the center of mass.

The remaining issue, then, is how to include the interactions among the particles in different simulation boxes. If the interaction is a short-range interaction, one can truncate it at a cut-off length r_c. The interaction $V(r_c)$ has to be small enough that the truncation does not affect the simulation results significantly. A typical simulation box usually has much larger dimensions than r_c. For a three-dimensional cubic box with sides of length L, the total interaction potential can be evaluated with many fewer summations than $N!/2$, the number of possible pairs in the system. For example, if we have $L/2 > r_c$, and if $|x_{ij}|$, $|y_{ij}|$, and $|z_{ij}|$ are all smaller than $L/2$, we can use $V_{ij} = V(r_{ij})$; otherwise, we use the corresponding point in the neighboring box. For example, if $|x_{ij}| > L/2$, we can replace x_{ij} with $x_{ij} \pm L$ in the interaction. We can deal with y and z coordinates similarly. In order to avoid a finite jump at the truncation, one can always shift the interaction to $V(r) - V(r_c)$ to make sure that it is zero at the truncation.

The pressure of a bulk system can be evaluated from the pair-distribution function through

$$P = \rho k_B T - \frac{2\pi\rho^2}{3}\int_0^\infty \frac{\partial V(r)}{\partial r} g(r) r^3\, dr, \tag{8.27}$$

which is the result of the virial theorem that relates the average kinetic energy to the average potential energy of the system. The correction due to the truncation of the potential is then given by

$$\Delta P = -\frac{2\pi\rho^2}{3}\int_{r_c}^\infty \frac{\partial V(r)}{\partial r} g(r) r^3\, dr, \tag{8.28}$$

which is useful for estimating the influence on the pressure from the truncation in the interaction potential. Numerically, one can also evaluate the pressure from the time average

$$\langle w \rangle = \frac{1}{3}\sum_{i>j}\langle \mathbf{r}_{ij} \cdot \mathbf{f}_{ij} \rangle, \tag{8.29}$$

because $g(r)$ can be interpreted as the probability of seeing another particle at a distance r. Then we have

$$P = \rho k_B T + \frac{\rho}{N}\langle w \rangle + \Delta P, \tag{8.30}$$

which can be evaluated quite easily, because at every time step the force $\mathbf{f}_{ij} = -\nabla V(r_{ij})$ is calculated for each particle pair.

8.3 The Verlet algorithm

Hamilton's equations given in Eqs. (8.2) and (8.3) are equivalent to Newton's equation

$$m_i \frac{d^2\mathbf{r}_i}{dt^2} = \mathbf{f}_i, \tag{8.31}$$

for the ith particle in the system. To simplify the notation, we will use $\mathbf{R}$ to represent all the coordinates $(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)$ and $\mathbf{G}$ to represent all the accelerations $(\mathbf{f}_1/m_1, \mathbf{f}_2/m_2, \dots, \mathbf{f}_N/m_N)$. Thus, we can rewrite Newton's equations for all the particles as

$$\frac{d^2\mathbf{R}}{dt^2} = \mathbf{G}. \tag{8.32}$$

If we apply the three-point formula to the second-order derivative $d^2\mathbf{R}/dt^2$, we have

$$\frac{d^2\mathbf{R}}{dt^2} = \frac{1}{\tau^2}(\mathbf{R}_{k+1} - 2\mathbf{R}_k + \mathbf{R}_{k-1}) + O(\tau^2), \tag{8.33}$$

with $t = k\tau$. We can also apply the three-point formula to the velocity

$$\mathbf{V} = \frac{d\mathbf{R}}{dt} = \frac{1}{2\tau}(\mathbf{R}_{k+1} - \mathbf{R}_{k-1}) + O(\tau^2). \tag{8.34}$$

After we put all the above together, we obtain the simplest algorithm, which is called the *Verlet algorithm*, for a classical many-body system, with

$$\mathbf{R}_{k+1} = 2\mathbf{R}_k - \mathbf{R}_{k-1} + \tau^2\mathbf{G}_k + O(\tau^4), \tag{8.35}$$

$$\mathbf{V}_k = \frac{\mathbf{R}_{k+1} - \mathbf{R}_{k-1}}{2\tau} + O(\tau^2). \tag{8.36}$$

The Verlet algorithm can be started if the first two positions $\mathbf{R}_0$ and $\mathbf{R}_1$ of the particles are given. However, in practice, only the initial position $\mathbf{R}_0$ and initial velocity $\mathbf{V}_0$ are given. Therefore, we need to figure out $\mathbf{R}_1$ before we can start the recursion. A common practice is to treat the force during the first time interval $[0, \tau]$ as a constant, and then to apply the kinematic equation to obtain

$$\mathbf{R}_1 \simeq \mathbf{R}_0 + \tau\mathbf{V}_0 + \frac{\tau^2}{2}\mathbf{G}_0, \tag{8.37}$$

where $\mathbf{G}_0$ is the acceleration vector evaluated at the initial configuration $\mathbf{R}_0$. Of course, the position $\mathbf{R}_1$ can be improved by carrying out the Taylor expansion to higher-order terms if the accuracy in the first two points is critical. We can also replace $\mathbf{G}_0$ in Eq. (8.37) with the average $(\mathbf{G}_0 + \mathbf{G}_1)/2$, with $\mathbf{G}_1$ evaluated at $\mathbf{R}_1$, given from Eq. (8.37). This procedure can be iterated several times before starting the algorithm for the velocity $\mathbf{V}_1$ and the next position $\mathbf{R}_2$.

The Verlet algorithm has advantages and disadvantages. It preserves the time reversibility that is one of the important properties of Newton's equation. The rounding error may eventually destroy this time symmetry. The error in the velocity is two orders of magnitude higher than the error in the position. In many applications, we may only need information about the positions of the particles, and the Verlet algorithm yields very high accuracy for the position. If the velocity is not needed, we can totally ignore the evaluation of the velocity, since the evaluation of the position does not depend on the velocity at each time step. The biggest disadvantage of the Verlet algorithm is that the velocity is evaluated one time step behind the position. However, this lag can be removed if the velocity is evaluated directly from the force. A two-point formula would yield

$$\mathbf{V}_{k+1} = \mathbf{V}_k + \tau \mathbf{G}_k + O(\tau^2). \tag{8.38}$$

We would get much better accuracy if we replaced $\mathbf{G}_k$ with the average $(\mathbf{G}_k + \mathbf{G}_{k+1})/2$. The new position can be obtained by treating the motion within $t \in [k\tau, (k+1)\tau]$ as motion with a constant acceleration $\mathbf{G}_k$; that is,

$$\mathbf{R}_{k+1} = \mathbf{R}_k + \tau \mathbf{V}_k + \frac{\tau^2}{2}\mathbf{G}_k. \tag{8.39}$$

Then a variation of the Verlet algorithm with the velocity calculated at the same time step of the position is

$$\mathbf{R}_{k+1} = \mathbf{R}_k + \tau \mathbf{V}_k + \frac{\tau^2}{2}\mathbf{G}_k + O(\tau^4), \tag{8.40}$$

$$\mathbf{V}_{k+1} = \mathbf{V}_k + \frac{\tau}{2}(\mathbf{G}_{k+1} + \mathbf{G}_k) + O(\tau^2). \tag{8.41}$$

Note that the evaluation of the position still has the same accuracy because the velocity is now updated according to Eq. (8.41), which provides the cancelation of the third-order term in the new position.

Here we demonstrate this *velocity version* of the Verlet algorithm with a very simple example, the motion of Halley's comet, which has a period of about 76 years.

The potential between the comet and the Sun is given by

$$V(r) = -G\frac{Mm}{r}, \tag{8.42}$$

where r is the distance between the comet and the Sun, M and m are the respective masses of the Sun and the comet, and G is the gravitational constant. If we use the center-of-mass coordinate system as described in Chapter 3 for the two-body collision, the dynamics of the comet is governed by

$$\mu\frac{d^2\mathbf{r}}{dt^2} = \mathbf{f} = -GMm\frac{\mathbf{r}}{r^3}, \tag{8.43}$$

with the reduced mass

$$\mu = \frac{Mm}{M+m} \simeq m, \tag{8.44}$$

for the case of Halley's comet. We can take the farthest point (aphelion) as the starting point, and then we can easily obtain the comet's whole orbit with one of the versions of the Verlet algorithm. Two quantities can be assumed as the known quantities, the total energy and the angular momentum, which are the constants of motion. We can describe the motion of the comet in the xy plane and choose $x_0 = r_{\max}$, $v_{x0} = 0$, $y_0 = 0$, and $v_{y0} = v_{\min}$. From well-known results we have that $r_{\max} = 5.28 \times 10^{12}$ m and $v_{\min} = 9.13 \times 10^2$ m/s. Let us apply the velocity version of the Verlet algorithm to this problem. Then we have

$$x^{(k+1)} = x^{(k)} + \tau v_x^{(k)} + \frac{\tau^2}{2} g_x^{(k)}, \tag{8.45}$$

$$v_x^{(k+1)} = v_x^{(k)} + \frac{\tau}{2}\left[g_x^{(k+1)} + g_x^{(k)}\right], \tag{8.46}$$

$$y^{(k+1)} = y^{(k)} + \tau v_y^{(k)} + \frac{\tau^2}{2} g_y^{(k)}, \tag{8.47}$$

$$v_y^{(k+1)} = v_y^{(k)} + \frac{\tau}{2}\left[g_y^{(k+1)} + g_y^{(k)}\right], \tag{8.48}$$

where the time-step index is given in parentheses as superscripts in order to distinguish it from the x-component or y-component index. The acceleration components are given by

$$g_x = -\kappa \frac{x}{r^3}, \tag{8.49}$$

$$g_y = -\kappa \frac{y}{r^3}, \tag{8.50}$$

with $r = \sqrt{x^2 + y^2}$ and $\kappa = GM$. We can use more specific units in the numerical calculations, for example, 76 years as the time unit and the semimajor axis of the orbital $a = 2.68 \times 10^{12}$ m as the length unit. Then we have $r_{\max} = 1.97$, $v_{\min} = 0.816$, and $\kappa = 39.5$. The following program is the implementation of the algorithm outlined above for Halley's comet.

```
// An example to study the time-dependent position and
// velocity of Halley's comet via the Verlet algorithm.

import java.lang.*;
public class Comet {
  static final int n = 20000, m = 200;
  public static void main(String argv[]) {
    double t[] = new double [n];
    double x[] = new double [n];
    double y[] = new double [n];
    double r[] = new double [n];
    double vx[] = new double [n];
    double vy[] = new double [n];
    double gx[] = new double [n];
    double gy[] = new double [n];
    double h = 2.0/(n-1), h2 = h*h/2, k = 39.478428;
```

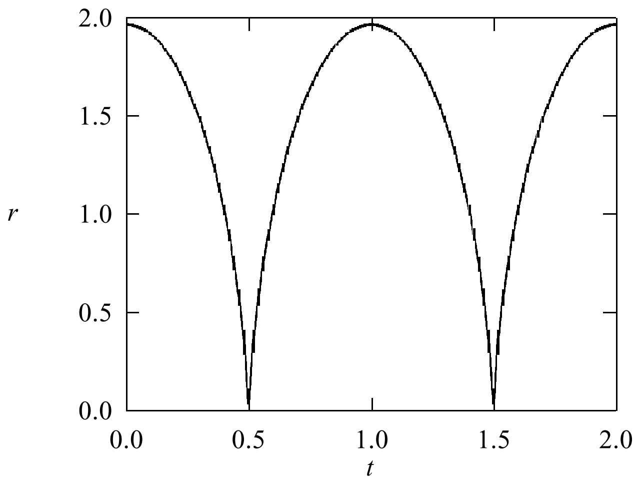


Fig. 8.1 The distance between Halley's comet and the Sun calculated with the program given in the text. The period of the comet is used as the unit of time, and the semimajor axis of the orbit is used as the unit of distance.

```
// Initialization of the problem
   t[0] = 0;
   x[0] = 1.966843;
   y[0] = 0;
   r[0] = x[0];
   vx[0] = 0;
   vy[0] = 0.815795;
   gx[0] = -k/(r[0]*r[0]);
   gy[0] = 0;

// Verlet algorithm for the position and velocity
   for (int i=0; i<n-1; ++i) {
     t[i+1]   = h*(i+1);
     x[i+1]   = x[i]+h*vx[i]+h2*gx[i];
     y[i+1]   = y[i]+h*vy[i]+h2*gy[i];
     double r2 = x[i+1]*x[i+1]+y[i+1]*y[i+1];
     r[i+1] = Math.sqrt(r2);
     double r3 = r2*r[i+1];
     gx[i+1] = -k*x[i+1]/r3;
     gy[i+1] = -k*y[i+1]/r3;
     vx[i+1]= vx[i]+h*(gx[i+1]+gx[i])/2;
     vy[i+1]= vy[i]+h*(gy[i+1]+gy[i])/2;
   }
   for (int i=0; i<n-m; i+=m) {
     System.out.println(t[i]);
     System.out.println(r[i]);
     System.out.println();
   }
  }
}
```

In Fig. 8.1, we show the result for the distance between Halley's comet and the Sun calculated using the above program.

The accuracy of the velocity version of the Verlet algorithm is reasonable in practice; it is usually accurate enough because the corresponding physical quantities are rescaled according to, for example, the temperature of the system in most molecular dynamics simulations. The accumulated errors are thus removed when the rescaling is applied. More accurate algorithms, such as the Gear predictor–corrector algorithm, will be discussed later in this chapter.

8.4 Structure of atomic clusters

One of the simplest applications of the Verlet algorithm is in the study of an isolated collection of particles. For an isolated system, there is a significant difference between a small system and a large system. For a small system, the ergodic assumption of statistical mechanics fails and the system may never reach the so-called equilibrium state. However, some parallel conclusions can be drawn on the thermodynamics of small systems against infinite systems (Gross, 2001). For a very large system, standard statistical mechanics applies, even if it is isolated from the environment; the interactions among the particles cause the exchange of energies and drive the system to equilibrium. Very small clusters with just a few particles usually behave like molecules (Sugano and Koizumi, 1998). What is unclear is the behavior of a cluster of an intermediate size, say, about 100 atoms.

In this section, we demonstrate the application of the velocity version of the Verlet algorithm in determining the structure and dynamics of clusters of an intermediate size. We will assume that the system consists of N atoms that interact with each other through the Lennard–Jones potential

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \tag{8.51}$$

where r is the distance between the two particles, and ε and σ are the system-dependent parameters. The force exerted on the ith particle is therefore given by

$$\mathbf{f}_i = \frac{48\varepsilon}{\sigma^2}\sum_{j\neq i}^{N}(\mathbf{r}_i - \mathbf{r}_j)\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]. \tag{8.52}$$

In order to simplify the notation, ε is usually used as the unit of energy, ε/k_B as the unit of temperature, and σ as the unit of length. Then the unit of time is given by $\sqrt{m\sigma^2/48\varepsilon}$. Newton's equation for each particle then becomes dimensionless,

$$\frac{d^2\mathbf{r}_i}{dt^2} = \mathbf{g}_i = \sum_{j\neq i}^{N}(\mathbf{r}_i - \mathbf{r}_j)\left(\frac{1}{r_{ij}^{14}} - \frac{1}{2r_{ij}^{8}}\right). \tag{8.53}$$

After discretization with the Verlet algorithm, we have

$$\mathbf{r}_i^{(k+1)} = \mathbf{r}_i^{(k)} + \tau \mathbf{v}_i^{(k)} + \frac{\tau^2}{2}\mathbf{g}_i^{(k)}, \tag{8.54}$$

$$\mathbf{v}_i^{(k+1)} = \mathbf{v}_i^{(k)} + \frac{\tau}{2}\left[\mathbf{g}_i^{(k+1)} + \mathbf{g}_i^{(k)}\right]. \tag{8.55}$$

The update of the velocity is usually done in two steps in practice. When $\mathbf{g}_i^{(k)}$ is evaluated, the velocity is partially updated with

$$\mathbf{v}_i^{(k+1/2)} = \mathbf{v}_i^{(k)} + \frac{\tau}{2}\mathbf{g}_i^{(k)} \tag{8.56}$$

and then updated again when $\mathbf{g}_i^{(k+1)}$ becomes available after the coordinate is updated, with

$$\mathbf{v}_i^{(k+1)} = \mathbf{v}_i^{(k+1/2)} + \frac{\tau}{2}\mathbf{g}_i^{(k+1)}. \tag{8.57}$$

This is equivalent to the one-step update but more economical, because it does not require storage of the acceleration of the previous time step. We can then simulate the structure and dynamics of the cluster starting from a given initial position and velocity for each particle. However, several aspects still need special care. The initial positions of the particles are usually taken as the lattice points on a closely packed structure, for example, the face-centered cubic structure. What we want to avoid is the breaking up of the system in the first few time steps, which happens if some particles are too close to each other. Another way to set up a relatively stable initial cluster is to cut out a piece from a bulk simulation. The corresponding bulk simulation is achieved with the application of the periodic boundary condition. The initial velocities of the particles should also be assigned reasonably. A common practice is to assign velocities from the Maxwell distribution

$$\mathcal{W}(v_x) \propto e^{-mv_x^2/2k_\mathrm{B}T}, \tag{8.58}$$

which can be achieved numerically quite easily with the availability of Gaussian random numbers. The variance in the Maxwell distribution is $\sqrt{k_\mathrm{B}T/m}$ for each velocity component. For example, the following method returns the Maxwell distribution of the velocity for a given temperature.

```
// Method to draw initial velocities from the Maxwell
// distribution for a given mass m, temperature T, and
// paritcle number N.

  public static double[] maxwell(doubule m, double T,
    int N) {
    int nv = 3*N;
    int ng = nv-6;
    double v[] = new double[nv];
    double r[] = new double[2];

// Assign a Gaussian number to each velocity component

    for (int i=0; i<nv-1; i+=2){
      r = rang();
      g[i]   = r[0];
      g[i+1]   = r[1];
    }
// Scale the velocity to satisfy the partition theorem
    double ek = 0;
    for (int i=0; i<nv; ++i) ek += v[i]*v[i];
    double vs = Math.sqrt(m*ek*nv/(ng*T));
    for (int i=0; i<nv; ++i) v[i] /= vs;
    return v;
  }
```

However, we always have difficulty in defining a temperature if the system is very small, that is, where the thermodynamic limit is not applicable. In practice, we can still call a quantity associated with the kinetic energy of the system the *temperature*, which is basically a measure of the kinetic energy at each time step,

$$E_{\mathrm{K}} = \frac{G}{2} k_{\mathrm{B}} T, \tag{8.59}$$

where G is the total number of independent degrees of freedom in the system. Note that now T is not necessarily a constant and that G is equal to $3N - 6$, with N being the number of particles in the system, because we have to remove the center-of-mass motion and rotation around the center of mass when we study the structure and dynamics of the isolated cluster.

The total energy of the system is given by

$$E = \sum_{i=1}^{N} \frac{m_i v_i^2}{2} + \sum_{i>j=1}^{N} V(r_{ij}), \tag{8.60}$$

where the kinetic energy E_{K} is given by the first term on the right-hand side and the potential energy E_{P} by the second.

One remarkable effect observed in the simulations of finite clusters is that there is a temperature region where the system fluctuates from a liquid state to a solid state over time. A phase is identified as solid if the particles in the system vibrate only around their equilibrium positions; otherwise they are in a liquid phase. Simulations of clusters have revealed some very interesting phenomena that are unique to clusters of intermediate size (with $N \simeq 100$). We have discussed how to analyze the structural and dynamical information of a collection of particles in Sections 8.1 and 8.2. One can evaluate the mean square of the displacement of each particle. For the solid state, $\Delta^2(t)$ is relatively small and does not grow with time, and the particles are nondiffusive but oscillatory. For the liquid state, $\Delta^2(t)$ grows with time close to the linear relation $\Delta^2(t) = 6Dt + \Delta^2(0)$, where D is the self-diffusion coefficient and $\Delta^2(0)$ is a constant. In a small system, this relation in time may not be exactly linear. One can also measure the specific structure of the cluster from the pair-distribution function $g(r)$ and the orientational correlation function of the bonds. The temperature (or total kinetic energy) can be gradually changed to cool or heat the system.

The results from simulations on clusters of about 100 particles show that the clusters can fluctuate from a group of solid states with lower energy levels to a group of liquid states that lie at higher energy levels in a specific temperature region, with $k_B T$ in the same order as the energy that separates the two groups. This is not expected because the higher-energy-level liquid states are not supposed to have an observable lifetime. However, the statistical mechanics may not be accurate here, because the number of particles is relatively small. More detailed simulations also reveal that the melting of the cluster usually starts from the surface. Interested readers can find more discussions of this topic in Matsuoka *et al.* (1992) and Kunz and Berry (1993; 1994).

8.5 The Gear predictor--corrector method

We discussed multistep predictor–corrector methods in Chapter 4 when solving initial-value problems. Another type of predictor–corrector method uses the truncated Taylor expansion of the function and its derivatives as a prediction and then evaluates the function and its derivatives at the same time step with a correction given from the restriction of the differential equation. This multivalue predictor–corrector scheme was developed by Nordsieck and Gear. Details of the derivations can be found in Gear (1971).

We will take a first-order differential equation

$$\frac{d\mathbf{r}}{dt} = \mathbf{f}(\mathbf{r}, t) \tag{8.61}$$

as an example and then generalize the method to other types of initial-value problems, such as Newton's equation. For simplicity, we will introduce the rescaled quantities

$$\mathbf{r}^{(l)} = \frac{\tau^l}{l!}\frac{d^l\mathbf{r}}{dt^l}, \tag{8.62}$$

with $l = 0, 1, 2, \ldots$. Note that $\mathbf{r}^{(0)} = \mathbf{r}$. Now if we define a vector

$$\mathbf{x} = (\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \mathbf{r}^{(2)}, \ldots), \tag{8.63}$$

we can obtain the predicted value of $\mathbf{x}$ in the next time step $\mathbf{x}_{k+1}$, with $t_{k+1} = (k+1)\tau$, from that of the current time step $\mathbf{x}_k$, with $t_k = k\tau$, from the Taylor expansion for each component of $\mathbf{x}_{k+1}$: that is,

$$\mathbf{x}_{k+1} = \mathbf{B}\mathbf{x}_k, \tag{8.64}$$

where $\mathbf{B}$ is the coefficient matrix from the Taylor expansion. One can easily show that $\mathbf{B}$ is an upper triangular matrix with unit diagonal and first row elements, with the other elements given by

$$B_{ij} = \binom{j-1}{i} = \frac{(j-1)!}{i!\,(j-i-1)!}. \tag{8.65}$$

Note that the dimension of $\mathbf{B}$ is $(n+1) \times (n+1)$, that is, $\mathbf{x}$ is a vector of dimension $n+1$, if the Taylor expansion is carried out up to the term of $d^n\mathbf{r}/dt^n$.

The correction in the Gear method is performed with the application of the differential equation under study. The difference between the predicted value of the first-order derivative $\mathbf{r}^{(1)}_{k+1}$ and the velocity field $\tau\mathbf{f}_{k+1}$ is

$$\delta\mathbf{r}^{(1)}_{k+1} = \tau\mathbf{f}_{k+1} - \mathbf{r}^{(1)}_{k+1}, \tag{8.66}$$

which would be zero if the exact solution were obtained with $\tau \to 0$. Note that $\mathbf{r}^{(1)}_{k+1}$ in the above equation is from the prediction of the Taylor expansion. So if we combine the prediction and the correction in one expression, we have

$$\mathbf{x}_{k+1} = \mathbf{B}\mathbf{x}_k + \mathbf{C}\delta\mathbf{x}_{k+1}, \tag{8.67}$$

where $\delta\mathbf{x}_{k+1}$ has only one nonzero component, $\delta\mathbf{r}^{(1)}_{k+1}$, and $\mathbf{C}$ is the correction coefficient matrix, which is diagonal with zero off-diagonal elements and nonzero diagonal elements $C_{ii} = c_i$ for $i = 0, 1, \ldots, n$. Here c_i are obtained by solving a matrix eigenvalue problem involving $\mathbf{B}$ and the gradient of $\delta\mathbf{x}_{k+1}$ with respect to $\mathbf{x}_{k+1}$. It is straightforward to solve for c_i if the truncation in the Taylor expansion is not very high. Interested readers can find the derivation in Gear (1971), with a detailed table listing up to the fourth-order differential equations (Gear 1971, p. 154). The most commonly used Gear scheme for the first-order differential equation is the fifth-order Gear algorithm (with the Taylor expansion carried out up to $n = 5$), with $c_0 = 95/288$, $c_1 = 1$, $c_2 = 25/24$, $c_3 = 35/72$, $c_4 = 5/48$, and $c_5 = 1/120$.

We should point out that the above procedure is not unique to first-order differential equations. The predictor part is identical for any higher-order differential equation. The only change one needs to make is the correction, which is the result of the difference between the prediction and the solution restricted by the equation. In molecular dynamics, we are interested in the solution of Newton's equation, which is a second-order differential equation with

$$\frac{d^2\mathbf{r}}{dt^2} = \frac{\mathbf{f}(\mathbf{r}, t)}{m}, \tag{8.68}$$

where $\mathbf{f}$ is the force on a specific particle of mass m. We can still formally express the algorithm as

$$\mathbf{x}_{k+1} = \mathbf{B}\mathbf{x}_k + \mathbf{C}\delta\mathbf{x}_{k+1}, \tag{8.69}$$

where $\delta\mathbf{x}_{k+1}$ also has only one nonzero element

$$\delta\mathbf{r}^{(2)}_{k+1} = \tau^2\mathbf{f}_{k+1}/2m - \mathbf{r}^{(2)}_{k+1}, \tag{8.70}$$

which provides the corrections to all the components of $\mathbf{x}$. Similarly, $\mathbf{r}^{(2)}_{k+1}$ in the above equation is from the Taylor expansion. The corrector coefficient matrix is still diagonal with zero off-diagonal elements and nonzero diagonal elements $C_{ii} = c_i$ for $i = 0, 1, \ldots, n$. Here c_i for the second-order differential equation have also been worked out by Gear (1971, p. 154). The most commonly used Gear scheme for the second-order differential equation is still the fifth-order Gear algorithm (with the Taylor expansion carried out up to $n = 5$), with $c_0 = 3/20$, $c_1 = 251/360, c_2 = 1, c_3 = 11/18, c_4 = 1/6$, and $c_5 = 1/60$. The values of these coefficients are obtained with the assumption that the force in Eq. (8.68) does not have an explicit dependence on the velocity, that is, the first-order derivative of $\mathbf{r}$. If it does, $c_0 = 3/20$ needs to be changed to $c_0 = 3/16$ (Allen and Tildesley, 1987, pp. 340–2).

8.6 Constant pressure, temperature, and bond length

Several issues still need to be addressed when we want to compare the simulation results with the experimental measurements. For example, environmental parameters, such as temperature or pressure, are the result of thermal or mechanical contact of the system with the environment or the result of the equilibrium of the system. For a macroscopic system, we deal with average quantities by means of statistical mechanics, so it is highly desirable for the simulation environment to be as close as possible to a specific ensemble. The scheme adopted in Section 8.4 for atomic cluster systems is closely related to the microcanonical ensemble, which has the total energy of the system conserved. It is important that we also be able to find ways to deal with constant-temperature and/or constant-pressure conditions. We should emphasize that there have been many efforts to model realistic systems with simulation boxes by introducing some specific procedures. However, all these procedures do not introduce any new concepts in physics. They are merely numerical techniques to make the simulation boxes as close as possible to the physical systems under study.

Constant pressure: the Andersen scheme

The scheme for dealing with a constant-pressure environment was devised by Andersen (1980) with the introduction of an environmental variable, the instantaneous volume of the system, in the effective Lagrangian. When the Lagrange equation is applied to the Lagrangian, the equations of motion for the coordinates of the particles and the volume result. The constant pressure is then a result of the zero average of the second-order time derivative of the volume.

The effective Lagrangian of Andersen (1980) is given by

$$\mathcal{L} = \sum_{i=1}^{N} \frac{m_i}{2} L^2 \dot{\mathbf{x}}_i^2 - \sum_{i>j} V(Lx_{ij}) + \frac{M}{2}\dot{\Omega}^2 - P_0\Omega, \tag{8.71}$$

where the last two terms are added to deal with the constant pressure from the environment. The parameter M can be viewed here as an effective inertia associated with the expansion and contraction of the volume Ω, and P_0 is the external pressure, which introduces a potential energy $P_0\Omega$ to the system under the assumption that the system is in contact with the constant-pressure environment. The coordinate of each particle $\mathbf{r}_i$ is rescaled with the dimension of the simulation box, $L = \Omega^{1/3}$, because the distance between any two particles changes with L and the coordinates independent of the volume are then given by

$$\mathbf{x}_i = \mathbf{r}_i / L, \tag{8.72}$$

which are not directly related to the changing volume. Note that this effective Lagrangian is not the result of any new physical principles or concepts but is merely a method of modeling the effect of the environment in realistic systems. Now if we apply the Lagrange equation to the above Lagrangian, we obtain the equations of motion for the particles and the volume Ω,

$$\ddot{\mathbf{x}}_i = \frac{\mathbf{g}_i}{L} - \frac{2\dot{\Omega}}{3\Omega}\dot{\mathbf{x}}_i, \tag{8.73}$$

$$\ddot{\Omega} = \frac{P - P_0}{M}, \tag{8.74}$$

where $\mathbf{g}_i = \mathbf{f}_i / m_i$ and P is given by

$$P = \frac{1}{3\Omega}\left(\sum_i m_i L^2 \dot{\mathbf{x}}_i^2 + \sum_{i>j}^{N} \mathbf{r}_{ij} \cdot \mathbf{f}_{ij}\right), \tag{8.75}$$

which can be interpreted as the instantaneous pressure of the system and has a constant average P_0, because $\langle \ddot{\Omega} \rangle \equiv 0$.

After we have the effective equations of motion, the algorithm can be worked out quite easily. We will use

$$\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N), \tag{8.76}$$

$$\mathbf{G} = (\mathbf{f}_1/m_1, \mathbf{f}_2/m_2, \cdots, \mathbf{f}_N/m_N), \tag{8.77}$$

to simplify the notation. If we apply the velocity version of the Verlet algorithm, the difference equations for the volume and the rescaled coordinates are given by

$$\Omega_{k+1} = \Omega_k + \tau\dot{\Omega}_k + \frac{\tau^2(P_k - P_0)}{2M}, \tag{8.78}$$

$$\mathbf{X}_{k+1} = \left(1 - \frac{\tau^2\dot{\Omega}_k}{2\Omega_k}\right)\mathbf{X}_k + \tau\dot{\mathbf{X}}_k + \frac{\tau^2\mathbf{G}_k}{2L_k}, \tag{8.79}$$

$$\dot{\Omega}_{k+1} = \dot{\Omega}_{k+1/2} + \frac{\tau(P_{k+1} - P_0)}{2M}, \tag{8.80}$$

$$\dot{\mathbf{X}}_{k+1} = \left(1 - \frac{\tau\dot{\Omega}_{k+1}}{2\Omega_{k+1}}\right)\dot{\mathbf{X}}_{k+1/2} + \frac{\tau\mathbf{G}_{k+1}}{2L_{k+1}}, \tag{8.81}$$

where the values with index $k + 1/2$ are intermediate values before the pressure and force are updated, with

$$\dot{\Omega}_{k+1/2} = \dot{\Omega}_k + \frac{\tau(P_k - P_0)}{2M}, \tag{8.82}$$

$$\dot{\mathbf{X}}_{k+1/2} = \left(1 - \frac{\tau\dot{\Omega}_k}{2\Omega_k}\right)\dot{\mathbf{X}}_k + \frac{\tau\mathbf{G}_k}{2L_{k+1}}, \tag{8.83}$$

which are usually evaluated immediately after the volume and the coordinates are updated.

In practice, we first need to set up the initial positions and velocities of the particles and the initial volume and its time derivative. The initial volume is determined from the given particle number and density, and its initial time derivative is usually set to be zero. The initial coordinates of the particles are usually arranged on a densely packed lattice, for example, a face-centered cubic lattice, and the initial velocities are usually drawn from the Maxwell distribution. One should test the program with different Ms in order to find the value of M that minimizes the fluctuation.

A generalization of the Andersen constant-pressure scheme was introduced by Parrinello and Rahman (1980; 1981) to allow the shape of the simulation box to change as well. This generalization is important in the study of the structural phase transition. With the shape of the simulation box allowed to vary, the particles can easily move to the lattice points of the structure with the lowest free energy. The idea of Parrinello and Rahman can be summarized in the Lagrangian

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N} m_i \dot{\mathbf{x}}_i^{\mathrm{T}} \mathbf{B} \dot{\mathbf{x}}_i - \sum_{i>j}^{N} V(x_{ij}) + \frac{M}{2}\sum_{i,j=1}^{3} \dot{A}_{ij}^2 - P_0 \Omega, \tag{8.84}$$

where $\mathbf{y}_i$ is the coordinate of the ith particle in the vector representation of the simulation box, $\Omega = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$, with

$$\mathbf{r}_i = x_i^{(1)}\mathbf{a} + x_i^{(2)}\mathbf{b} + x_i^{(3)}\mathbf{c}. \tag{8.85}$$

Here $\mathbf{A}$ is the matrix representation of $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ in Cartesian coordinates and $\mathbf{B} = \mathbf{A}^{\mathrm{T}}\mathbf{A}$. Instead of a single variable Ω, there are nine variables A_{ij}, with i, $j = 1, 2, 3$; this allows both the volume size and the shape of the simulation box to change. The equations of motion for $\mathbf{x}_i$ and A_{ij} can be derived by applying the Lagrange equation to the above Lagrangian. An external stress can also be included in such a procedure (Parrinello and Rahman, 1981).

Constant temperature: the Nosé scheme

The constant-pressure scheme discussed above is usually performed with an ad hoc constant-temperature constraint, which is done by rescaling the velocities during the simulation to ensure the relation between the total kinetic energy and the desired temperature in the canonical ensemble.

This rescaling can be shown to be equivalent to a force constraint up to first order in the time step τ. The constraint method for the constant-temperature simulation is achieved by introducing an artificial force $-\eta\mathbf{p}_i$, which is similar to a frictional force if η is greater than zero or to a heating process if η is less than zero. The equations of motion are modified under this force to

$$\dot{\mathbf{r}}_i = \frac{\mathbf{p}_i}{m_i}, \tag{8.86}$$

$$\dot{\mathbf{v}}_i = \frac{\mathbf{f}_i}{m_i} - \eta \mathbf{v}_i, \tag{8.87}$$

where $\mathbf{p}_i$ is the momentum of the ith particle and η is the constraint parameter, which can be obtained from the relevant Lagrange multiplier (Evans *et al.*, 1983) in the Lagrange equations with

$$\eta = \frac{2\sum \mathbf{f}_i \cdot \mathbf{v}_i}{\sum m_i \mathbf{v}_i^2} = -\frac{dE_{\mathrm{P}}/dt}{Gk_{\mathrm{B}}T}, \tag{8.88}$$

which can be evaluated at every time step. Here G is the total number of degrees of freedom of the system, and E_{P} is the total potential energy. So one can simulate the canonical ensemble averages from the equations for $\mathbf{r}_i$ and $\mathbf{v}_i$ given in Eqs. (8.86) and (8.87).

The most popular constant-temperature scheme is that of Nosé (1984a; 1984b), who introduced a fictitious dynamical variable to take the constant-temperature environment into account. The idea is very similar to that of Andersen for the constant-pressure case. In fact, one can put both fictitious variables together to have simulations for constant pressure and constant temperature together. Here we will briefly discuss the Nosé scheme.

We can introduce a rescaled effective Lagrangian

$$\mathcal{L} = \sum_{i=1}^{N} \frac{m_i}{2} s^2 \dot{\mathbf{x}}_i^2 - \sum_{i>j} V(x_{ij}) + \frac{m_s}{2} v_s^2 - Gk_{\mathrm{B}}T \ln s, \tag{8.89}$$

where s and v_s are the coordinate and velocity of an introduced fictitious variable that rescales the time and the kinetic energy in order to have the constraint of the canonical ensemble satisfied. The rescaling is achieved by replacing the time element dt with dt/s and holding the coordinates unchanged, that is, $\mathbf{x}_i = \mathbf{r}_i$. The velocity is rescaled with time: $\dot{\mathbf{r}}_i = s\dot{\mathbf{x}}_i$. We can then obtain the equation of motion for the coordinate $\mathbf{x}_i$ and the variable s by applying the Lagrange equation. Hoover (1985; 1999) showed that the Nosé Lagrangian leads to a set of equations very similar to the result of the constraint force scheme discussed at the beginning of this subsection. The Nosé equations of motion are given in the Hoover version by

$$\dot{\mathbf{r}}_i = \mathbf{v}_i, \tag{8.90}$$

$$\dot{\mathbf{v}}_i = \frac{\mathbf{f}_i}{m_i} - \eta \mathbf{v}_i, \tag{8.91}$$

where η is given in a differential form,

$$\dot{\eta} = \frac{1}{m_s}\left(\sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{m_i} - Gk_{\mathrm{B}}T\right), \tag{8.92}$$

and the original variable s, introduced by Nosé, is related to η by

$$s = s_0 e^{\eta(t-t_0)}, \tag{8.93}$$

with s_0 as the initial value of s at $t = t_0$. We can discretize the above equation set easily with either the Verlet algorithm or one of the Gear schemes. Note that the behavior of the parameter s is no longer directly related to the simulation; it is merely a parameter Nosé introduced to accomplish the microscopic processes happening in the constant-temperature environment. We can also combine the Andersen constant-pressure scheme with the Nosé constant-temperature scheme in a single effective Lagrangian

$$\mathcal{L} = \sum_{i=1}^{N} \frac{m_i}{2} s^2 L^2 \dot{\mathbf{x}}_i^2 - \sum_{i>j} V(Lx_{ij}) + \frac{m_s}{2}\dot{s}^2 - Gk_\mathrm{B}T \ln s + \frac{M}{2}\dot{\Omega}^2 - P_0\Omega, \tag{8.94}$$

which is worked out in detail in the original work of Nosé (1984a; 1984b). Another constant-temperature scheme was introduced by Berendsen *et al.* (1984) with the parameter η given by

$$\eta = \frac{1}{m_s}\left(\sum_{i=1}^{N} m_i \mathbf{v}_i^2 - Gk_\mathrm{B}T\right), \tag{8.95}$$

which can be interpreted as a similar form of the constraint that differs from the Hoover–Nosé form in the choice of η. For a review on the subject, see Nosé (1991).

Constant bond length

Another issue we have to deal with in practice is that for large molecular systems, such as biopolymers, the bond length of a pair of nearest neighbors does not change very much even though the angle between a pair of nearest bonds does. If we want to obtain accurate simulation results, we have to choose a time step much smaller than the period of the vibration of each pair of atoms. This costs a lot of computing time and might exclude the applicability of the simulation to more complicated systems, such as biopolymers.

A procedure commonly known as the SHAKE algorithm (Ryckaert, Ciccotti, and Berendsen, 1977; van Gunsteren and Berendsen, 1977) was introduced to deal with the constraint on the distance between a pair of particles in the system. The idea of this procedure is to adjust each pair of particles iteratively to have

$$\left(\mathbf{r}_{ij}^2 - d_{ij}^2\right)/d_{ij}^2 \le \Delta \tag{8.96}$$

in each time step. Here d_{ij} is the distance constraint between the ith and jth particles and Δ is the tolerance in the simulation. The adjustment of the position of each particle is performed after each time step of the molecular dynamics simulation. Assume that we are working on a specific pair of particles and for the lth constraint and that we would like to have

$$(\mathbf{r}_{ij} + \delta\mathbf{r}_{ij})^2 - d_{ij}^2 = 0, \tag{8.97}$$

where $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ is the new position vector difference after a molecular time step starting from $\mathbf{r}_{ij}^{(0)}$ and the adjustments for the $l - 1$ constraints have been completed. Here $\delta\mathbf{r}_{ij} = \delta\mathbf{r}_j - \delta\mathbf{r}_i$ is the total amount of adjustment needed for both particles.

One can show, in conjunction with the Verlet algorithm, that the adjustments needed are given by

$$m_i \delta\mathbf{r}_i = -g_{ij}\mathbf{r}_{ij}^{(0)} = -m_j \delta\mathbf{r}_j, \tag{8.98}$$

with g_{ij} as a parameter to be determined. The center of mass of these two particles remains the same during the adjustment.

If we substitute $\delta\mathbf{r}_i$ and $\delta\mathbf{r}_j$ given in Eq. (8.98), we obtain

$$\left(r_{ij}^{(0)}\right)^2 g_{ij}^2 + 2\mu_{ij}\mathbf{r}_{ij}^{(0)} \cdot \mathbf{r}_{ij} g_{ij} + \mu_{ij}^2 \left(r_{ij}^2 - d^2\right) = 0, \tag{8.99}$$

where $\mu_{ij} = m_i m_j/(m_i + m_j)$ is the reduced mass of the two particles. If we keep only the linear term in g_{ij}, we have

$$g_{ij} = \frac{\mu_{ij}}{2\mathbf{r}_{ij}^{(0)} \cdot \mathbf{r}_{ij}} \left(d_{ij}^2 - r_{ij}^2\right), \tag{8.100}$$

which is reasonable, because g_{ij} is a small number during the simulation. More importantly, by the end of the iteration, all the constraints will be satisfied as well; all g_{ij} go to zero at the convergence. Equation (8.100) is used to estimate each g_{ij} for each constraint in each iteration. After one has the estimate of g_{ij} for each constraint, the positions of the relevant particles are all adjusted. The adjustments have to be performed several times until the convergence is reached. For more details on the algorithm, see Ryckaert *et al.* (1977).

This procedure has been used in the simulation of chain-like systems as well as of proteins and nucleic acids. Interested readers can find some detailed discussions on the dynamics of proteins and nucleic acids in McCammon and Harvey (1987).

8.7 Structure and dynamics of real materials

In this section, we will discuss some typical methods used to extract information about the structure and dynamics of real materials in molecular dynamics simulations.

A numerical simulation of a specific material starts with a determination of the interaction potential in the system. In most cases, the interaction potential is formulated in a parameterized form, which is usually determined separately from the available experimental data, first principles calculations, and condition of the system under study. The accuracy of the interaction potential determines the validity of the simulation results. Accurate model potentials have been developed for many realistic materials, for example, the Au (100) surface (Ercolessi, Tosatti, and Parrinello, 1986) and Si_3N_4 ceramics (Vashishta *et al.*, 1995). In the next section, we will discuss an ab initio molecular dynamics scheme in which the interaction potential is obtained by calculating the electronic structure of the system at each particle configuration.

We then need to set up a simulation box under the periodic boundary condition. Because most experiments are performed in a constant-pressure environment, we typically use the constant-pressure scheme developed by Andersen (1980) or its generalization (Parrinello and Rahman, 1980; 1981). The size of the simulation box has to be decided together with the available computing resources and the accuracy required for the quantities to be evaluated. The initial positions of the particles are usually assigned at the lattice points of a closely packed structure, for example, a face-centered cubic structure. The initial velocities of the particles are drawn from the Maxwell distribution for a given temperature. The temperature can be changed by rescaling the velocities. This is extremely useful in the study of phase transitions with varying temperature, such as the transition between different lattice structures, glass transition under quenching, or liquid–solid transition when the system is cooled down slowly. The advantage of simulation over actual experiment also shows up when we want to observe some behavior that is not achievable experimentally due to the limitations of the technique or equipment. For example, the glass transition in the Lennard–Jones system is observed in molecular dynamics simulations but not in the experiments for liquid Ar, because the necessary quenching rate is so high that it is impossible to achieve it experimentally.

Studying the dynamics of different materials requires a more general time-dependent density–density correlation function

$$C(\mathbf{r}, \mathbf{r}'; t) = \langle \hat{\rho}(\mathbf{r} + \mathbf{r}', t)\hat{\rho}(\mathbf{r}', 0)\rangle, \tag{8.101}$$

with the time-dependent density operator given by

$$\hat{\rho}(\mathbf{r}, t) = \sum_{i=1}^{N} \delta[\mathbf{r} - \mathbf{r}_i(t)]. \tag{8.102}$$

If the system is homogeneous, we can integrate out $\mathbf{r}'$ in the time-dependent density–density correlation function to reach the van Hove time-dependent distribution function (van Hove, 1954)

$$G(\mathbf{r}, t) = \frac{1}{\rho N}\left\langle \sum_{i,j}^{N} \delta\{\mathbf{r} - [\mathbf{r}_i(t) - \mathbf{r}_j(0)]\}\right\rangle. \tag{8.103}$$

The *dynamical structure factor* measured in an experiment, for example, neutron scattering, is given by the Fourier transform of $G(\mathbf{r}, t)$ as

$$S(\mathbf{k}, \omega) = \frac{\rho}{2\pi}\int e^{i(\omega t - \mathbf{k}\cdot\mathbf{r})} G(\mathbf{r}, t)\, d\mathbf{r}\, dt. \tag{8.104}$$

The above equation reduces to the static case with

$$S(k) - 1 = 4\pi\rho \int \frac{\sin kr}{kr}[g(r) - 1]r^2\, dr \tag{8.105}$$

if we realize that

$$G(\mathbf{r},0) = g(\mathbf{r}) + \frac{\delta(\mathbf{r})}{\rho} \tag{8.106}$$

and

$$S(\mathbf{k}) = \int_{-\infty}^{\infty} S(\mathbf{k},\omega)\,d\omega, \tag{8.107}$$

where $g(\mathbf{r})$ is the pair distribution discussed earlier in this chapter and $S(k)$ is the angular average of $S(\mathbf{k})$. Here $G(\mathbf{r},t)$ can be interpreted as the probability of observing one of the particles at $\mathbf{r}$ at time t if a particle was observed at $\mathbf{r} = 0$ at $t = 0$. This leads to the numerical evaluation of $G(r,t)$, which is the angular average of $G(\mathbf{r},t)$. If we write $G(r,t)$ in two parts,

$$G(r,t) = G_{\mathrm{s}}(r,t) + G_{\mathrm{d}}(r,t), \tag{8.108}$$

with $G_{\mathrm{s}}(r,t)$ the probability of observing the same particle that was at $r = 0$ at $t = 0$, and $G_{\mathrm{d}}(r,t)$ the probability of observing other particles, we have

$$G_{\mathrm{d}}(r,t) \simeq \frac{1}{\rho}\frac{\langle \Delta N(r,\Delta r;t)\rangle}{\Delta\Omega(r,\Delta r)}, \tag{8.109}$$

where $\Delta\Omega(r,\Delta r) \simeq 4\pi r^2 \Delta r$ is the volume of a spherical shell with radius r and thickness Δr, and $\Delta N(r,\Delta r;t)$ is the number of particles in the spherical shell at time t. The position of each particle at $t = 0$ is chosen as the origin in the evaluation of $G_{\mathrm{d}}(r,t)$ and the average is taken over all the particles in the system. Note that this is different from the evaluation of $g(r)$, in which we always select a particle position as the origin and take the average over time. We can also take the average over all the particles in the evaluation of $g(r)$. Here $G_{\mathrm{s}}(r,t)$ can be evaluated in a similar fashion. Because $G_{\mathrm{s}}(r,t)$ represents the probability for a particle to be at a distance r at time t from its original position at $t = 0$, we can introduce

$$\Delta^{2n}(t) = \frac{1}{N}\left\langle \sum_{i=1}^{N} [\mathbf{r}_i(t) - \mathbf{r}_i(0)]^{2n} \right\rangle = \int r^{2n} G_{\mathrm{s}}(r,t)\,d\mathbf{r} \tag{8.110}$$

in the evaluation of the diffusion coefficient with $n = 1$. The diffusion coefficient can also be evaluated from the autocorrelation function

$$c(t) = \langle \mathbf{v}(t)\cdot\mathbf{v}(0)\rangle = \frac{1}{N}\sum_{i=1}^{N} [\mathbf{v}_i(t)\cdot\mathbf{v}_i(0)], \tag{8.111}$$

with

$$D = \frac{1}{3c(0)}\int_0^{\infty} c(t)\,dt, \tag{8.112}$$

because the velocity of each particle at each time step $\mathbf{v}_i(t)$ is known from the simulation. The velocity correlation function can also be used to obtain the power spectrum

$$P(\omega) = \frac{6}{\pi c(0)}\int_0^{\infty} c(t)\cos\omega t\,dt, \tag{8.113}$$

which has many features similar to those of the phonon spectrum of the system: for example, a broad peak for the glassy state and sharp features for a crystalline state.

Thermodynamical quantities can also be evaluated from molecular dynamics simulations. For example, if a simulation is performed under the constant-pressure condition, we can obtain physical quantities such as the particle density, pair-distribution function, and so on, at different temperature. The inverse of the particle density is called the specific volume, denoted $V_P(T)$. The thermal expansion coefficient under the constant-pressure condition is then given by

$$\alpha_P = \frac{\partial V_P(T)}{\partial T}, \tag{8.114}$$

which is quite different when the system is in a liquid phase than it is in a solid phase. Furthermore, we can calculate the temperature-dependent enthalpy

$$H = E + P\Omega, \tag{8.115}$$

where E is the internal energy given by

$$E = \left\langle \sum_{i=1}^{N} \frac{m_i}{2} \mathbf{v}_i^2 + \sum_{i \neq j}^{N} V(r_{ij}) \right\rangle, \tag{8.116}$$

with P the pressure, and Ω the volume of the system. The specific heat under the constant-pressure condition is then obtained from

$$c_P = \frac{1}{N} \frac{\partial H}{\partial T}. \tag{8.117}$$

The specific heat under the constant-volume condition can be derived from the fluctuation of the internal energy $\langle (\delta E)^2 \rangle = \langle [E - \langle E \rangle]^2 \rangle$ with time, given as

$$c_V = \frac{\langle (\delta E)^2 \rangle}{k_B T^2}. \tag{8.118}$$

The isothermal compressibility κ_T is then obtained from the identity

$$\kappa_T = \frac{T \Omega \alpha_P^2}{c_P - c_V}, \tag{8.119}$$

which is also quite different for the liquid phase than for the solid phase. For more discussions on the molecular dynamics simulation of glass transition, see Yonezawa (1991).

Other aspects related to the structure and dynamics of a system can be studied through molecular dynamics simulations. The advantage of molecular dynamics over a typical stochastic simulation is that molecular dynamics can give all the information on the time dependence of the system, which is necessary for analyzing the structural and dynamical properties of the system. Molecular dynamics is therefore the method of choice in computer simulations of many-particle systems. However, stochastic simulations, such as Monte Carlo simulations, are sometimes easier to perform for some systems and are closely related to the simulations of quantum systems.

8.8 Ab initio molecular dynamics

In this section, we outline a very interesting simulation scheme that combines the calculation of the electronic structure and the molecular dynamics simulation for a system. This is known as *ab initio molecular dynamics*, which was devised and put into practice by Car and Parrinello (1985).

The maturity of molecular dynamics simulation schemes and the great advances in computing capacity have made it possible to perform molecular dynamics simulations for amorphous materials, biopolymers, and other complex systems. However, in order to obtain an accurate description of a specific system, we have to know the precise behavior of the interactions among the particles, that is, the ions in the system. Electrons move much faster than ions because the electron mass is much smaller than that of an ion. The position dependence of the interactions among the ions in a given system is therefore determined by the distribution of the electrons (electronic structure) at the specific moment. Thus, a good approximation of the electronic structure in a calculation can be obtained with all the nuclei fixed in space for that moment. This is the essence the Born–Oppenheimer approximation, which allows the degrees of freedom of the electrons to be treated separately from those of the ions.

In the past, the interactions among the ions were given in a parameterized form based on experimental data, quantum chemistry calculations, or the specific conditions of the system under study. All these procedures are limited due to the complexity of the electronic structure of the actual materials. We can easily obtain accurate parameterized interactions for the inert gases, such as Ar, but would have a lot of difficulties in obtaining an accurate parameterized interaction that can produce the various structures of ice correctly in the molecular dynamics simulation.

It seems that a combined scheme is highly desirable. We can calculate the many-body interactions among the ions in the system from the electronic structure calculated at every molecular dynamics time step and then determine the next configuration from such ab initio interactions. This can be achieved in principle, but in practice the scheme is restricted by the existing computing capacity. The combined method devised by Car and Parrinello (1985) was the first in its class and has been applied to the simulation of real materials.

Density functional theory

The density functional theory (Hohenberg and Kohn, 1964; Kohn and Sham, 1965) was introduced as a practical scheme to cope with the many-electron effect in atoms, molecules, and solids. The theorem proved by Hohenberg and Kohn (1964) states that the ground-state energy of an interacting system is the optimized value of an energy functional $E[\rho(\mathbf{r})]$ of the electron density $\rho(\mathbf{r})$ and that the corresponding density distribution of the optimization is the unique ground-state density distribution. Symbolically, we can write

$$E[\rho(\mathbf{r})] = E_{\text{ext}}[\rho(\mathbf{r})] + E_{\text{H}}[\rho(\mathbf{r})] + E_{\text{K}}[\rho(\mathbf{r})] + E_{\text{xc}}[\rho(\mathbf{r})], \tag{8.120}$$

where $E_{\text{ext}}[\rho(\mathbf{r})]$ is the contribution from the external potential $U_{\text{ext}}(\mathbf{r})$ with

$$E_{\text{ext}}[\rho(\mathbf{r})] = \int U_{\text{ext}}(\mathbf{r})\rho(\mathbf{r})\,d\mathbf{r}, \tag{8.121}$$

$E_H[\rho(\mathbf{r})]$ is the Hartree type of contribution due to the electron–electron interaction, given by

$$E_{\text{H}}[\rho(\mathbf{r})] = \frac{\frac{1}{2}e^2}{4\pi\epsilon_0}\int \frac{\rho(\mathbf{r}')\rho(\mathbf{r})}{|\mathbf{r}' - \mathbf{r}|}\,d\mathbf{r}\,d\mathbf{r}', \tag{8.122}$$

E_{K} is the contribution of the kinetic energy, and E_{xc} denotes the rest of the contributions and is termed the exchange–correlation energy functional.

In general, we can express the electron density in a spectral representation

$$\rho(\mathbf{r}) = \sum_i \psi_i^\dagger(\mathbf{r})\psi_i(\mathbf{r}), \tag{8.123}$$

where $\psi_i^\dagger(\mathbf{r})$ is the complex conjugate of the wavefunction $\psi_i(\mathbf{r})$ and the summation is over all the degrees of freedom, that is, all the occupied states with different spin orientations. Then the kinetic energy functional can be written as

$$E_{\text{K}}[\rho(\mathbf{r})] = -\frac{\hbar^2}{2m}\int \sum_i \psi_i^\dagger(\mathbf{r})\nabla^2\psi_i(\mathbf{r})\,d\mathbf{r}. \tag{8.124}$$

There is a constraint from the total number of electrons in the system, namely,

$$\int \rho(\mathbf{r})\,d\mathbf{r} = N, \tag{8.125}$$

which introduces the Lagrange multipliers into the variation. If we use the spectral representation of the density in the energy functional and apply the Euler equation with the Lagrange multipliers, we have

$$\frac{\delta E[\rho(\mathbf{r})]}{\delta\psi_i^\dagger(\mathbf{r})} - \varepsilon_i\psi_i(\mathbf{r}) = 0, \tag{8.126}$$

which leads to the Kohn–Sham equation

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\text{E}}(\mathbf{r})\right]\psi_i(\mathbf{r}) = \varepsilon_i\psi_i(\mathbf{r}), \tag{8.127}$$

where $V_{\text{E}}(\mathbf{r})$ is an effective potential given by

$$V_{\text{E}}(\mathbf{r}) = U_{\text{ext}}(\mathbf{r}) + V_{\text{H}}(\mathbf{r}) + V_{\text{xc}}(\mathbf{r}), \tag{8.128}$$

with

$$V_{\mathrm{xc}}(\mathbf{r}) = \frac{\delta E_{\mathrm{xc}}[\rho(\mathbf{r})]}{\delta\rho(\mathbf{r})}, \tag{8.129}$$

which cannot be obtained exactly. A common practice is to approximate it by its homogeneous density equivalent, the so-called local approximation, in which we assume that $V_{\mathrm{xc}}(\mathbf{r})$ is given by the same quantity of a uniform electron gas with density equal to $\rho(\mathbf{r})$. This is termed the *local density approximation*. The local density approximation has been successfully applied to many physical systems, including atomic, molecular, and condensed-matter systems. The unexpected success of the local density approximation in materials research has made it a standard technique for calculating electronic properties of new materials and systems. The procedure for calculating the electronic structure with the local density approximation can be described in several steps. We first construct the local approximation of $V_{\mathrm{xc}}(\mathbf{r})$ with a guessed density distribution. Then the Kohn–Sham equation is solved, and a new density distribution is constructed from the solution. With the new density distribution, we can improve $V_{\mathrm{xc}}(\mathbf{r})$ and then solve the Kohn–Sham equation again. This procedure is repeated until convergence is reached. Interested readers can find detailed discussions on the density functional theory in many monographs or review articles, for example, Kohn and Vashishta (1983), and Jones and Gunnarsson (1989).

The Car–Parrinello simulation scheme

The Hohenberg–Kohn energy functional forms the Born–Oppenheimer potential surface for the ions in the system. The idea of ab initio molecular dynamics is similar to the relaxation scheme we discussed in Chapter 7. We introduced a functional

$$U = \int \left\{ \frac{1}{2}\epsilon(x)\left[\frac{d\psi(x)}{dx}\right]^2 - \rho(x)\psi(x)\right\} dx \tag{8.130}$$

for the one-dimensional Poisson equation. Note that $\rho(x)$ here is the charge density instead of the particle density. The physical meaning of this functional is the electrostatic energy of the system. After applying the trapezoid rule to the integral and taking a partial derivative of U with respect to ϕ_i, we obtain the corresponding difference equation

$$(H_i + \rho_i)\psi_i = 0, \tag{8.131}$$

for the one-dimensional Poisson equation. Here $H_i\phi_i$ denotes $\epsilon_{i+1/2}\phi_{i+1} + \epsilon_{i-1/2}\phi_{i-1} - (\epsilon_{i+1/2} + \epsilon_{i-1/2})\phi_i$. If we combine the above equation with the relaxation scheme discussed in Section 7.5, we have

$$\psi_i^{(k+1)} = (1-p)\psi_i^{(k)} + p(H_i + \rho_i + 1)\psi_i^{(k)}, \tag{8.132}$$

which would optimize (minimize) the functional (the electrostatic energy) as $k \to \infty$. The indices k and $k+1$ are for iteration steps, and the index n is for the spatial points. The iteration can be interpreted as a fictitious time step, since we can rewrite the above equation as

$$\frac{\psi_i^{(k+1)} - \psi_i^{(k)}}{p} = (H_i + \rho_i)\psi_i^{(k)}, \tag{8.133}$$

with p acting like a fictitious time step. The solution converges to the true solution of the Poisson equation as k goes to infinity if the functional U decreases during the iterations.

The ab initio molecular dynamics is devised by introducing a fictitious time-dependent equation for the electron degrees of freedom:

$$\mu \frac{d^2\psi_i(\mathbf{r}, t)}{dt^2} = -\frac{1}{2}\frac{\delta E[\rho(\mathbf{r}, t); \mathbf{R}_n]}{\delta \psi_i^\dagger(\mathbf{r}, t)} + \sum_j \Lambda_{ij}\psi_j(\mathbf{r}), \tag{8.134}$$

where μ is an adjustable parameter introduced for convenience, Λ_{ij} is the Lagrange multiplier, introduced to ensure the orthonormal condition of the wave-functions $\psi_i(\mathbf{r}, t)$, and the summation is over all the occupied states. Note that the potential energy surface E is a functional of the electron density as well as a function of the ionic coordinates $\mathbf{R}_n$ for $n = 1, 2, \ldots, N_c$, with a total of N_c ions in the system. In practice, we can also consider the first-order time derivative equation, with $d^2\psi_i(\mathbf{r}, t)/dt^2$ replaced by the first-order derivative $d\psi_i(\mathbf{r}, t)/dt$, because either the first-order or the second-order derivative will approach zero at the limit of convergence. Second-order derivatives were used in the original work of Car and Parrinello and were later shown to yield a fast convergence if a special damping term is introduced (Tassone, Mauri, and Car, 1994). The ionic degrees of freedom are then simulated from Newton's equation

$$M_n \frac{d^2\mathbf{R}_n}{dt^2} = -\frac{\partial E[\rho(\mathbf{r}, t); \mathbf{R}_n]}{\partial \mathbf{R}_n}, \tag{8.135}$$

where M_n and $\mathbf{R}_n$ are the mass and the position vector of the nth particle. The advantage of ab initio molecular dynamics is that the electron degrees of freedom and the ionic degrees of freedom are simulated simultaneously by the above equations. Since its introduction by Car and Parrinello (1985), the method has been applied to many systems, especially those without a crystalline structure, namely, liquids and amorphous materials. We will not go into more detail on the method or its applications; interested readers can find them in the review by Tassone, Mauri, and Car (1994). Progress in ab initio molecular dynamics has also included mapping the Hamiltonian onto a tight-binding model in which the evaluation of the electron degrees of freedom is drastically simplified (Wang, Chan, and Ho, 1989). This approach has also been applied to many systems, for example, amorphous carbon and carbon clusters. More discussions on the method can be found in several review articles, for example, Oguchi and Sasaki (1991).

Exercises

8.1 Show that the Verlet algorithm preserves the time reversal of Newton's equation.

8.2 Derive the velocity version of the Verlet algorithm and show that the position update has the same accuracy as in the original Verlet algorithm.

8.3 Write a program that uses the velocity version of the Verlet algorithm to simulate the small clusters of ions $(\mathrm{Na}^+)_n(\mathrm{Cl}^-)_m$ for small n and m. Use the empirical interaction potential given in Eq. (5.64) for the ions and set up the initial configuration as the equilibrium configuration. Assign initial velocities from the Maxwell distribution. Discuss the time dependence of the kinetic energy with different total energies.

8.4 Explore the coexistence of the liquid and solid phases in a Lennard–Jones cluster that has an intermediate size of about 150 particles through the microcanonical molecular dynamics simulation. Analyze the melting process in the cluster and the size dependence of the coexistence region.

8.5 A two-dimensional system can behave quite differently from its three-dimensional counterpart. Use the molecular dynamics technique to study a two-dimensional Lennard–Jones cluster of the intermediate size of about 100 particles. Do the liquid and solid phases coexist in any temperature region? Discuss the difference found between the three-dimensional and two-dimensional clusters.

8.6 Develop a program with the fifth-order Gear predictor–corrector scheme and apply it to the damped pendulum under a sinusoidal driving force. Study the properties of the pendulum with different values of the parameters.

8.7 Apply the fifth-order Gear scheme to study the long-time behavior of a classical helium atom in two dimensions. Explore different initial conditions. Is the system unstable or chaotic under certain initial conditions?

8.8 Derive the Hoover equations from the Nosé Lagrangian. Show that the generalized Lagrangian given in Section 8.6 can provide the correct equations of motion to ensure the constraint of constant temperature as well as that of constant pressure.

8.9 Develop a molecular dynamics program to study the structure of a cluster of identical charges inside a three-dimensional isotropic, harmonic trap. Under what condition does the cluster form a crystal? What happens if the system is confined in a plane?

8.10 Show that the Parrinello–Rahman Lagrangian allows the shape of the simulation box to change, and derive equations of motion from it. Find the Lagrangian that combines the Parrinello–Rahman Lagrangian and Nosé Lagrangian and derive the equations of motion from this generalized Lagrangian.

8.11 For quantum many-body systems, the n-body density function is defined by

$$\rho_n(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n) = \frac{1}{\mathcal{Z}} \frac{N!}{(N-n)!} \int |\Psi(\mathbf{R})|^2 \, d\mathbf{r}_{n+1} \, d\mathbf{r}_{n+2} \cdots d\mathbf{r}_N,$$

where $\Psi(\mathbf{R})$ is the ground-state wavefunction of the many-body system and $\mathcal{Z}$ is the normalization constant given by

$$\mathcal{Z} = \int |\Psi(\mathbf{R})|^2 \, d\mathbf{r}_1 \, d\mathbf{r}_2 \cdots d\mathbf{r}_N.$$

The pair-distribution function is related to the two-body density function through

$$\rho(\mathbf{r}) g(\mathbf{r}, \mathbf{r}') \rho(\mathbf{r}') = \rho_2(\mathbf{r}, \mathbf{r}').$$

Show that

$$\int \rho(\mathbf{r}) [\tilde{g}(\mathbf{r}, \mathbf{r}') - 1] \, d\mathbf{r} = -1,$$

where

$$\tilde{g}(\mathbf{r}, \mathbf{r}') = \int_0^1 g(\mathbf{r}, \mathbf{r}'; \lambda) \, d\lambda,$$

with $g(\mathbf{r}, \mathbf{r}'; \lambda)$ as the pair-distribution function under the scaled interaction

$$V(\mathbf{r}, \mathbf{r}'; \lambda) = \lambda V(d\mathbf{r}, \mathbf{r}').$$

8.12 Show that the exchange–correlation energy functional is given by

$$E_{\text{xc}}[\rho(\mathbf{r})] = \frac{1}{2} \int \rho(\mathbf{r}) V(\mathbf{r}, \mathbf{r}') [\tilde{g}(\mathbf{r}, \mathbf{r}') - 1] \rho(\mathbf{r}') \, d\mathbf{r} \, d\mathbf{r}',$$

with $\tilde{g}(\mathbf{r}, d\mathbf{r}')$ given from the last problem.

Chapter 9
Modeling continuous systems

It is usually more difficult to simulate continuous systems than discrete ones, especially when the properties under study are governed by nonlinear equations. The systems can be so complex that the length scale at the atomic level can be as important as the length scale at the macroscopic level. The basic idea in dealing with complicated systems is similar to a divide-and-conquer concept, that is, dividing the systems with an understanding of the length scales involved and then solving the problem with an appropriate method at each length scale. A specific length scale is usually associated with an energy scale, such as the average temperature of the system or the average interaction of each pair of particles. The divide-and-conquer schemes are quite powerful in a wide range of applications. However, each method has its advantages and disadvantages, depending on the particular system.

9.1 Hydrodynamic equations

In this chapter, we will discuss several methods used in simulating continuous systems. First we will discuss a quite mature method, the *finite element method*, which sets up the idea of partitioning the system according to physical condition. Then we will discuss another method, the *particle-in-cell method*, which adopts a mean-field concept in dealing with a large system involving many, many atoms, for example, 10^{23} atoms. This method has been very successful in the simulations of plasma, galactic, hydrodynamic, and magnetohydrodynamic systems. Then we will briefly highlight a relatively new method, the *lattice Boltzmann method*, which is closely related to the lattice-gas method of *cellular automata* and has been applied in modeling several continuous systems.

Before going into the detail of each method, we introduce the hydrodynamic equations. The equations are obtained by analyzing the dynamics of a small element of fluid in the system and then taking the limit of continuity. We can obtain three basic equations by examining the changes in the mass, momentum, and energy of a small element in the system. For simplicity, let us first assume that the fluid is neutral and not under an external field. The three fundamental hydrodynamic equations are

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0, \tag{9.1}$$

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot \mathbf{\Pi} = 0, \tag{9.2}$$

$$\frac{\partial}{\partial t}\left(\rho\varepsilon + \frac{1}{2}\rho v^2\right) + \nabla \cdot \mathbf{j}_{\mathrm{e}} = 0. \tag{9.3}$$

The first equation is commonly known as the *continuity equation*, which is the result of mass conservation. The second equation is known as the Navier–Stokes equation, which comes from Newton's equation applied to an infinitesimal element in the fluid. The final equation is the result of the *work–energy theorem*. In these equations, ρ is the mass density, $\mathbf{v}$ is the velocity, ε is the internal energy per unit mass, $\mathbf{\Pi}$ is the momentum flux tensor, and $\mathbf{j}_{\mathrm{e}}$ is the energy flux density. The momentum flux tensor can be written as

$$\mathbf{\Pi} = \rho \mathbf{v}\mathbf{v} - \mathbf{\Gamma}, \tag{9.4}$$

where $\mathbf{\Gamma}$ is the stress tensor of the fluid, given as

$$\mathbf{\Gamma} = \eta\left[\nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{T}}\right] + \left[\left(\zeta - \frac{2\eta}{3}\right)\nabla\cdot\mathbf{v} - P\right]\mathbf{I}, \tag{9.5}$$

where η and ζ are coefficients of bulk and shear viscosity, P is the pressure in the fluid, and $\mathbf{I}$ is the unit tensor. The energy flux density is given by

$$\mathbf{j}_{\mathrm{e}} = \mathbf{v}\left(\rho\varepsilon + \frac{1}{2}\rho v^2\right) - \mathbf{v}\cdot\mathbf{\Gamma} - \kappa\nabla T, \tag{9.6}$$

where the last term is the thermal energy flow due to the gradient of the temperature T, with κ being the thermal conductivity. The above set of equations can be solved together with the equation of state

$$f(\rho, P, T) = 0, \tag{9.7}$$

which provides a particular relationship between several thermal variables for the given system. Now we can add the effects of external fields into the related equations. For example, if gravity is important, we can modify Eq. (9.2) to

$$\frac{\partial}{\partial t}(\rho\mathbf{v}) + \nabla\cdot\mathbf{\Pi} = \rho\mathbf{g}, \tag{9.8}$$

where $\mathbf{g}$ is the gravitational field given by

$$\mathbf{g} = -\nabla\Phi. \tag{9.9}$$

Here Φ is the gravitational potential from the Poisson equation

$$\nabla^2\Phi = 4\pi G\rho, \tag{9.10}$$

with G being the gravitational constant.

When the system is charged, electromagnetic fields become important. We then need to modify the momentum flux tensor, energy density, and energy flux density. A term $(\mathbf{BB} - \mathbf{I}B^2/2)/\mu$ should be added to the momentum flux tensor. A magnetic field energy density term $B^2/(2\mu)$ should be added to the energy density, and an extra energy flux term $\mathbf{E} \times \mathbf{B}/\mu$ should be added to the energy flux density. Here μ is the magnetic permeability with a limit of $\mu = \mu_0$ in free space. The electric current density $\mathbf{j}$, electric field $\mathbf{E}$, and magnetic field $\mathbf{B}$ are related through the Maxwell equations and the charge transport equation. We will come back to this point in the later sections of this chapter.

These hydrodynamic or magnetohydrodynamic equations can be solved in principle with the finite difference methods discussed in Chapter 7. However, in this chapter, we introduce some other methods, mainly based on the divide-and-conquer schemes. For more detailed discussions on the hydrodynamic and magnetohydrodynamic equations, see Landau and Lifshitz (1987) and Lifshitz and Pitaevskii (1981).

9.2 The basic finite element method

In order to show how a typical finite element method works for a specific problem, let us take the one-dimensional Poisson equation as an example. Assume that the charge distribution is $\rho(x)$ and the equation for the electrostatic potential is

$$\frac{d^2\phi(x)}{dx^2} = -\frac{\rho(x)}{\epsilon_0}. \tag{9.11}$$

In order to simplify our discussion here, let us assume that the boundary condition is given as $\phi(0) = \phi(1) = 0$. As discussed in Chapter 4, we can always express the solution in terms of a complete set of orthogonal basis functions as

$$\phi(x) = \sum_i a_i u_i(x), \tag{9.12}$$

where the basis functions $u_i(x)$ satisfy

$$\int_0^1 u_j(x) u_i(x)\, dx = \delta_{ij} \tag{9.13}$$

and $u_i(0) = u_i(1) = 0$. We have assumed that $u_i(x)$ is a real function and that the summation is over all the basis functions. One of the choices for $u_i(x)$ is to have $u_i(x) = \sqrt{2} \sin i\pi x$. In order to obtain the actual value of $\phi(x)$, we need to solve all the coefficients a_i by applying the differential equation and the orthogonal condition in Eq. (9.13). This becomes quite a difficult task if the system has a higher dimensionality and an irregular boundary.

The finite element method is designed to find a good approximation of the solution for irregular boundary conditions or in situations of rapid change of the solution. Typically, the approximation is still written as a series

$$\phi_n(x) = \sum_{i=1}^{n} a_i u_i(x), \tag{9.14}$$

with a finite number of terms. Here $u_i(x)$ is a set of linearly independent local functions, each defined in a small region around a *node* x_i. If we use this approximation in the differential equation, we have a nonzero value

$$r_n(x) = \phi_n''(x) + \frac{\rho(x)}{\epsilon_0}, \tag{9.15}$$

which would be zero if $\phi_n(x)$ were the exact solution. The goal now is to set up a scheme that would make $r_n(x)$ small over the whole region during the process of determining all a_i. The selection of $u_i(x)$ and the procedure for optimizing $r_n(x)$ determine how accurate the solution is for a given n. We can always improve the accuracy with a higher n. One scheme for performing optimization is to introduce a weighted integral

$$g_i = \int_0^1 r_n(x) w_i(x)\, dx, \tag{9.16}$$

which is then forced to be zero during the determination of a_i. Here $w_i(x)$ is a selected weight.

The procedure just described is the essence of the finite element method. The region of interest is divided into many small pieces, usually of the same topology but not the same size, for example, different triangles for a two-dimensional domain. Then a set of linearly independent local functions $u_i(x)$ is selected, with each defined around a node and its neighborhood. We then select the weight $w_i(x)$, which is usually chosen as a local function as well. For the one-dimensional Poisson equation, the weighted integral becomes

$$g_i = \int_0^1 \left[\sum_{j=1}^{n} a_j u_j''(x) + \frac{\rho(x)}{\epsilon_0} \right] w_i(x)\, dx = 0, \tag{9.17}$$

which is equivalent to a linear equation set

$$\mathbf{A}\mathbf{a} = \mathbf{b}, \tag{9.18}$$

with

$$A_{ij} = -\int_0^1 u_i''(x) w_j(x)\, dx \tag{9.19}$$

and

$$b_i = \frac{1}{\epsilon_0} \int_0^1 \rho(x) w_i(x)\, dx. \tag{9.20}$$

The advantage of choosing $u_i(x)$ and $w_i(x)$ as local functions lies in the solution of the linear equation set given in Eq. (9.18). Basically, we need to solve only a tridiagonal or a band matrix problem, which can be done quite quickly. The idea is to make the problem computationally simple but numerically accurate. That means the choice of the weight $w_i(x)$ is rather an art. A common choice is that $w_i(x) = u_i(x)$, which is called the *Galerkin method*.

Now let us consider in detail the application of the Galerkin method to the one-dimensional Poisson equation. We can divide the region [0, 1] into $n + 1$ equal intervals with $x_0 = 0$ and $x_{n+1} = 1$ and take

$$u_i(x) = \begin{cases} (x - x_{i-1})/h & \text{for } x \in [x_{i-1}, x_i], \\ (x_{i+1} - x)/h & \text{for } x \in [x_i, x_{i+1}], \\ 0 & \text{otherwise.} \end{cases} \tag{9.21}$$

Here $h = x_i - x_{i-1} = 1/(n + 1)$ is the spatial interval. Note that this choice of $u_i(x)$ satisfies the boundary condition. Let us take a very simple charge distribution

$$\rho(x) = \epsilon_0 \pi^2 \sin \pi x \tag{9.22}$$

for illustrative purposes. We can easily determine the weighted integrals to obtain the matrix elements

$$A_{ij} = \int_0^1 u_i'(x) u_j'(x)\, dx = \begin{cases} 2/h & \text{for } i = j, \\ -1/h & \text{for } i = j \pm 1, \\ 0 & \text{otherwise} \end{cases} \tag{9.23}$$

and the vector

$$\begin{aligned} b_i &= \frac{1}{\epsilon_0} \int_0^1 \rho(x) u_i(x)\, dx \\ &= \frac{\pi}{h}(x_{i-1} + x_{i+1} - 2x_i) \cos \pi x_i \\ &\quad + \frac{1}{h}(2 \sin \pi x_i - \sin \pi x_{i-1} - \sin \pi x_{i+1}). \end{aligned} \tag{9.24}$$

Now we are ready to put all these values into a program. Note that the coefficient matrix **A** is automatically symmetric and tridiagonal because the local basis $u_i(x)$ is confined in the region $[x_{i-1}, x_{i+1}]$. As we discussed in Chapters 2, 5, and 7, an LU decomposition scheme can easily be devised to solve this tridiagonal matrix problem. Here is the implementation of the algorithm.

```
// Program for the one-dimensional Poisson equation.

import java.lang.*;
public class Poisson {
  final static int n = 99, m = 2;
  public static void main(String argv[]) {
  double d[] = new double[n];
  double b[] = new double[n];
  double c[] = new double[n];
  double h = 1.0/(n+1), pi = Math.PI;

// Evaluate the coefficient matrix elements
```

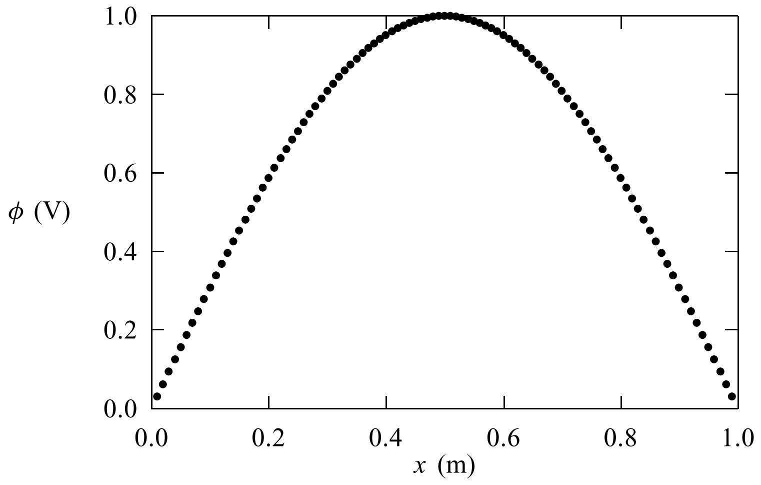


Fig. 9.1 Numerical solution of the one-dimensional Poisson equation obtained with the program given in the text.

```
    for (int i=0; i<n; ++i) {
      d[i] =  2;
      c[i] = -1;
      double xm = h*i;
      double xi = xm + h;
      double xp = xi + h;
      b[i] = 2*Math.sin(pi*xi) - Math.sin(pi*xm)
             -Math.sin(pi*xp);
    }

// Obtain the solution
    double u[] = tridiagonalLinearEq(d, c, c, b);

// Output the result
    double x = h;
    double mh = m*h;
    for (int i=0; i<n; i+=m) {
      System.out.println(x + " " + u[i]);
      x += mh;
    }
  }

// Method to solve the tridiagonal linear equation set.

  public static double[] tridiagonalLinearEq(double d[],
    double e[], double c[], double b[]) {...}
}
```

The result from the above program is plotted in Fig. 9.1. As we can easily show, the above problem has an analytic solution, $\phi(x) = \sin \pi x$. There is one more related issue involved in the Galerkin method, that is, dealing with different boundary conditions. We have selected a very simple boundary condition in the above example. In general, we may have nonzero boundary conditions, for example,

$\phi(0) = \phi_0$ and $\phi(1) = \phi_1$. We can write the solution as

$$\phi_n(x) = (1 - x)\phi_0 + x\phi_1 + \sum_{i=1}^{n} a_i u_i(x), \tag{9.25}$$

in which the first two terms satisfy the boundary conditions; the summation part is zero at the boundaries. Similarly, if the boundary conditions are of the Neumann type or a mixed type, special care can be taken by adding a couple of terms and the local basis $u_i(x)$ can be made to satisfy homogeneous boundary conditions only. We will discuss a few of other situations in the next section.

9.3 The Ritz variational method

The basic finite element method discussed above resembles the idea of the discretization scheme of the finite difference method discussed in Chapter 7. In the relaxation scheme further developed there a functional was constructed that would lead to the original differential equation when optimized. For example, the functional for the one-dimensional Helmholtz equation

$$\frac{d^2\phi(x)}{dx^2} + k^2\phi(x) = -s(x), \tag{9.26}$$

where k^2 is a parameter (or eigenvalue) and $s(x)$ is a function of x, defined in the region [0, 1], is given by

$$E[\phi(x)] = \int_0^1 \left\{ \frac{1}{2} \left[\phi'^2(x) - k^2\phi^2(x)\right] - s(x)\phi(x) \right\} dx, \tag{9.27}$$

which leads to the original differential equation from the Euler–Lagrange equation

$$\frac{\delta E[\phi(x)]}{\delta\phi(x)} = 0, \tag{9.28}$$

with the condition that the variations at the boundaries are zero. What we did in Chapter 7 was to discretize the integrand of the functional and then consider the solution at each lattice point as an independent function when the optimization was performed. A difference equation was obtained from the Euler–Lagrange equation and acted as a finite difference approximation of the original differential equation.

The variational principle used is called the *Ritz variational principle*, which ensures that the true solution of the differential equation has the lowest value of $E[\phi(x)]$. An approximate solution is better the lower its value of $E[\phi(x)]$. Thus, there are two basic steps in the Ritz variational scheme. A proper functional for a specific differential equation has to be constructed first. Then we can approximate the solution of the equation piece by piece in each finite element and apply the Ritz variational principle to obtain the linear equation set for the coefficients of the approximate solution.

Let us here use the one-dimensional Helmholtz equation to demonstrate some details of the Ritz variational method. The approximate solution is still expressed as a linear combination of a set of local functions

$$\phi_n(x) \simeq \sum_{i=1}^{n} a_i u_i(x), \tag{9.29}$$

which are linearly independent. Here $u_i(x)$ is usually defined in the neighborhood of the ith node. The essence of the Ritz variational method is to treat each coefficient a_i as an independent variational parameter. The optimization is then done with respect to each a_i with

$$\frac{\partial E[(\phi_n(x)]}{\partial a_i} = 0, \tag{9.30}$$

for $i = 1, 2, \ldots, n$, which produces a linear equation set

$$\mathbf{Aa} = \mathbf{b}, \tag{9.31}$$

with

$$A_{ij} = \frac{\partial^2 E[\phi_n(x)]}{\partial a_i \partial a_j} = \int_0^1 [u_i'(x)u_j'(x) - k^2 u_i(x)u_j(x)]\, dx \tag{9.32}$$

and

$$b_i = \int_0^1 s(x)u_i(x)\, dx. \tag{9.33}$$

The Ritz variational scheme actually reaches the same linear equation set as the Galerkin method for this specific problem, because

$$\int_0^1 u_i''(x)u_j(x)\, dx = -\int_0^1 u_i'(x)u_j'(x)\, dx, \tag{9.34}$$

if $u_i(x)$ or $u_i'(x)$ is zero at the boundaries. In fact, if the first two terms in $E[\phi(x)]$ together are positive definite, the Ritz method is equivalent to the Galerkin method for problems with homogeneous boundary conditions in general.

In most cases, it is much easier to deal with the so-called weak form of the original differential equation. Here we demonstrate how to obtain the weak form of a differential equation. If we still take the Helmholtz equation as an example, we can multiply it by a function $\psi(x)$, then we have

$$\int_0^1 [\phi''(x) + k^2\phi(x) + s(x)]\psi(x)\, dx = 0 \tag{9.35}$$

after integration. Here $\psi(x)$ is assumed to be squarely integrable, with $\psi(0) = \psi(1) = 0$. The reverse is also true from the fundamental variational theorem, which states that if we have

$$\int_a^b r(x)\psi(x)\, dx = 0 \tag{9.36}$$

for any given function $\psi(x)$, with $\psi(a) = \psi(b) = 0$, then the equation

$$r(x) = 0 \tag{9.37}$$

will result. The only restriction on $\psi(x)$ is that it has to be squarely integrable. Now if we integrate the first term of Eq. (9.35) by parts, we have

$$\int_0^1 \{\phi'(x)\psi'(x) - [k^2\phi(x) + s(x)]\psi(x)\}dx = 0, \tag{9.38}$$

which is the so-called *weak form* of the original differential equation, because now only the first-order derivatives are involved in the above integral. We need to realize that the solution of the weak form is not necessarily the same as the original differential equation, especially when the second-order derivative of $\phi(x)$ is not well behaved. The weak form can be solved approximately by taking the linear combination

$$\phi_n(x) = \sum_{i=0}^{n} a_i u_i(x), \tag{9.39}$$

with $\psi(x) = u_j(x)$. Then the weak form of the equation becomes a linear equation set

$$\sum_{i=1}^{n} a_i \int_0^1 [u_i'(x)u_j'(x) - k^2 u_i(x)u_j(x) - s(x)u_i(x)]dx = 0, \tag{9.40}$$

for $j = 1, 2, \ldots, n$. This is exactly the same as the equation derived from the Ritz variational scheme. In most cases, the first and second parts of the coefficient matrix are written separately as $\mathbf{A} = \mathbf{K} + \mathbf{M}$, where

$$K_{ij} = \int_0^1 u_i'(x)u_j'(x)\,dx \tag{9.41}$$

is referred to as the *stiffness matrix* and

$$M_{ij} = -k^2 \int_0^1 u_i(x)u_j(x)\,dx \tag{9.42}$$

is referred to as the *mass matrix*. For the case of $s(x) = 0$, we have an eigenvalue problem with $\mathbf{A}$ still being an $n \times n$ matrix and $\mathbf{b} = \mathbf{0}$. The actual forms of the matrix elements are given by the choice of $u_i(x)$. A good example is the choice of $u_i(x)$ discussed in the preceding section. The eigenvalues can be obtained from

$$|\mathbf{A}| = 0. \tag{9.43}$$

When we have the exact forms of $\mathbf{K}$ and $\mathbf{M}$, we can use the schemes developed in Chapter 5 to find all the quantities associated with them. Note that if we choose $u_i(x)$ to make $\mathbf{K}$ and $\mathbf{M}$ tridiagonal, the numerical complexity is reduced drastically.

As we have mentioned, the boundary condition always needs special attention in the choice of $u_i(x)$ and the construction of $\phi_n(x)$. Here we illustrate the modification under different boundary conditions from the solution of the Sturm–Liouville equation

$$[p(x)\phi'(x)]' - r(x)\phi(x) + \lambda w(x)\phi(x) = s(x), \tag{9.44}$$

where λ is the eigenvalue and $p(x)$, $r(x)$, and $w(x)$ are functions of x. The Sturm–Liouville equation is a general form of some very important equations in physics and engineering, for example, the Legendre equation and the spherical Bessel equation. We can rewrite the Sturm–Liouville equation as

$$-[p(x)\phi'(x)]' + q(x)\phi(x) + s(x) = 0, \tag{9.45}$$

for the convenience of the variational procedure and assume that $p(x)$, $q(x)$, and $s(x)$ are well-behaved functions, that is, continuous and squarely integrable in most cases. We can easily show that, for homogeneous boundary conditions, the functional

$$E[\phi(x)] = \frac{1}{2}\int_0^1 [p(x)\phi'^2(x) + q(x)\phi^2(x) + 2s(x)\phi(x)]dx \tag{9.46}$$

will produce the given differential equation under optimization. Now if the boundary condition is the so-called natural boundary condition

$$\phi'(0) + \alpha\phi(0) = 0, \tag{9.47}$$

$$\phi'(1) + \beta\phi(1) = 0, \tag{9.48}$$

where α and β are some given parameters, we can show that the functional

$$E[\phi(x)] = \frac{1}{2}\int_0^1 [p(x)\phi'^2(x) + q(x)\phi^2(x) + 2s(x)\phi(x)]dx - \frac{\alpha}{2}p(0)\phi^2(0) + \frac{\beta}{2}p(1)\phi^2(1) \tag{9.49}$$

will produce the Sturm–Liouville equation with the correct boundary condition. We can convince ourselves by taking the functional variation of $E[\phi(x)]$ with the given boundary condition; the Sturm–Liouville equation should result. When we set up the finite elements, we can include the boundary points in the expression of the approximate solution, and the modified functional will take care of the necessary adjustment. Of course, the local functions $u_0(x)$ and $u_{n+1}(x)$ are set to zero outside of the region [0, 1].

However, when the boundary values are not zero, special care is needed in the construction of $u_i(x)$ and $\phi_n(x)$. For example, for the inhomogeneous Dirichlet boundary condition, we can introduce two more terms that explicitly take care of the boundary condition. Another way of doing this is to include the boundary points in the approximation

$$\phi_n(x) = \sum_{i=0}^{n+1} a_i u_i(x). \tag{9.50}$$

If we substitute the above approximate solution into the inhomogeneous Dirichlet boundary condition, we have

$$a_0 = \frac{\phi(0)}{u_0(0)}, \tag{9.51}$$

$$a_{n+1} = \frac{\phi(1)}{u_{n+1}(1)}. \tag{9.52}$$

We have assumed that $u_i(0) = u_i(1) = 0$ for $i \neq 0, n+1$. The coefficients a_0 and a_{n+1} are obtained immediately if the form of $u_i(x)$ is given. The approximation $\phi_n(x)$ can be substituted into the functional before taking the Ritz variation. The resulting linear equation set is still in an $n \times n$ matrix form. We will come back to this point again in the next section when we discuss higher-dimensional systems.

9.4 Higher-dimensional systems

The Galerkin or Ritz scheme can be generalized for higher-dimensional systems. For example, if we want to study a two-dimensional system, the nodes can be assigned at the vertices of the polygons that cover the specified domain. For convenience in solving the linear equation set, we need to construct all the elements with a similar shape, for example, all triangular. The finite element coefficient matrix is then a band matrix, which is much easier to solve.

The simplest way to construct an approximate solution is to choose $u_i(x, y)$ as a function around the ith node, for example, a pyramidal function with $u_i(x_i, y_i) = 1$ that linearly goes to zero at the nearest neighboring nodes. Then the approximate solution of a differential equation is represented by a linear combination of the local functions $u_i(x, y)$ as

$$\phi_n(x, y) = \sum_{i=1}^{n} a_i u_i(x, y), \tag{9.53}$$

where the index i runs through all the internal nodes if the boundary satisfies the homogeneous Dirichlet condition. For illustrative purposes, let us still take the Helmholtz equation

$$\nabla^2\phi(x, y) + k^2\phi(x, y) = -s(x, y) \tag{9.54}$$

as an example. The corresponding functional with the approximate solution is given by

$$E[\phi_n] = \frac{1}{2}\int \left\{[\nabla\phi_n(x, y)]^2 - k^2\phi_n^2(x, y) - 2s(x, y)\phi_n(x, y)\right\} dx\, dy, \tag{9.55}$$

which is optimized with the Ritz variational procedure:

$$\frac{\partial E[\phi_n]}{\partial a_i} = 0. \tag{9.56}$$

This variation produces a linear equation set

$$\mathbf{A}\mathbf{a} = \mathbf{b} \tag{9.57}$$

with

$$A_{ij} = \int [\nabla u_i(x, y) \cdot \nabla u_j(x, y) - k^2 u_i(x, y) u_j(x, y)] dx\, dy \tag{9.58}$$

and

$$b_i = \int s(x, y) u_i(x, y)\, dx\, dy. \tag{9.59}$$

Note that the node index runs through all the vertices of the finite elements in the system except the ones on the boundary.

In reality, the homogeneous Dirichlet boundary condition does not always hold. A more general situation is given by the boundary condition

$$\nabla_n \phi(x, y) + \alpha(x, y)\phi(x, y) + \beta(x, y) = 0, \tag{9.60}$$

which includes most cases in practice. Here $\alpha(x, y)$ and $\beta(x, y)$ are assumed to be given functions, and $\nabla_n \phi(x, y)$ is the gradient projected outward perpendicular to the boundary line. It is straightforward to show that the functional

$$\begin{aligned} E[\phi] = \frac{1}{2} \int & \left\{ [\nabla \phi(x, y)]^2 - k^2 \phi^2(x, y) - 2s(x, y)\phi(x, y) \right\} dx\, dy \\ & + \int_B [\alpha(x, y)\phi^2(x, y) + \beta(x, y)\phi(x, y)] d\ell \end{aligned} \tag{9.61}$$

can produce the differential equation with the given general boundary condition if the Euler–Lagrange equation is applied. Here the line integral is performed along the boundary B with the domain on the left-hand side. As we discussed for one-dimensional equations, the boundary points can be included in the expansion of the approximate solution for the inhomogeneous boundary conditions. The local functions $u_i(x, y)$ at the boundaries are the same as other local functions except that they are set to zero outside the domain of the problem specified. When the local functions for the internal nodes are selected so as to be zero at the boundary, the coefficients for the terms from the nodes at the boundary points are given by the values of the local function at those points.

A very efficient way of constructing the stiffness matrix and the mass matrix is to view the system as a collection of elements and the solution of the equation is given in each individual element separately. For example, we can divide the domain into many triangles and label all the triangles sequentially. The solution of the equation is then given in each triangle separately. For simplicity, we will use the pyramidal functions $u_i(x, y)$ as the local functions for the expansion of the approximate solution; $u_i(x, y)$ is 1 at the ith node and goes to zero linearly at the nearest neighboring nodes. Let us select a triangle element labeled by σ. Assume that the three nodes at the three corners of the selected element are labeled i, j, and k. The approximate solution in the element can be expressed as

$$\phi_{n\sigma}(x, y) = a_i u_i(x, y) + a_j u_j(x, y) + a_k u_k(x, y) \tag{9.62}$$

because the local basis $u_l(x)$ is taken to be a function that is zero beyond the nearest neighbors of the lth node. As we can show, for the case of a local pyramidal function, the approximation is given by a linear function

$$\phi_{n\sigma}(x, y) = \alpha + \beta x + \gamma y, \tag{9.63}$$

with the constant α given by a determinant

$$\alpha = \frac{1}{\Delta} \begin{vmatrix} a_i & a_j & a_k \\ x_i & x_j & x_k \\ y_i & y_j & y_k \end{vmatrix} \tag{9.64}$$

and the coefficients β and γ given by two other determinants

$$\beta = \frac{1}{\Delta} \begin{vmatrix} 1 & 1 & 1 \\ a_i & a_j & a_k \\ y_i & y_j & y_k \end{vmatrix} \tag{9.65}$$

and

$$\gamma = \frac{1}{\Delta} \begin{vmatrix} 1 & 1 & 1 \\ x_i & x_j & x_k \\ a_i & a_j & a_k \end{vmatrix}. \tag{9.66}$$

The constant Δ can also be expressed in a determinant form with

$$\Delta = \begin{vmatrix} 1 & 1 & 1 \\ x_i & x_j & x_k \\ y_i & y_j & y_k \end{vmatrix}. \tag{9.67}$$

The functional can now be expressed as the summation of integrals performed on each triangular element. The elements of the coefficient matrix are then obtained from

$$A_{ij} = \sum_{\sigma} A_{ij}^{\sigma}, \tag{9.68}$$

where the summation over σ is for the integrals over two adjacent elements that share the nodes i and j if $i \neq j$, and six elements that share the ith node if $i = j$. For the Helmholtz equation, we have

$$A_{ij}^{\sigma} = \int_{\sigma} [\nabla u_i(x) \cdot \nabla u_j(x) - k^2 u_i(x) u_j(x)] dx\, dy, \tag{9.69}$$

which is performed over the element σ. Similarly, the constant vector element can also be obtained from

$$b_i = \sum_{\sigma} b_i^{\sigma}, \tag{9.70}$$

where the summation is over all the six elements that share the ith node. As we have discussed in earlier sections in this chapter, there are other choices of $u_i(x, y)$ than just a linear pyramidal function. Interested readers can find these choices in Strang and Fix (1973). A Fortran program that solves the extended Helmholtz equation in two dimensions (a two-dimensional version of the Sturm–Liouville equation) is given in Vichnevetsky (1981, pp. 269–78). Extensive discussions on the method and its applications can be found in Burnett (1987) and Cook *et al.* (2001).

An extremely important issue in the finite element method is the analysis of errors, which would require more space than is available here. Similar procedures can also be developed for three-dimensional systems. Because most applications of the finite element method are in one-dimensional or two-dimensional problems, we will not go any further here. Interested readers can find examples in the literature.

9.5 The finite element method for nonlinear equations

The examples discussed in this chapter so far are all confined to linear equations. However, the finite element method can also be applied to nonlinear equations. In this section, we use the two-dimensional Navier–Stokes equation as an illustrative example to demonstrate the method for such cases.

For simplicity, we will assume that the system is stationary, that is, it has no time dependence, and under constant temperature and density, that is, a stationary isothermal and incompressible fluid. The Navier–Stokes equation under such conditions is given by

$$\rho\mathbf{v}\cdot\boldsymbol{\nabla}\mathbf{v}+\boldsymbol{\nabla}P-\eta\nabla^2\mathbf{v}=\rho\mathbf{g}, \tag{9.71}$$

where ρ is the density, $\mathbf{v}(x, y)$ is the velocity, η is the viscosity coefficient, $P(x, y)$ is the pressure, and $\mathbf{g}(x, y)$ is the external force field. The continuity equation under the incompressible condition is simply given by

$$\boldsymbol{\nabla}\cdot\mathbf{v}=0, \tag{9.72}$$

which is the continuity equation under the given conditions. Because the system is two-dimensional, we have three coupled equations for $v_x(x, y)$, $v_y(x, y)$, and $P(x, y)$. Two of them are from the vector form of the Navier–Stokes equation and the third is the continuity equation. Note that the first term in the Navier–Stokes equation is nonlinear.

Before we introduce the numerical scheme to solve this equation set, we still need to define the appropriate boundary condition. Let us assume that the domain of interest is D and the boundary of the domain is B. Because the solution and the existence of the solution are very sensitive to the boundary condition, we have to be very careful in selecting a valid one (Bristeau *et al.*, 1985). A common choice in numerical solution of the Navier–Stokes equation is to have an extended domain, $D_\mathrm{e} > D$, with a boundary B_e. Either the velocity

$$\mathbf{v}(x, y)=\mathbf{v}_\mathrm{e}(x, y) \tag{9.73}$$

for $\mathbf{n}\cdot\mathbf{v} < 0$ or the combination

$$P(x, y)\mathbf{n}-\eta\mathbf{n}\cdot\boldsymbol{\nabla}\mathbf{v}(x, y)=\mathbf{q}(x, y) \tag{9.74}$$

for $\mathbf{n}\cdot\mathbf{v} \geq 0$ is given at the boundary B_e. Here $\mathbf{n}$ is the normal unit vector at the boundary pointing outward. The weak form of the Navier–Stokes equation is obtained if we take a dot product of the equation with a vector function $\mathbf{u}(x, y)$ and then integrate both sides:

$$\int_{D_\mathrm{e}} (\rho\mathbf{v}\cdot\nabla\mathbf{v} + \nabla P - \eta\nabla^2\mathbf{v})\cdot\mathbf{u}\,dx\,dy = \int_{D_\mathrm{e}} \rho\mathbf{g}\cdot\mathbf{u}\,dx\,dy, \tag{9.75}$$

which can be transformed into the weak form with integration by parts of the Laplace term and the pressure gradient term. Then we have

$$\begin{aligned}\int_{D_\mathrm{e}} &[\rho(\mathbf{v}\cdot\nabla\mathbf{v})\cdot\mathbf{u} - P\nabla\cdot\mathbf{u} - \eta\nabla\mathbf{v} : \nabla\mathbf{u}]\,dx\,dy \\ &= \int \rho\mathbf{g}\cdot\mathbf{u}\,dx\,dy - \int_{B_\mathrm{e}} (P\mathbf{n} - \eta\mathbf{n}\cdot\nabla\mathbf{v})\cdot\mathbf{u}\,d\ell,\end{aligned} \tag{9.76}$$

where : denotes a double dot product, a combination of the dot product between ∇ and ∇ and the dot product between $\mathbf{v}$ and $\mathbf{u}$. The line integral is along the boundary with the domain on the left.

The weak form of the continuity equation can be obtained by multiplying the equation by a scalar function $f(x, y)$ and then integrating it in the domain D_e. We obtain

$$\int_{D_\mathrm{e}} [\nabla\cdot\mathbf{v}(x, y)]\, f(x, y)\,dx\,dy = 0, \tag{9.77}$$

which can be solved together with the weak form of the Navier–Stokes equation. The line integral along the boundary can be found with the given $\mathbf{v}(x, y)$ and $P(x, y)$ at the boundary or replaced with $\int_{B_\mathrm{e}} \mathbf{q}\cdot\mathbf{u}\,d\ell$ after applying Eq. (9.74). Either way, this integral is given after we have selected $\mathbf{u}(x, y)$.

Let us represent the approximate solutions by linear combinations of the local functions,

$$\mathbf{v}_n(x, y) = \sum_{i=1}^{n} \mathbf{e}_i u_i(x, y), \tag{9.78}$$

$$P_n(x, y) = \sum_{i=1}^{n} c_i w_i(x, y), \tag{9.79}$$

where $\mathbf{e}_i = (a_i, b_i)$ is a two-component vector corresponding to the two components of $\mathbf{v}$, and $u_i(x, y)$ and $w_i(x, y)$ are chosen to be different because, in the weak form, the pressure appears only in its own form but the velocity field also appears in the form of its first-order derivative. This means that we can choose a simpler function $w_i(x, y)$ (for example, differentiable once) than $u_i(x, y)$ (for example, differentiable twice) in order to have the same level of smoothness in $P(x, y)$ and $\mathbf{v}(x, y)$.

In order to convert the weak form into a matrix form, we will use $\mathbf{u} = (u_i, u_j)$ and $f(x, y) = w_i(x, y)$. More importantly, we have to come up with a scheme to deal with the nonlinear term. An iterative scheme can be devised as follows:

(1) We first make a guess of the solution, $a_i^{(0)}$, $b_i^{(0)}$, and $c_i^{(0)}$, for $i = 1, 2, \ldots, n$. This can be achieved in most cases by solving the problem with a mean-field type of approximation to the nonlinear term, for example,

$$\mathbf{v} \cdot \nabla \mathbf{v} \simeq \mathbf{v}_0 \cdot \nabla \mathbf{v}, \tag{9.80}$$

where $\mathbf{v}_0$ is a constant vector that is more or less the average of $\mathbf{v}$ over the whole domain.

(2) Then we can improve the solution iteratively. For the $(k + 1)$th iteration, we can use the result of the kth iteration for part of the nonlinear term; for example, the first term in the weak form of the Navier–Stokes equation can be written as $\mathbf{v}^{(k)} \cdot \nabla \mathbf{v}^{(k+1)}$. Then we can solve $a_i^{(k+1)}$, $b_i^{(k+1)}$, and $c_i^{(k+1)}$ for $i = 1, 2, \ldots, n$. The weak form at each iteration is a linear equation set, because $a_i^{(k)}$, $b_i^{(k)}$, and $c_i^{(k)}$ have already been solved in the previous step.

This iterative scheme is rather general. We can also devise similar schemes for solving other nonlinear equations with the nonlinear terms rewritten into linear terms with part of each term represented by the result of the previous step. Note that the scheme outlined here is similar to the scheme discussed in Chapter 5 for solving multivariable nonlinear problems. The scheme outlined here can also be interpreted as a special choice of a relaxation scheme (Fortin and Thomasset, 1983) with

$$\mathbf{v}^{(k+1)} = (1 - p)\mathbf{v}^{(k)} + p\mathbf{v}^{(k+1/2)}, \tag{9.81}$$

where $\mathbf{v}^{(k+1/2)}$ is the solution of the equation set

$$\rho \mathbf{v}^{(k)} \cdot \nabla \mathbf{v}^{(k+1/2)} + \nabla P^{(k+1)} - \eta \nabla^2 \mathbf{v}^{(k+1/2)} = \rho \mathbf{g}, \tag{9.82}$$

$$\nabla \cdot \mathbf{v}^{(k+1/2)} = 0. \tag{9.83}$$

Here p is a parameter that can be adjusted to achieve the best convergence.

An important aspect of the solution of the Navier–Stokes equation is the solution of the time-dependent equation, which requires us to combine the spatial solution just discussed and the initial-value problem discussed in Chapter 7. Interested readers can find detailed discussions on the numerical algorithms for solving the time-dependent Navier–Stokes equation in Bristeau *et al.* (1985).

9.6 The particle-in-cell method

In order to simulate a large continuous system such as a hydrodynamic fluid, we have to devise a method that can deal with the macroscopic phenomena observed. Finite element methods and finite difference methods can be used in the solution of macroscopic equations that describe the dynamics of continuous systems.

However, the behavior of the systems may involve different length scales, which will make a finite difference or finite element method quite difficult.

We can, in principle, treat each atom or molecule as an individual particle and then solve Newton's equations for all the particles in the many-body system. This can be a good method if the structural and dynamical properties under study are at the length scale of the average interparticle distance, which is clearly the case when we study the behavior of salt melting, glass formation, or structure factors of specific liquids. This was the subject of the preceding chapter on molecular dynamics simulation.

Schemes dealing with the simulation of the dynamics of each individual particle in the system will run into difficulty when applied to a typical hydrodynamic system, because such a system has more than 10^{23} particles and the phenomena take place at macroscopic length scales. The common practice is to solve the set of macroscopic equations discussed at the beginning of this chapter with either the finite difference method introduced in Chapter 7 or the finite element method discussed in this chapter. However, we cannot always do so because some dynamical phenomena, such as the nonlinear properties observed in plasma or galactic systems, may derive from the fundamental interaction in the system – for example, the Coulomb interaction between electrons and ions, or gravity between stars. More importantly, density fluctuations around a phase transition or the onset of chaotic behavior might involve several orders of magnitude in the length scale. So a scheme that can bridge the microscopic dynamics of each particle to the macroscopic phenomena in a system is highly desirable. The *particle-in-cell method* was first introduced by Evans and Harlow (1957) for this purpose. For a review on the early development of the method, see Harlow (1964). More recent progress can be found in Grigoryev, Vshivkov, and Fedoruk (2002) and in Büchner, Dum, and Scholer (2003).

Let us assume that the system that we are interested in is a large continuous system. We can divide the system into many cells, which are similar to the finite elements discussed earlier in this chapter. In each cell there are still a large number of particles. For example, if we are interested in a two-dimensional system with 10^{14} particles and divide it into one million cells, we still have 100 million particles in each cell. A common practice is to construct pseudo-particles, which are still collections of many true particles. For example, if we take one million particles as one pseudo-particle for the two-dimensional system with 10^{14} particles, we can easily divide the system into 10 000 cells, each of which has 10 000 particles. This becomes a manageable problem with currently available computing capacity. Note that because the pseudo-particles are still collections of particles, simulations with such pseudo-particles are not completely described by microscopic dynamics. However, because the division of the system into cells allows us to cover several orders of magnitude in length scales, we can simulate many phenomena that involve different length scales in macroscopic systems.

The size of the cells is usually determined by the relevant length scales in the system. If we are interested in a collection of charged particles, we must have each cell small enough that the detailed distribution of the charges in each cell does not affect the behavior of each particle or pseudo-particle very much. This means that the contribution to the energy of each particle due to the other charges in the same cell can be ignored in comparison with the dominant energy scale, for example, the thermal energy of each particle in this case. The electrostatic potential energy of a particle due to the uniform distribution of the charges around it inside a sphere with a radius a is given by

$$U_\mathrm{c} = ne^2a^2/\epsilon_0, \tag{9.84}$$

which should be much smaller than the thermal energy of one degree of freedom, $k_\mathrm{B}T/2$. The dimensions of the cell then must satisfy

$$a \ll \lambda_\mathrm{D} = \sqrt{\frac{\epsilon_0 k_\mathrm{B} T}{ne^2}}, \tag{9.85}$$

where λ_D is called the Debye length, e is the charge on each particle, and n is the number density. However, because we are interested in the hydrodynamics of the system, we cannot choose too small a cell. It has to be much larger than the average volume occupied by each particle, that is,

$$a \gg a_0 = \frac{1}{n^{1/3}}, \tag{9.86}$$

for a three-dimensional system. Similar relations can be established for other systems, such as galaxies. After the size of the cell is determined, we can decide whether or not a pseudo-particle picture is needed based on how many particles are in the cell. Basically, we have to make sure that the finite size effect of each cell will not dominate the behavior of the system under study.

Let us first sketch the idea of the method for a specific example: a three-dimensional charged particle system. If the cell size is much larger than the average distance between two nearest neighbors, the system is nearly collisionless. The dynamics of each particle or pseudo-particle is then governed by the instantaneous velocity $\mathbf{v}$ of the particle and the average electrostatic field $\mathbf{E}$ at the position of the particle. For example, at time $t = n\tau$, where τ is the time step, the ith particle has a velocity $\mathbf{v}_i^{(n)}$ and an acceleration

$$\mathbf{g}_i^{(n)} = e\mathbf{E}_i^{(n)}/m, \tag{9.87}$$

with e being the charge and m the mass of the particle. Then the next position $\mathbf{r}_i^{(n+1)}$ and velocity $\mathbf{v}_i^{(n+1)}$ of the same particle are given by

$$\mathbf{r}_i^{(n+1)} = \mathbf{r}_i^{(n)} + \tau\mathbf{v}_i^{(n)} + \frac{\tau^2}{2}\mathbf{g}_i^{(n)}, \tag{9.88}$$

$$\mathbf{v}_i^{(n+1)} = \mathbf{v}_i^{(n)} + \frac{\tau}{2}\left(\mathbf{g}_i^{(n)} + \mathbf{g}_i^{(n+1)}\right), \tag{9.89}$$

with the Verlet velocity algorithm. We have assumed that the charge and the mass of each particle or pseudo-particle are e and m, respectively. Note that there is no difference between the current situation and the molecular dynamics case except that we are dealing only with an average acceleration $\mathbf{g}_i$, which is an interpolated value from the fields at the grid points around the particle, instead of a detailed calculation from the particle–particle interactions. Most probably the "particles" here are pseudo-particles, each equivalent to many, many particles. It is worth pointing out that other numerical schemes developed in molecular dynamics can be applied here as well. As we have already discussed in molecular dynamics, the most time-consuming part of the simulation is the evaluation of the interactions between the particles. The basic idea of the particle-in-cell method is to circumvent such an effort. Instead, we try to solve the field on the grid points and then find the average field at the particle from the interpolation of the values from the grid points around it. For a charged particle system, this can be achieved by solving the Poisson equation for the electrostatic potential at the grid points, which are typically set at the centers of the cells. The accuracy of the interpolation kernel is the key to a stable and accurate algorithm. We will here incorporate the idea of the improved interpolation kernels, introduced by Monaghan (1985), into our discussions. For convenience of notation, we will use Greek letters for the indices for the grid points and Latin letters for the particles. The charge density on a specific grid point can be determined from the distribution of the particles in the system at the same moment, that is,

$$\rho^{(n+1)}(\mathbf{r}_\sigma) = e \sum_{i=1}^{N} \mathcal{W}\left(\mathbf{r}_\sigma - \mathbf{r}_i^{(n+1)}, h\right), \tag{9.90}$$

where $\mathcal{W}(\mathbf{r}, h)$ is the interpolation kernel, which can be interpreted as a distribution function and satisfies

$$\int \mathcal{W}(\mathbf{r}, h)\, d\mathbf{r} = 1, \tag{9.91}$$

and N is the total number of pseudo-particles in the system. Note that $\mathcal{W}(\mathbf{r}, h)$ is usually a function ranged with the grid spacing h. This is due to the fact that $\mathcal{W}(\mathbf{r}, h)$ has to be very small beyond $|\mathbf{r}| = h$. Monaghan (1985) has shown that we can improve interpolation accuracy by choosing

$$\mathcal{W}(\mathbf{r}, h) = \frac{1}{2}\left[(d+2)\mathcal{G}(\mathbf{r}, h) - h\frac{\partial \mathcal{G}(\mathbf{r}, h)}{\partial h}\right], \tag{9.92}$$

where $\mathcal{G}(\mathbf{r}, h)$ is a smooth interpolation function and d is the number of the spatial dimensions of the system. For example, if $\mathcal{G}(\mathbf{r}, h)$ is a Gaussian function

$$\mathcal{G}(\mathbf{r}, h) = \frac{1}{\pi^{3/2}} e^{-r^2/h^2}, \tag{9.93}$$

$\mathcal{W}(\mathbf{r}, h)$ is given by

$$\mathcal{W}(\mathbf{r}, h) = \frac{1}{\pi^{3/2}}\left(\frac{d+2}{2} - \frac{r^2}{h^2}\right) e^{-r^2/h^2}. \tag{9.94}$$

From the interpolated grid values of the density, we can solve the Poisson equation

$$\nabla^2 \Phi(\mathbf{r}) = -\rho(\mathbf{r})/\epsilon_0, \tag{9.95}$$

with any of the finite difference methods discussed in Chapter 7. Then the electric field $\mathbf{E} = -\nabla\Phi(\mathbf{r})$ at each grid point can be evaluated through, for example, a three-point or two-point formula. In order to have a more accurate value for the field at the particle site, we can also interpolate the potential at the particle site and then evaluate the field

$$\mathbf{E}_i^{(n+1)} = -\sum_{\sigma=1}^{z} \Phi_\sigma^{(n+1)} \nabla_\sigma \mathcal{U}(\mathbf{r}_i - \mathbf{r}_\sigma), \tag{9.96}$$

where $\mathcal{U}(\mathbf{r})$ is another kernel that can be either the same as or different from the kernel $\mathcal{W}(\mathbf{r}, h)$. The gradient is taken on the continuous function of $\mathbf{r}_\sigma$, and z is the number of grid points nearby that we would like to include in the interpolation scheme. For example, if only the nearest neighbors are included, $z = 4$ for a square lattice. The new velocity of the particle in the Verlet algorithm above is then given by

$$\mathbf{v}_i^{(n+1)} = \mathbf{v}_i^{(n)} + \frac{e\tau}{2m}\left(\mathbf{E}_i^{(n)} + \mathbf{E}_i^{(n+1)}\right). \tag{9.97}$$

The number of points to be included in the summation is determined from the dimensionality of the system and the geometry of the cells. Like $\mathcal{W}(\mathbf{r}, h)$, $\mathcal{U}(\mathbf{r})$ also has to be a smooth, even function of $\mathbf{r}$ and it must satisfy

$$\sum_{\sigma=1}^{z} \mathcal{U}(\mathbf{r}_\sigma - \mathbf{r}_i) = 1, \tag{9.98}$$

for any given $\mathbf{r}_i$. This can be done by normalizing $\mathcal{U}(\mathbf{r}_\sigma - \mathbf{r}_i)$ numerically at every time step. Note that the order of the position update and the velocity update in Eqs. (9.88) and (9.89) can be interchanged. In practice, this makes no difference to the numerical difficulty, but provides some flexibility in improving the accuracy for a specific problem.

So a typical particle-in-cell scheme has four major steps: solving the microscopic equations, interpolating microscopic quantities to the grid points, solving the macroscopic equations on the grid points, and then interpolating the macroscopic quantities back to the particles. The order of the steps may vary depending on the specific problem in question, but the goal is always the same – to avoid direct simulation of the system from the microscopic equations while still maintaining the input from the microscopic scales to the macroscopic phenomena. The particle-in-cell method is very powerful in many practical applications, ranging from plasma simulations to galactic dynamics. In the next section, we will discuss a typical application of the method in hydrodynamics and magnetohydrodynamics. For more details of the method, see Potter (1977), Hockney and Eastwood (1988), and Monaghan (1985).

9.7 Hydrodynamics and magnetohydrodynamics

Now let us turn to hydrodynamic systems. We have already discussed the equations of hydrodynamics at the beginning of this chapter. To simplify our discussion here, we will first consider the case of an ideal fluid, that is, one for which it can be assumed that the viscosity, temperature gradient, and external field in the system are all very small and can be ignored. The Navier–Stokes equation then becomes

$$\rho \frac{\partial \mathbf{v}}{\partial t} + \rho \mathbf{v} \cdot \nabla \mathbf{v} + \nabla P = 0, \tag{9.99}$$

where the second term is the convective (nonlinear) term and the third term results from the change of the pressure. The continuity equation is still the same,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0. \tag{9.100}$$

Because there is no dissipation, the equation for the internal energy per unit mass is given by

$$\rho \frac{\partial \varepsilon}{\partial t} + \rho \mathbf{v} \cdot \nabla \varepsilon + P \nabla \cdot \mathbf{v} = 0. \tag{9.101}$$

The pressure P, the density ρ, and the internal energy ε are all related by the equation of state

$$f(P, \rho, \varepsilon) = 0. \tag{9.102}$$

Sometimes entropy is a more convenient choice than the internal energy.

Once again we will incorporate the improved interpolation scheme of Monaghan (1985) into our discussion. We divide the hydrodynamic system into many cells, and in each cell we have many particles or pseudo-particles. Assuming that the equation of state and the initial conditions are given, we can set up the initial density, velocity, energy per unit mass, and pressure associated with the lattice points as well as the particles.

If we use the two-point formula for the partial time derivative and an interpolated pressure from an interpolation kernel $\mathcal{V}(\mathbf{r})$, the velocity at the lattice points can be partially updated as

$$\mathbf{v}_\sigma^{(n+1/2)} = \mathbf{v}_\sigma^{(n)} - \frac{\tau}{\rho_\sigma^{(n)}} \sum_\mu P_\mu \nabla_\sigma \mathcal{V}(\mathbf{r}_\sigma - \mathbf{r}_\mu), \tag{9.103}$$

which does not include the effect due to the convective term $\rho \mathbf{v} \cdot \nabla \mathbf{v}$. Here index μ runs through all the nearest neighbors of the lattice point $\mathbf{r}_\sigma$ and $\mathcal{V}(\mathbf{r}_\sigma - \mathbf{r}_\mu)$ is normalized as

$$\sum_\mu \mathcal{V}(\mathbf{r}_\sigma - \mathbf{r}_\mu) = 1. \tag{9.104}$$

We will explain later that the velocity at the lattice points is modified by convection to

$$\mathbf{v}_\sigma^{(n+1)} = \frac{1}{2} \left(\mathbf{v}_\sigma^{(n)} + \mathbf{v}_\sigma^{(n+1/2)} \right) \tag{9.105}$$

if the particle positions are updated as discussed below. The velocity $\mathbf{v}_\sigma^{(n+1)}$ can be used to update the internal energy partially at the lattice points with

$$\varepsilon_\sigma^{(n+1/2)} = \varepsilon_\sigma^{(n)} - \frac{\tau P_\sigma^{(n)}}{\rho_\sigma^{(n)}} \sum_\mu \mathbf{v}_\mu^{(n+1)} \cdot \nabla_\sigma \mathcal{V}(\mathbf{r}_\sigma - \mathbf{r}_\mu), \tag{9.106}$$

which does not include the effect of the convective term $\rho \mathbf{v} \cdot \nabla \varepsilon$ in Eq. (9.101). This is also done with the application of the two-point formula to the partial time derivative.

The total thermal energy per unit mass at the lattice points with the partial updates of the velocity and the internal energy is then given by

$$e_\sigma^{(n+1/2)} = \varepsilon_\sigma^{(n+1/2)} + \frac{1}{2}\left(v_\sigma^{(n+1/2)}\right)^2, \tag{9.107}$$

which in turn can be used to interpolate the energy per unit mass at the particle sites,

$$e_j^{(n+1/2)} = \sum_{\sigma=1}^{z} e_\sigma^{(n+1/2)} \mathcal{U}(\mathbf{r}_i - \mathbf{r}_\sigma), \tag{9.108}$$

where $\mathcal{U}(\mathbf{r})$ is the interpolation kernel from lattice points to particle positions used in the preceding section. The new particle velocity is obtained from the interpolation

$$\mathbf{v}_i^{(n+1)} = \sum_{\sigma=1}^{z} \mathbf{v}_\sigma^{(n+1)} \mathcal{U}(\mathbf{r}_i - \mathbf{r}_\sigma), \tag{9.109}$$

which can then be used to update the particle positions with a simple two-point formula

$$\mathbf{r}_i^{(n+1)} = \mathbf{r}_i^{(n)} + \tau \mathbf{v}_i^{(n+1)}, \tag{9.110}$$

which completes the convective motion. We can easily show that the updating formula for the particle position is exactly the same as in the Verlet algorithm with

$$\mathbf{r}_i^{(n+1)} = \mathbf{r}_i^{(n)} + \tau \mathbf{v}_i^{(n)} + \frac{\tau^2}{2} \mathbf{g}_i^{(n)}, \tag{9.111}$$

where

$$\mathbf{g}(\mathbf{r}) = -\frac{1}{\rho(\mathbf{r})} \boldsymbol{\nabla} P(\mathbf{r}). \tag{9.112}$$

This has exactly the same mathematical structure as the electrostatic field discussed in the preceding section. The gradient of the pressure at the particle position can therefore be obtained from the pressure at the lattice points with

$$\boldsymbol{\nabla} P_i = \sum_{\sigma=1}^{z} P_\sigma \boldsymbol{\nabla}_\sigma \mathcal{U}(\mathbf{r}_\sigma - \mathbf{r}_i). \tag{9.113}$$

We can show that the updated particle velocity is also equivalent to that of the Verlet algorithm with

$$\mathbf{v}_i^{(n+1)} = \mathbf{v}_i^{(n)} + \frac{\tau}{4}\left(\mathbf{g}_i^{(n)} + \mathbf{g}_i^{(n+1)}\right), \tag{9.114}$$

to the same order of accuracy with the convective term included. Note that the second term is only half of the corresponding term in the nonconvective case.

Now we can update the quantities at the lattice points. Following the discussions in the preceding section, we have

$$\rho_\sigma^{(n+1)} = m\sum_{i=1}^{N}\mathcal{W}\left(\mathbf{r}_\sigma - \mathbf{r}_i^{(n+1)}\right), \tag{9.115}$$

$$\rho_\sigma^{(n+1)}\mathbf{v}_\sigma^{(n+1)} = m\sum_{i=1}^{N}\mathbf{v}_i\mathcal{W}\left(\mathbf{r}_\sigma - \mathbf{r}_i^{(n+1)}\right), \tag{9.116}$$

$$\rho_\sigma^{(n+1)}e_\sigma^{(n+1)} = m\sum_{i=1}^{N}e_i^{(n+1)}\mathcal{W}\left(\mathbf{r}_\sigma - \mathbf{r}_i^{(n+1)}\right), \tag{9.117}$$

$$\varepsilon_\sigma^{(n+1)} = e_\sigma^{(n+1)} - \frac{1}{2}\left(v_\sigma^{(n+1)}\right)^2. \tag{9.118}$$

The values of ρ_σ and e_σ can then be used in the equation of state to obtain the new values for the pressure at the lattice points. Then the steps outlined above can be repeated for the next time step.

The above scheme can easily be extended to include viscosity and magnetic field effects. The Navier–Stokes equation is then given by

$$\frac{\partial}{\partial t}(\rho\mathbf{v}) = -\nabla\cdot\mathbf{\Pi}, \tag{9.119}$$

where $\mathbf{\Pi} = \rho\mathbf{v}\mathbf{v} - \mathbf{\Gamma}$ with

$$\mathbf{\Gamma} = \eta\left[\nabla\mathbf{v} + (\nabla\mathbf{v})^{\mathrm{T}}\right] + \left[\left(\zeta - \frac{2\eta}{3}\right)\nabla\cdot\mathbf{v} - P - \frac{B^2}{2\mu}\right]\mathbf{I} + \frac{\mathbf{B}\mathbf{B}}{\mu} \tag{9.120}$$

and the magnetic field satisfies

$$\frac{\partial\mathbf{B}}{\partial t} = \nabla\times\left[\mathbf{v}\times\mathbf{B} - \frac{\boldsymbol{\rho}}{\mu}\cdot(\nabla\times\mathbf{B})\right], \tag{9.121}$$

where $\boldsymbol{\rho}$ is the resistivity tensor of the system. The energy equation is then given by

$$\frac{\partial}{\partial t}\left(\varepsilon + \rho\frac{v^2}{2} + \frac{B^2}{2\mu}\right) = \nabla\cdot\mathbf{j}_{\mathrm{e}}, \tag{9.122}$$

where

$$\mathbf{j}_{\mathrm{e}} = \mathbf{v}\left(\rho\varepsilon + \frac{1}{2}\rho v^2 + \frac{B^2}{2\mu}\right) - \mathbf{v}\cdot\mathbf{\Gamma} - \kappa\nabla T \tag{9.123}$$

is the energy current. All these extra terms can be incorporated into the particle-in-cell scheme outlined earlier in this section. We will not go into more details here, but interested readers can find discussions in Monaghan (1985).

9.8 The lattice Boltzmann method

As discussed so far, continuous systems can properly be represented by discrete models if the choice of discretization still accounts for the basic physical processes involved in the systems. This basic idea was put forward by Frisch, Hasslacher, and Pomeau (1986), Frisch *et al.* (1987), and Wolfram (1986) in a model now known as lattice-gas cellular automata. The model treats the system as a lattice gas and the occupancy at each site of the lattice as a Boolean number, 0 or 1, corresponding to an unoccupied or occupied state of a few possible velocities. We can recover the macroscopic equations of fluid dynamics from the microscopic Boltzmann equation under the constraint of mass and momentum conservation. This lattice-gas model has the advantage of using Boolean numbers in simulations and therefore is free of rounding errors. Another advantage of the lattice-gas model is that it is completely local and can easily be implemented in parallel or distributed computing environments. However, the lattice-gas model also has some disadvantages in simulations, such as large fluctuations in the density distribution and other physical quantities due to the discrete occupancy of each site and an exponential increase with the number of nearest vertices in the collision rules. These two disadvantages were overcome by the introduction of the lattice Boltzmann method (McNamara and Zanetti, 1988; Higuera and Jiménez, 1989), which preserves most of the advantages of the lattice-gas model in simulations but allows a smooth occupancy, which reduces the fluctuations. The introduction of the BGK (Bhatnagar–Gross–Krook) form of the collision term in the Boltzmann equation also reduces the exponential complexity in the collision rules with the number of the nearest vertices.

Before discussing the lattice-gas model and the lattice Boltzmann method, we first give a brief outline of the Boltzmann kinetic theory. All the ideas behind the classic kinetic theory are based on the Boltzmann transport equation. Assume that $f(\mathbf{r}, \mathbf{v}, t)$ is the distribution function at the point $(\mathbf{r}, \mathbf{v})$ in the phase space at time t. Then the number of particles in the phase-space volume element $d\mathbf{r}\, d\mathbf{v}$ is $f(\mathbf{r}, \mathbf{v}, t)\, d\mathbf{r}\, d\mathbf{v}$. Because the number of particles and the volume element in the phase space are conserved in equilibrium based on Liouville's theorem, we have

$$\frac{df(\mathbf{r}, \mathbf{v}, t)}{dt} = 0 \qquad (9.124)$$

if the particles in the system do not interact with each other. Furthermore, both $\mathbf{r}$ and $\mathbf{v}$ are functions of time and we can then rewrite the above derivative as

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \mathbf{v} \cdot \boldsymbol{\nabla} f + \mathbf{g} \cdot \boldsymbol{\nabla}_{\mathrm{v}} f = 0, \qquad (9.125)$$

where we have used the definition of the velocity $\mathbf{v} = d\mathbf{r}/dt$, the acceleration $\mathbf{g} = d\mathbf{v}/dt$, and the gradient in velocity $\boldsymbol{\nabla}_{\mathrm{v}} = \partial/\partial\mathbf{v}$ for convenience. We will consider only the zero external field case, that is, $\mathbf{g} = \mathbf{0}$, for simplicity. The above equation will not hold if the collisions in the system cannot be ignored. Instead, the equation is modified to

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f = \Omega(\mathbf{r}, \mathbf{v}, t), \tag{9.126}$$

where $\Omega(\mathbf{r}, \mathbf{v}, t)$ is a symbolic representation of the effects of all the collisions between the particles in the system. The collision contribution $\Omega(\mathbf{r}, \mathbf{v}, t)$ can be expressed in terms of an integral that involves the product of $f(\mathbf{r}, \mathbf{v}, t)$ at different velocities as well as the scattering cross section of many-body collisions in the integrand. We will not derive this expression here because we will need only the sum rules of $\Omega(\mathbf{r}, \mathbf{v}, t)$. The simplest approximation is the BGK form with

$$\Omega(\mathbf{r}, \mathbf{v}, t) = -\frac{f(\mathbf{r}, \mathbf{v}, t) - f_0(\mathbf{r}, \mathbf{v}, t)}{\tau}, \tag{9.127}$$

where f_0 is the equilibrium distribution and τ is the relaxation time due to the collisions. The conservation laws are reflected in the sum rules of $\Omega(\mathbf{r}, \mathbf{v}, t)$. For example, the conservation of the total mass ensures

$$\int m\Omega(\mathbf{r}, \mathbf{v}, t)\, d\mathbf{r}\, d\mathbf{v} = 0, \tag{9.128}$$

and the conservation of the linear momentum would require

$$\int m\mathbf{v}\Omega(\mathbf{r}, \mathbf{v}, t)\, d\mathbf{r}\, d\mathbf{v} = 0. \tag{9.129}$$

We can also obtain the hydrodynamic equations from the Boltzmann equation (Lifshitz and Pitaevskii, 1981) after taking the statistical averages of the relevant physical quantities. The point we want to make is that the macroscopic behavior of a dynamic system is the result of the average effects of all the particles involved.

As we have pointed out, it is impossible to simulate 10^{23} particles from their equations of motion with current computing capacity. Alternative methods have to be devised in the simulation of hydrodynamics. As we discussed earlier in this chapter, we can use the particle-in-cell method to simulate hydrodynamic systems.

The idea of the lattice-gas model resembles the particle-in-cell concept in two respects: the particles in the lattice are pseudo-particles, and the lattice sites are similar to the grid points in the particle-in-cell schemes. The difference is in the assignment of velocities to the particles. In the lattice-gas model, only a few discrete velocities are allowed. Let us take a triangular lattice as an illustrative example. At each site, there cannot be more than seven particles, each of which has to have a unique velocity

$$\mathbf{v}_\sigma = \frac{\mathbf{e}_\sigma}{\tau}, \tag{9.130}$$

where $\mathbf{e}_\sigma$ is one of the vectors pointing to the nearest neighboring vertices or 0, that is,

$$|\mathbf{v}_\sigma| = \begin{cases} a/\tau & \text{for } \sigma = 1, \ldots, 6, \\ 0 & \text{for } \sigma = 0, \end{cases} \tag{9.131}$$

where a is the distance to a nearest neighbor. Here the index σ runs from 0 to z, where z is the number of nearest neighbors of a lattice point. Now if we take τ as the unit of time and $f_\sigma(\mathbf{r}, t)$ as the occupancy of the site $\mathbf{r}$ in the state with velocity $\mathbf{v}_\sigma$, the Boltzmann equation for each lattice point becomes

$$f_\sigma(\mathbf{r} + \mathbf{e}_\sigma, t + 1) - f_\sigma(\mathbf{r}, t) = \Omega_\sigma[f_\mu(\mathbf{r}, t)], \tag{9.132}$$

with $\sigma, \mu = 0, 1, \ldots, z$. Note that f_σ here is a Boolean number that can only be 0 for the unoccupied state or 1 for the occupied state. Now if we require the mass and momentum at each site to be constant as $\tau \to 0$, we have

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{v}), \tag{9.133}$$

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) = -\nabla \cdot \mathbf{\Pi}, \tag{9.134}$$

where ρ is the local density given by

$$\rho(\mathbf{r}, t) = m \sum_{\sigma=0}^{z} f_\sigma(\mathbf{r}, t), \tag{9.135}$$

$\rho\mathbf{v}$ is the corresponding momentum density with

$$\rho(\mathbf{r}, t)\mathbf{v}(\mathbf{r}, t) = m \sum_{\sigma=0}^{z} \mathbf{v}_\sigma f_\sigma(\mathbf{r}, t), \tag{9.136}$$

and

$$\mathbf{\Pi} = m \sum_{\sigma=0}^{z} \mathbf{v}_\sigma \mathbf{v}_\sigma f_\sigma(\mathbf{r}, t) \tag{9.137}$$

is the momentum flux density. We can show that the above equations leads to a macroscopic equation that resembles the Navier–Stokes equation if we perform the Chapman–Enskog expansion of $f_\sigma(\mathbf{r}, t)$ up to the τ^2 terms (Frisch *et al.*, 1987; Wolfram, 1986).

Two very important aspects of lattice-gas automaton simulations are the geometry of the lattice and the collision rules at each vertex. For example, we have used a triangular lattice in two dimensions in order to have isotropic macroscopic behavior. Another popular choice is to use a square lattice to have nine choices of velocities associated with the eight displacement vectors to nearest neighbors and next nearest neighbors and zero velocity. It is more difficult to ensure isotropic macroscopic behavior in three dimensions. One way to have a macroscopic isotropy is to simulate a three-dimensional projection of a four-dimensional face-centered cubic lattice. For more details on lattice-gas automata, see the articles in Doolen (1991).

Now we turn to the lattice Boltzmann method. As we have stated, the distribution at each site for a specific state is assumed to be a smooth function of the position instead of a Boolean number. We will take

$$n_\sigma(\mathbf{r}, t) = \langle f_\sigma(\mathbf{r}, t) \rangle \tag{9.138}$$

as the distribution function. The lattice Boltzmann equation then becomes

$$n_\sigma(\mathbf{r}+\mathbf{e}_\sigma, t+1) - n_\sigma(\mathbf{r}, t) = \Omega_\sigma[n_\mu(\mathbf{r}, t)] \tag{9.139}$$

for $\sigma, \mu = 0, 1, \ldots, z$. The collision term can be further approximated by keeping only the linear term of the deviation of the distribution from its equilibrium value (Higuera and Jiménez, 1989), and then we have

$$n_\sigma(\mathbf{r}+\mathbf{e}_\sigma, t+1) - n_\sigma(\mathbf{r}, t) = \sum_{\mu=0}^{z} \Delta_{\sigma\mu}\left[n_\mu(\mathbf{r}, t) - n_\mu^0(\mathbf{r}, t)\right], \tag{9.140}$$

where the matrix Δ is determined from the symmetry of the lattice and conservation of mass and momentum. We can expand the equilibrium distribution function $n_\sigma^0(\mathbf{r}, t)$ into a power series of $\mathbf{v}$, and then we can show that macroscopic equations resemble hydrodynamic equations. The selection of the parameters in the expansion can also lead to the equation of state for a specific system under study (Chen, Chen, and Matthaeus, 1992). Interested readers can find more detailed discussions in the review by Chen and Doolen (1998) and books by Wolf-Gladrow (2000), Succi (2001), and Zhou (2004)

Exercises

9.1 Derive Eqs. (9.1), (9.2), and (9.3) by considering an infinitesimal cubic element in a bulk fluid. Find their modifications if the particles are charged and the system is under the influence of external gravitational and electromagnetic fields.

9.2 Solve the one-dimensional Poisson equation

$$\phi''(x) = -\rho(x)/\epsilon_0,$$

by the Galerkin method. Assume that all the quantities are in the SI units, the boundary condition is $u(0) = u(1) = 0$, and the density distribution is given by

$$\rho(x)/\epsilon_0 = \begin{cases} x^2 & \text{if } x \in [0, 0.5], \\ (1-x)^2 & \text{if } x \in [0.5, 1]. \end{cases}$$

Use the basis functions

$$u_i(x) = \begin{cases} (x - x_{i-1})/h & \text{if } x \in [x_{i-1}, x_i], \\ (x_{i+1} - x)/h & \text{if } x \in [x_i, x_{i+1}], \\ 0 & \text{otherwise,} \end{cases}$$

to construct the matrix $\mathbf{A}$ and the vector $\mathbf{b}$. Write a program that solves the resulting equation set through the LU decomposition.

9.3 Consider the electrostatic problem within an equilateral triangle with side length of 2.0 m. The potentials at the sides are 0 V, 0.5 V, and 1.0 V. The charge distribution is zero at the highest potential side and linearly increases to ϵ_0 at the opposite angle. Construct an equally spaced triangular grid with lines parallel to the sides with a total of eleven points on each side. Find the electrostatic potential at the field points through the finite element method with the pyramid basis functions that reach zero at the nearest neighboring points.

9.4 Consider a rope stretched between two anchoring points 3.0 m apart, subject to a tension $T = 1.00 \times 10^3$ N, on which a person is sitting. Assume that the mass distribution (mass per unit length) of the system is given by $m(x) = m_0 e^{-(x-x_0)^2/a^2}$, where $m_0 = 80.0$ kg/m, $a = 0.30$ m, and $x_0 = 1.0$ m from one end of the rope. Find the wave equation that describes the motion of the rope. What is the weak form of the wave equation? Solve the displacement of the rope under the equilibrium condition with a finite element scheme. Determine the three lowest vibrational frequencies of the system.

9.5 Solve the Helmholtz equation

$$\phi''(x) + k^2\phi(x) = -s(x)$$

through its weak form. Use the boundary condition $\phi(0) = \phi(1) = 0$ and assume that $k = \pi$ and $s(x) = \delta(x - 0.5)$. Is the solution of the weak form the same as the solution of the equation?

9.6 Show that the optimization of the generalized functional introduced in the text for the Sturm–Liouville equation with the natural boundary condition can produce the equation with the correct boundary condition. Construct a simple basis function set that is suitable for a finite element solution of the equation. Find the coefficient matrix and the constant vector in the corresponding linear equation set with the natural boundary condition for different α and β. Develop a computer program that implements the scheme. Test the program with the Schrödinger equation for the one-dimensional harmonic oscillator.

9.7 Show that the optimization of the functional introduced for the two-dimensional Helmholtz equation with the general boundary condition can produce the equation with the correct boundary condition. Construct a simple basis function set that is suitable for a finite element solution of the equation. Find the coefficient matrix and the constant vector in the corresponding linear equation set with the general boundary condition for different functions $\alpha(x, y)$ and $\beta(x, y)$. Develop a computer program that implements the scheme. Test the program with the electrostatic problem given in Exercise 9.3.

9.8 Show that the finite element solution for a specific element in two dimensions with triangular elements and linear local pyramidal functions is given by Eq. (9.63). Work out the coefficient matrix and the constant vector if the two-dimensional space is divided into a triangular lattice with the lattice constant h.

9.9 Develop a computer program that solves the two-dimensional, stationary Navier–Stokes equation described in Section 9.5. Test your program for a rectangular domain under a given boundary condition. How sensitive is the solution to the variance in the boundary condition?

9.10 An infinite cylindrical conducting shell of radius a is cut along its axis into four equal parts, which are then insulated from each other. If the potential on the first and third quarters is V and the potential on the second and fourth quarters is $-V$, find the potential in the perpendicular plane through a finite element method. Compare the numerical result with the exact solution.

9.11 Obtain the Debye length for a galaxy. Estimate the size of the cell needed if it is simulated by the particle-in-cell method. Develop a computer program to simulate the dynamics of a galaxy. Use the parameters associated with the Milky Way to test the program.

9.12 Consider a two-dimensional nonviscous fluid with the same equation of state as that of an adiabatic ideal gas. Develop a particle-in-cell scheme and a computer program to simulate this system. Assume that there are 10^6 particles in the system with 100 in each cell on average.

9.13 Construct a lattice-gas model for a square lattice with nine choices of velocity at each site from the eight displacement vectors to the nearest neighbors and next nearest neighbors and zero velocity.

9.14 Apply the lattice Boltzmann method to simulate a two-dimensional nonviscous fluid with the same equation of state as that of an adiabatic ideal gas. Based on the simulation results, discuss the differences found between a seven-velocity triangular lattice model and a nine-velocity square lattice model.

Chapter 10
Monte Carlo simulations

One of the major numerical techniques developed in the last half century for evaluating multidimensional integrals or solving integral equations is the Monte Carlo method. The basic idea of the method is to select points in the region enclosed by the boundary and then take the weighted data as the estimated value of the integral. Early Monte Carlo simulations go back to the 1950s, when the first computers became available. In this chapter, we will introduce the basic idea of the Monte Carlo method with applications in statistical physics, quantum mechanics, and other related fields, highlighting some recent developments.

10.1 Sampling and integration

Let us first use a simple example to illustrate how a basic Monte Carlo scheme works. If we want to find the numerical value of the integral

$$S = \int_0^1 f(x)\, dx, \tag{10.1}$$

we can simply divide the region [0, 1] evenly into M slices with $x_0 = 0$ and $x_M = 1$, and then the integral can be approximated as

$$S = \frac{1}{M} \sum_{n=1}^{M} f(x_n) + O(h^2), \tag{10.2}$$

which is equivalent to sampling from a set of points $x_1, x_2, \dots, x_M$ in the region [0, 1] with an equal weight, in this case, 1, at each point. We can, on the other hand, select x_n with $n = 1, 2, \dots, M$ from a uniform random number generator in the region [0, 1] to accomplish the same goal. If M is very large, we would expect x_n to be a set of numbers uniformly distributed in the region [0, 1] with fluctuations proportional to $1/\sqrt{M}$. Then the integral can be approximated by the average

$$S \simeq \frac{1}{M} \sum_{n=1}^{M} f(x_n), \tag{10.3}$$

where x_n is a set of M points generated from a uniform random number generator in the region [0, 1]. Note that the possible error in the evaluation of the integral is now given by the fluctuation of the distribution x_n. If we use the standard deviation of statistics to estimate the possible error of the random sampling, we have

$$(\Delta S)^2 = \frac{1}{M}\left(\langle f_n^2\rangle - \langle f_n\rangle^2\right). \tag{10.4}$$

Here the average of a quantity is defined as

$$\langle A_n\rangle = \frac{1}{M}\sum_{n=1}^{M} A_n, \tag{10.5}$$

where A_n is the sampled data. We have used $f_n = f(x_n)$ for notational convenience.

Now we illustrate how the scheme works in an actual numerical example. In order to demonstrate the algorithm clearly, let us take a very simple integrand $f(x) = x^2$. The exact result of the integral is 1/3 for this simple example. The following program implements such a sampling.

```java
// An example of integration with direct Monte Carlo
// scheme with integrand f(x) = x*x.

import java.lang.*;
import java.util.Random;
public class Monte {
  public static void main(String argv[]) {
    Random r = new Random();
    int n = 1000000;
    double s0 = 0;
    double ds = 0;
    for (int i=0; i<n; ++i) {
      double x = r.nextDouble();
      double f = x*x;
      s0 += f;
      ds += f*f;
    }
    s0 /= n;
    ds /= n;
    ds = Math.sqrt(Math.abs(ds-s0*s0)/n);
    System.out.println("S = " + s0 + " +- " + ds);
  }
}
```

We have used the uniform random number generator in Java for convenience. The numerical result and estimated error from the above program is 0.3334 ± 0.0003. More reliable results can be obtained from the average of several independent runs. For example, with four independent runs, we obtain an average of 0.3332 with an estimated error of 0.0002.

The simple Monte Carlo quadrature used in the above program does not show any advantage. For example, the trapezoid rule yields much higher accuracy with the same number of points. The reason is that the error from the Monte Carlo quadrature is

$$\Delta S \propto \frac{1}{M^{1/2}}, \tag{10.6}$$

whereas the trapezoid rule yields an estimated error of

$$\Delta S \propto \frac{1}{M^2}, \tag{10.7}$$

which is much smaller for a large M. The true advantage of the Monte Carlo method comes in the evaluation of multidimensional integrals. For example, if we are interested in a many-body system, such as the electrons in a neon atom, we have ten particles, and the integral for the expectation value can have as many as 30 dimensions. For a d-dimensional space, the Monte Carlo quadrature will still yield the same error behavior, that is, it is proportional to $1/\sqrt{M}$, where M is the number of points sampled. However, the trapezoid rule has an error of

$$\Delta S \propto \frac{1}{M^{2/d}}, \tag{10.8}$$

which becomes greater than the Monte Carlo error estimate with the same number of points if d is greater than 4. The point is that the Monte Carlo quadrature produces a much more reasonable estimate of a d-dimensional integral when d is very large. There are some specially designed numerical quadratures that would work still better than the Monte Carlo quadrature when d is slightly larger than 4. However, for a real many-body system, the dimensionality in an integral is $3N$, where N is the number of particles in the system. When N is on the order of 10 or larger, any other workable quadrature would perform worse than the Monte Carlo quadrature.

10.2 The Metropolis algorithm

From the discussion in the preceding section, we see that for an arbitrary integrand such as $f(x) = x^2$ in the region [0, 1], the accuracy of the integral evaluated from the Monte Carlo quadrature is very low. We used 10^6 points in the numerical example and obtained an accuracy of only 0.1%. Is there any way to increase the accuracy for some specific types of integrands?

If $f(x)$ were a constant, we would need only a single point to obtain the exact result from the Monte Carlo quadrature. Or if $f(x)$ is smooth and close to constant, the accuracy from the Monte Carlo quadrature would be much higher. Here we consider a more general case in which there are $3N$ variables, that is, $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$, with each $\mathbf{r}_i$ for $i = 1, 2, \ldots, N$ being a three-dimensional vector. The $3N$-dimensional integral is then written as

$$S = \int_D F(\mathbf{R})\, d\mathbf{R}, \tag{10.9}$$

where D is the domain of the integral. In many cases, the function $F(\mathbf{R})$ is not a smooth function. The idea of importance sampling introduced by Metropolis *et al.* (1953) is to sample the points from a nonuniform distribution. If a distribution function $\mathcal{W}(\mathbf{R})$ can mimic the drastic changes in $F(\mathbf{R})$, we should expect a much faster convergence with

$$S \simeq \frac{1}{M}\sum_{i=1}^{M}\frac{F(\mathbf{R}_i)}{\mathcal{W}(\mathbf{R}_i)}, \tag{10.10}$$

where M is the total number of points of the configurations $\mathbf{R}_i$ sampled according to the distribution function $\mathcal{W}(\mathbf{R})$.

Now we show some of the details of this sampling scheme. We can, of course, rewrite the integral as

$$S = \int \mathcal{W}(\mathbf{R})G(\mathbf{R})\,d\mathbf{R}, \tag{10.11}$$

where $\mathcal{W}(\mathbf{R})$ is positive definite and satisfies the normalization condition

$$\int \mathcal{W}(\mathbf{R})\,d\mathbf{R} = 1, \tag{10.12}$$

which ensures that $\mathcal{W}(\mathbf{R})$ can be viewed as a probability function. From the two expressions of the integral we have $G(\mathbf{R}) = F(\mathbf{R})/\mathcal{W}(\mathbf{R})$. The problem is solved if $G(\mathbf{R})$ is a smooth function of $\mathbf{R}$, that is, nearly a constant. We can imagine a statistical process that leads to an equilibrium distribution $\mathcal{W}(\mathbf{R})$ and the integral S is merely a statistical average of $G(\mathbf{R})$. This can be compared with the canonical ensemble average

$$\langle A\rangle = \int A(\mathbf{R})\mathcal{W}(\mathbf{R})\,d\mathbf{R}, \tag{10.13}$$

where $A(\mathbf{R})$ denotes the physical quantities to be averaged and the probability or distribution function $\mathcal{W}(\mathbf{R})$ is given by

$$\mathcal{W}(\mathbf{R}) = \frac{e^{-U(\mathbf{R})/k_{\mathrm{B}}T}}{\int e^{-U(\mathbf{R}')/k_{\mathrm{B}}T}\,d\mathbf{R}'}, \tag{10.14}$$

with $U(\mathbf{R})$ being the potential energy of the system for a given configuration $\mathbf{R}$. Here k_{B} is the Boltzmann constant and T is the temperature of the system.

A procedure introduced by Metropolis *et al.* (1953) is extremely powerful in the evaluation of the multidimensional integral defined in Eq. (10.11). Here we give a brief outline of the procedure and refer to it as the *Metropolis algorithm*. The selection of the sampling points is viewed as a Markov process. In equilibrium, the values of the distribution function at different points of the phase space are related by

$$\mathcal{W}(\mathbf{R})T(\mathbf{R}\to\mathbf{R}') = \mathcal{W}(\mathbf{R}')T(\mathbf{R}'\to\mathbf{R}), \tag{10.15}$$

where $T(\mathbf{R} \to \mathbf{R}')$ is the transition rate from the state characterized by $\mathbf{R}$ to the state characterized by $\mathbf{R}'$. This is usually referred to as *detailed balance* in statistical mechanics. Now the points are no longer sampled randomly but rather by following the Markov chain. The transition from one point $\mathbf{R}$ to another point $\mathbf{R}'$ is accepted if the ratio of the transition rates satisfies

$$\frac{T(\mathbf{R} \to \mathbf{R}')}{T(\mathbf{R}' \to \mathbf{R})} = \frac{\mathcal{W}(\mathbf{R}')}{\mathcal{W}(\mathbf{R})} \geq w_i, \tag{10.16}$$

where w_i is a uniform random number in the region [0, 1]. Let us here outline the steps used for the evaluation of the integral defined in Eq. (10.11) or Eq. (10.13). We first randomly select a configuration $\mathbf{R}_0$ inside the specified domain D. Then $\mathcal{W}(\mathbf{R}_0)$ is evaluated. A new configuration $\mathbf{R}_1$ is tried with

$$\mathbf{R}_1 = \mathbf{R}_0 + \Delta\mathbf{R}, \tag{10.17}$$

where $\Delta\mathbf{R}$ is a $3N$-dimensional vector with each component from a uniform distribution between $[-h, h]$, which is achieved, for example, with

$$\Delta x_i = h(2\eta_i - 1), \tag{10.18}$$

for the x component of $\mathbf{r}_i$. Here η_i is a uniform random number generated in the region [0, 1]. The actual value of the step size h is determined from the desired accepting rate (the ratio of the accepted to the attempted steps). A large h will result in a small accepting rate. In practice, h is commonly chosen so that the accepting rate of moves is around 50%. After all the components of $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ have been tried, we can evaluate the distribution function at the new configuration $\mathbf{R}_1$, which is then accepted according to the probability

$$p = \frac{\mathcal{W}(\mathbf{R}_1)}{\mathcal{W}(\mathbf{R}_0)}, \tag{10.19}$$

that is, comparing p with a uniform random number w_i in the region [0, 1]. If $p \geq w_i$, the new configuration is accepted; otherwise, the old configuration is assumed to be the new configuration. This procedure is repeated and the physical quantity $A(\mathbf{R}_k)$ for $k = n_1, n_1 + n_0, \ldots, n_1 + (M-1)n_0$ is evaluated. The numerical result of the integral is then given by

$$\langle A \rangle \simeq \frac{1}{M} \sum_{l=0}^{M-1} A(\mathbf{R}_{n_1+n_0 l}). \tag{10.20}$$

Note that n_1 steps are used to remove the influence of the initial selection of the configuration, that is, $\mathbf{R}_0$. The data are taken n_0 steps apart to avoid high correlation between the data points, because they are generated consecutively and a reasonable number of points must be skipped when the next data point is taken.

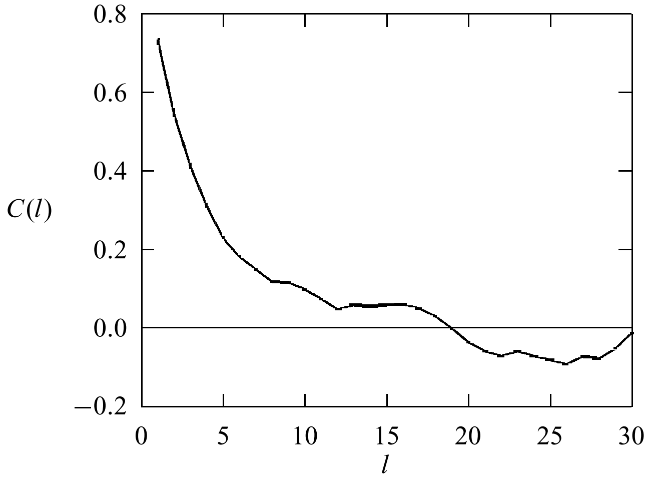


Fig. 10.1 Autocorrelation function obtained from the numerical example given in the text.

We can, however, minimize the correlation and at the same time the number of steps needed to reach a required accuracy by analyzing the autocorrelation function of $A(\mathbf{R}_n)$ for $n = 1, 2, \ldots$. The autocorrelation function is defined as

$$C(l) = \frac{\langle A_{n+l}A_n\rangle - \langle A_n\rangle^2}{\langle A_n^2\rangle - \langle A_n\rangle^2}, \tag{10.21}$$

where the average is taken from consecutive steps, that is,

$$\langle A_{n+l}A_n\rangle = \frac{1}{M}\sum_{n=1}^{M} A_{n+l}A_n. \tag{10.22}$$

We have used A_n for $A(\mathbf{R}_n)$ to simplify the notation. A typical $C(l)$, obtained from the numerical example later in this section is shown in Fig. 10.1. As we can see, the autocorrelation function starts to decrease at the beginning and then saturates around $l = l_c \simeq 12$, which is a good choice of the number of steps to be skipped, $n_0 = l_c$, for the two adjacent data points to be used in the sampling.

In most cases, the distribution function $\mathcal{W}(\mathbf{R})$ varies by several orders of magnitude, whereas $A(\mathbf{R})$ stays smooth or nearly constant. The sampling scheme outlined above will find the average of a physical quantity according to the distribution $\mathcal{W}(\mathbf{R})$. In other words, there will be more points showing up with configurations of higher $\mathcal{W}(\mathbf{R})$. This *sampling by importance* is much more efficient than the direct, random sampling presented in the preceding section.

Now we would like to compare this procedure numerically with the direct, random sampling. We still consider the integral

$$S = \int_0^1 f(x)\,dx = \int_0^1 \mathcal{W}(x)g(x)\,dx, \tag{10.23}$$

with $f(x) = x^2$. We can choose the distribution function as

$$\mathcal{W}(x) = \frac{1}{\mathcal{Z}}\left(e^{x^2} - 1\right), \tag{10.24}$$

which is positive definite. The normalization constant $\mathcal{Z}$ is given by

$$\mathcal{Z} = \int_0^1 \left(e^{x^2} - 1\right) dx = 0.462\,651\,67, \tag{10.25}$$

which is calculated from another numerical scheme for convenience. Then the corresponding function $g(x) = f(x)/\mathcal{W}(x)$ is given by

$$g(x) = \mathcal{Z}\frac{x^2}{e^{x^2} - 1}. \tag{10.26}$$

Now we are ready to put all of these into a program that is a realization of the Metropolis algorithm for the integral specified.

```
// An example of Monte Carlo simulation with the
// Metropolis scheme with integrand f(x) = x*x.

import java.lang.*;
import java.util.Random;
public class Carlo {
  static final int nsize = 10000;
  static final int nskip = 15;
  static final int ntotal = nsize*nskip;
  static final int neq = 10000;
  static int iaccept = 0;
  static double x, w, h = 0.4, z = 0.46265167;
  static Random r = new Random();
  public static void main(String argv[]) {
    x = r.nextDouble();
    w = weight();
    for (int i=0; i<neq; ++i) metropolis();

    double s0 = 0;
    double ds = 0;
    iaccept = 0;
    for (int i=0; i<ntotal; ++i) {
      metropolis();
      if (i%nskip == 0) {
        double f = g(x);
        s0 += f;
        ds += f*f;
      }
    }
    s0 /= nsize;
    ds /= nsize;
    ds = Math.sqrt(Math.abs(ds-s0*s0)/nsize);
    s0 *= z;
    ds *= z;
    double accept = 100.0*iaccept/(ntotal);
    System.out.println("S = " + s0 + " +- " + ds);
    System.out.println("Accept rate = " + accept + "%");
  }
  public static void metropolis() {
    double xold = x;
    x = x + 2*h*(r.nextDouble()-0.5);
    if ((x<0) || (x>1)) x = xold;
    else {
      double wnew = weight();
      if (wnew > w*r.nextDouble()) {
        w = wnew;
        ++iaccept;
      }
      else x = xold;
    }
  }

  public static double weight() {
    return Math.exp(x*x) - 1;
  }

  public static double g(double y) {
    return y*y/(Math.exp(y*y)-1);
  }
}
```

The numerical result obtained with the above program is 0.3334 ± 0.0005. The step size is adjustable, and we should try to keep it such that the accepting rate of the new configurations is less than or around 50%. In practice, it seems that such a choice of accepting rate is compatible with considerations of speed and accuracy. The above program is structured so that we can easily modify it to study other problems.

10.3 Applications in statistical physics

The Metropolis algorithm was first applied in the simulation of the structure of classical liquids and is still used in research into the study of glass transitions and polymer systems. In this section, we discuss some applications of the Metropolis algorithm in statistical physics. We will use the classical liquid system and the Ising model as the two illustrative examples. The classical liquid system is continuous in spatial coordinates of atoms or molecules in the system, whereas the Ising model is a discrete lattice system.

Structure of classical liquids

First we will consider the classical liquid system. Let us assume that the system is in good contact with a thermal bath and has a fixed number of particles and volume size: that is, that the system is described by the canonical ensemble. Then the average of a physical quantity A is given by

$$\langle A\rangle = \frac{1}{\mathcal{Z}}\int A(\mathbf{R})e^{-U(\mathbf{R})/k_B T}\,d\mathbf{R}, \tag{10.27}$$

where $\mathcal{Z}$ is the partition function of the system, which is given by

$$\mathcal{Z} = \int e^{-U(\mathbf{R})/k_B T}\, d\mathbf{R}. \tag{10.28}$$

Here $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ is a $3N$-dimensional vector for the coordinates of all the particles in the system. We have suppressed the velocity dependence in the expression with the understanding that the distribution of the velocity is given by the Maxwell distribution, which can be used to calculate the averages of any velocity-related physical quantities. For example, the average kinetic energy component of a particle is given by

$$\left\langle \frac{m v_{ik}^2}{2} \right\rangle = \frac{1}{2} k_B T, \tag{10.29}$$

where $i = 1, 2, \ldots, N$ is the index for the ith particle and $k = 1, 2, 3$ is the index for the directions x, y, and z. The average of any physical quantity associated with velocity can be obtained through the partition theorem. For example, a quantity $B(\mathbf{V})$ with $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N)$ can always be expanded in the Taylor series of the velocities, and then each term can be evaluated with

$$\left\langle v_{ik}^n \right\rangle = \begin{cases} n k_B T/m & \text{if } n \text{ is even,} \\ 0 & \text{otherwise.} \end{cases} \tag{10.30}$$

Now we can concentrate on the evaluation of a physical quantity that depends on the coordinates only, such as the total potential energy, the pair distribution, or the pressure. The average of the physical quantity is then given by

$$\langle A \rangle = \frac{1}{\mathcal{Z}} \int A(\mathbf{R}) e^{-U(\mathbf{R})/k_B T}\, d\mathbf{R} = \frac{1}{M} \sum_{i=1}^{M} A(\mathbf{R}_i), \tag{10.31}$$

where $\mathbf{R}_i$ is a set of points in the phase space sampled according to the distribution function

$$\mathcal{W}(\mathbf{R}) = \frac{e^{-U(\mathbf{R})/k_B T}}{\mathcal{Z}}. \tag{10.32}$$

Assuming that we have a pairwise interaction between any two particles, the total potential energy for a specific configuration is then given by

$$U(\mathbf{R}) = \sum_{j>i}^{N} V(r_{ij}). \tag{10.33}$$

In the preceding section, we discussed how to update all the components in the old configuration in order to reach a new configuration. However, sometimes it is more efficient to update the coordinates of just one particle at a time in the old configuration to reach the new configuration, especially when the system is very close to equilibrium or near a phase transition. Here we show how to update the coordinates of one particle to obtain the new configuration. The coordinates for the ith particle are updated with

$$x_i^{(n+1)} = x_i^{(n)} + h_x(2\alpha_i - 1), \tag{10.34}$$

$$y_i^{(n+1)} = y_i^{(n)} + h_y(2\beta_i - 1), \tag{10.35}$$

$$z_i^{(n+1)} = z_i^{(n)} + h_z(2\gamma_i - 1), \tag{10.36}$$

where h_k is the step size along the kth direction and α_i, β_i, and γ_i are uniform random numbers in the region [0, 1]. The acceptance of the move is determined by importance sampling, that is, by comparing the ratio

$$p = \frac{\mathcal{W}(\mathbf{R}_{n+1})}{\mathcal{W}(\mathbf{R}_n)} \tag{10.37}$$

with a uniform random number w_i. The attempted move is accepted if $p \geq w_i$ and rejected if $p < w_i$. Note that in the new configuration only the coordinates of the ith particle are moved, so we do not need to evaluate the whole $U(\mathbf{R}_{n+1})$ again in order to obtain the ratio p for the pairwise interactions. We can express the ratio in terms of the potential energy difference between the old configuration and the new configuration as

$$p = e^{-\Delta U/k_B T}, \tag{10.38}$$

with

$$\Delta U = \sum_{j \neq i}^{N} \left[V\left(\left| \mathbf{r}_i^{(n+1)} - \mathbf{r}_j^{(n)} \right| \right) - V\left(\left| \mathbf{r}_i^{(n)} - \mathbf{r}_j^{(n)} \right| \right) \right], \tag{10.39}$$

which can usually be truncated for particles within a distance $r_{ij} \leq r_c$. Here r_c is the typical distance at which the effect of the interaction $V(r_c)$ is negligible. For example, the interaction in a simple liquid is typically the Lennard–Jones potential, which decreases drastically with the separation of the two particles. The typical distance for the truncation in the Lennard–Jones potential is about $r_c = 3\sigma$, where σ is the separation of the zero potential. As we discussed earlier, we should not take *all* the discrete data points in the sampling for the average of a physical quantity, because the autocorrelation of the data is very high if the points are not far apart. Typically, we need to skip about 10–15 data points before taking another value for the average. The evaluations of physical quantities such as total potential energy, structural factor, and pressure are performed almost the same as in the molecular dynamics if we treat the Monte Carlo steps as the time steps in the molecular dynamics.

Another issue for the simulation of infinite systems is the extension of the finite box with a periodic boundary condition. Long-range interactions, such as the Coulomb interaction, cannot be truncated; a summation of the interaction between particles in all the periodic boxes is needed. The Ewald sum is used to include the interactions between a charged particle and the images in other boxes under the periodic boundary condition. Discussion on the Ewald summation can be found in standard textbooks of solid-state physics, for example, Madelung (1978).

The Monte Carlo step size h is determined from the desired rejection rate. For example, if we want 70% of the moves to be rejected, we can adjust h to satisfy such a rejection rate. A larger h produces a higher rejection rate. In practice, it is clear that a higher rejection rate produces data points with less fluctuation. However, we also have to consider the computing time needed. The decision is made according to the optimization of the computing time and the accuracy of the data sought. Typically, 50–75% used as the rejection rate in the most Monte Carlo simulations. Here h along each different direction, that is, h_x, h_y, or h_z, is determined according to the length scale along that direction. An isotropic system would have $h_x = h_y = h_z$. A system confined along the z direction, for example, would need a smaller h_z.

Properties of lattice systems

Now let us turn to discrete model systems. The scheme is more or less the same. The difference comes mainly from the way the new configuration is tried. Because the variables are discrete, instead of moving the particles in the system, we need to update the configuration by changing the local state of each lattice site. Here we will use the classical three-dimensional Ising model as an illustrative example. The Hamiltonian is

$$\mathcal{H} = -J\sum_{\langle ij\rangle}^{N} s_i s_j - B\sum_{i=1}^{N} s_i, \tag{10.40}$$

where J is the exchange interaction strength, B is the external field, $\langle ij\rangle$ denotes the summation over all nearest neighbors, and N is the total number of sites in the system. The spin s_i for $i = 1, 2, \ldots, N$ can take values of either 1 or -1, so the summation for the interactions is for energies carried by all the bonds between nearest neighboring sites.

The Ising model was used historically to study magnetic phase transitions. The magnetization is defined as

$$m = \frac{1}{N}\sum_{i=1}^{N}\langle s_i\rangle, \tag{10.41}$$

which is a function of the temperature and external magnetic field. For the $B = 0$ case, there is a critical temperature T_c that separates different phases of the system. For example, the system is ferromagnetic if $T < T_c$, paramagnetic if $T > T_c$, and unstable if $T = T_c$. The complete plot of T, m, and B forms the so-called phase diagram. Another interesting application of the Ising model is that it is also a generic model for binary lattices: that is, two types of particles can occupy the lattice sites with two on-site energies which differ by $2B$. So the results obtained from the study of the Ising model apply also to other systems in its class, such as binary lattices. Readers who are interested in these aspects can find good discussions in Parisi (1988). Other quantities of interest include, but are not limited to, the total energy

$$E = \langle \mathcal{H} \rangle, \tag{10.42}$$

and the specific heat

$$C = \frac{\langle \mathcal{H}^2 \rangle - \langle \mathcal{H} \rangle^2}{N k_\mathrm{B} T^2}. \tag{10.43}$$

In order to simulate the Ising model, for example, in the calculation of m, we can apply more or less the same idea as for the continuous system. The statistical average of the spin at each site is given by

$$m = \frac{1}{\mathcal{Z}} \sum_\sigma s_\sigma e^{-\mathcal{H}_\sigma / k_\mathrm{B} T}, \tag{10.44}$$

where $s_\sigma = S_\sigma / N$, with $S_\sigma = \sum s_i$ being the total spin of a specific configuration labeled by σ and $\mathcal{H}_\sigma$ being the corresponding Hamiltonian (energy). The summation in Eq. (10.44) is over all the possible configurations. Here $\mathcal{Z}$ is the partition function given by

$$\mathcal{Z} = \sum_\sigma e^{-\mathcal{H}_\sigma / k_\mathrm{B} T}. \tag{10.45}$$

The average of a physical quantity, such as the magnetization, can be obtained from

$$m \simeq \frac{1}{M} \sum_{\sigma=1}^{M} s_\sigma, \tag{10.46}$$

with $\sigma = 1, 2, \ldots, M$ indicating the configurations sampled according to the distribution function

$$\mathcal{W}(S_\sigma) = \frac{\exp[-\mathcal{H}_\sigma / k_\mathrm{B} T]}{\mathcal{Z}}. \tag{10.47}$$

Let us consider the actual simulations in more detail. First we randomly assign 1 and -1 to the spins on all the lattice sites. Then we select one site, which can be picked either randomly or sequentially, to be updated. Assume that the ith site is selected. The update is attempted with the spin at the site reversed: that is,

$$s_i^{(n+1)} = -s_i^{(n)}, \tag{10.48}$$

which is accepted into the new configuration if

$$p = e^{-\Delta \mathcal{H} / k_\mathrm{B} T} \geq w_i, \tag{10.49}$$

where w_i is a uniform random number in the region [0, 1] and $\Delta \mathcal{H}$ is the energy difference caused by the reversal of the spin, given by

$$\Delta \mathcal{H} = -2 J s_i^{(n+1)} \sum_{j=1}^{z} s_j^{(n)}. \tag{10.50}$$

Here j runs over all the nearest neighboring sites of i. Note that the quantity $\sum s_j^{(n)}$ associated with a specific site may be stored and updated during the simulation. Every time a spin is reversed and its new value is accepted, we can update the summations accordingly. If the reversal is rejected in the new configuration, no update is needed. The detailed balance condition

$$\mathcal{W}(S_\sigma)T(S_\sigma \to S_\mu) = \mathcal{W}(S_\mu)T(S_\mu \to S_\sigma) \tag{10.51}$$

does not determine the transition rate $T(S_\sigma \to S_\mu)$ uniquely; therefore we can also use the other properly normalized probability for the Metropolis steps, for example

$$q = \frac{1}{1 + e^{\Delta\mathcal{H}/k_{\mathrm{B}}T}} \tag{10.52}$$

instead of p in the simulation without changing the equilibrium results. The use of q speeds up the convergence at higher temperature. There has been considerable discussion on Monte Carlo simulations for the Ising model or related lattice models, such as percolation models. Interested readers can find these discussions in Binder and Heermann (1988), and Binder (1986; 1987; 1992).

10.4 Critical slowing down and block algorithms

There is a practical problem with the Metropolis algorithm in statistical physics when the system under study is approaching a critical point. Imagine that the system is a three-dimensional Ising system. At very high temperature, almost all the spins are uncorrelated, and therefore flipping of a spin has a very high chance of being accepted into the new configuration. So the different configurations can be accessed rather quickly and provide an average close to ergodic behavior. However, when the system is moved toward the critical temperature from above, domains with lower energy start to form. If the interaction is ferromagnetic, we start to see large clusters with all the spins pointing in the same direction. Now, if only one spin is flipped, the configuration becomes much less favorable because of the increase in energy. The favorable configurations are the ones with all the spins in the domain reversed, a point that it will take a very long time to reach. It means that we need to have all the spins in the domain flipped. This requires a very long sequence of accepted moves. This is known as the *critical slowing down*.

Another way to view it is from the autocorrelation function of the total energy of the system. Because now most steps are not accepted in generating new configurations, the total energy evaluated at each Monte Carlo step is highly correlated. In fact, the relaxation time needed to have the autocorrelation function decreased to near zero, t_{c}, is proportional to the power of the correlation length ξ, which diverges at the critical point in a bulk system. We have

$$t_{\mathrm{c}} = c\xi^z, \tag{10.53}$$

where c is a constant and the exponent z is called the *dynamical critical exponent*, which is about 2, estimated from standard Monte Carlo simulations. Note that the correlation length is bounded by the size of the simulation box L. Then the relaxation time is

$$t_c = cL^z, \tag{10.54}$$

when the system is very close to the critical point.

The first solution of this problem was devised by Swendsen and Wang (1987). Their block update scheme is based on the nearest neighbor pair picture of the Ising model, in which the partition function is a result of the contributions of all the nearest pairs of sites in the system. Let us examine closely the effect of a specific pair between sites i and j on the total partition function. We express the partition function of the system in terms of $\mathcal{Z}_p$, the partition function from the rest of the pairs with spins at sites i and j parallel, and $\mathcal{Z}_f$, the corresponding partition function without any restriction on the spins at sites i and j. Then the total partition function of the system is

$$\mathcal{Z} = q\mathcal{Z}_p + (1-q)\mathcal{Z}_f, \tag{10.55}$$

with

$$q = e^{-4\beta J}, \tag{10.56}$$

which can be interpreted as the probability of having a pair of correlated nearest neighbors decoupled. This is the basis on which Swendsen and Wang devised the block algorithm to remove the critical slowing down.

Here is a summary of their algorithm. We first cluster the sites next to each other that have the same spin orientation. A bond is introduced for any pair of nearest neighbors in a cluster. Each bond is then removed with probability q. After all the bonds are tried with some removed and the rest kept, there are more, but smaller, clusters still connected through the remaining bonds. The spins in each small cluster are flipped all together with a probability of 50%. The new configuration is then used for the next Monte Carlo step. This procedure in fact updates the configuration by flipping blocks of parallel spins, which is similar to the physical process that we would expect when the system is close to the critical point. The speedup in this block algorithm is extremely significant. Numerical simulations show that the dynamical critical exponent is significantly reduced, with $z \simeq 0.35$ for the two-dimensional Ising model and $z \simeq 0.53$ for the three-dimensional Ising model, instead of $z \simeq 2$ in the spin-by-spin update scheme.

The algorithm of Swendsen and Wang is important but still does not eliminate the critical slowing down completely, because the time needed to reach an uncorrelated energy still increases with the correlation length. An algorithm devised by Wolff (1989) provides about the same improvement in the two-dimensional Ising model as the Swendsen–Wang algorithm but greater improvement in the three-dimensional Ising model. More interestingly, the Wolff algorithm eliminates the critical slowing down completely, that is, $z \simeq 0$, in the four-dimensional Ising model, while the Swendsen–Wang algorithm has $z \simeq 1$.

The idea of Wolff is very similar to the idea of Swendsen and Wang. Instead of removing bonds, Wolff proposed constructing a cluster in which the nearest sites have the same spin orientation. We first select a site randomly from the system and add its nearest neighbors with the same spin orientation to the cluster with the probability $p = 1 - q$. This is continued, with sites added to the cluster, until all the sites in the system have been tried. All the spins in the constructed cluster are then flipped together to reach a new spin configuration of the system. We can easily show that the Wolff algorithm ensures detailed balance.

The Swendsen–Wang and Wolff algorithms can also be generalized to other spin models, such as the xy model and the Heisenberg model (Wolff, 1989). The idea is based on the construction of the probability q in both of the above-mentioned algorithms. At every step, we first generate a random unit vector $\mathbf{e}$ with each component a uniform random number x_i,

$$\mathbf{e} = \frac{1}{r}(x_1, x_2, \dots, x_n), \tag{10.57}$$

with

$$r^2 = \sum_{i=1}^{n} x_i^2, \tag{10.58}$$

where n is the dimension of the vector space of the spin. We can then define the probability of breaking a bond as

$$q = \min\{1, e^{-4\beta J(\mathbf{e}\cdot\mathbf{s}_i)(\mathbf{e}\cdot\mathbf{s}_j)}\}, \tag{10.59}$$

which reduces to Eq. (10.56) for the Ising model. The Swendsen–Wang algorithm and the Wolff algorithm have also shown significant improvement in the xy model, the Heisenberg model, and the Potts model. For more details, see Swendsen, Wang, and Ferrenberg (1992).

10.5 Variational quantum Monte Carlo simulations

The Monte Carlo method used in the simulations of classical many-body systems can be generalized to study quantum systems as well. In this section, we will introduce the most direct generalization of the Metropolis algorithm in the framework of the variational principle. We will start our discussion with a general quantum many-body problem, that is, the approximate solution of the Hamiltonian

$$\mathcal{H} = \sum_{i=1}^{N} \left[-\frac{\hbar^2}{2m_i}\nabla_i^2 + U_{\text{ext}}(\mathbf{r}_i) \right] + \sum_{i>j} V(\mathbf{r}_i, \mathbf{r}_j), \tag{10.60}$$

where $-\hbar^2\nabla_i^2/(2m_i)$ is the kinetic energy and $U_{\text{ext}}(\mathbf{r}_i)$ is the external potential of the ith particle and $V(\mathbf{r}_i, \mathbf{r}_j)$ is the interaction potential between the ith and jth particles. In most cases, the interaction is spherically symmetric, that is,

$V(\mathbf{r}_i, \mathbf{r}_j) = V(r_{ij})$. The masses of all particles are the same in an identical-particle system, of course.

We can symbolically write the time-independent many-body Schrödinger equation as

$$\mathcal{H}\Psi_n(\mathbf{R}) = E_n\Psi_n(\mathbf{R}), \tag{10.61}$$

where $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ and $\Psi_n(\mathbf{R})$ and E_n are the nth eigenstate and its corresponding eigenvalue of $\mathcal{H}$. We usually cannot obtain analytic or exact solutions for a system with more than two particles due to the complexity of the interaction potential. Approximate schemes are thus important tools for studying the different aspects of the many-body problem. It is easier to study the ground state of a many-body system. Properties associated with the ground state include the order of the state, collective excitations, and structural information.

Based on the variational principle that any other state would have a higher expectation value of the total energy than the ground state of system, we can always introduce a trial state $\Phi(\mathbf{R})$ to approximate the ground state. Here $\Phi(\mathbf{R})$ can be a parameterized function or some specific function form. The parameters or the variational function in $\Phi(\mathbf{R})$ can be optimized through the variational principle with

$$E[\alpha_i] = \frac{\langle\Phi|\mathcal{H}|\Phi\rangle}{\langle\Phi|\Phi\rangle} \geq E_0, \tag{10.62}$$

where α_i can be a set of variational parameters or functions of $\mathbf{R}$, which are linearly independent. This inequality can easily be shown by expanding $\Phi(\mathbf{R})$ in terms of the eigenstates of the Hamiltonian with

$$\Phi(\mathbf{R}) = \sum_{n=0}^{\infty} a_n\Psi_n(\mathbf{R}). \tag{10.63}$$

The expansion above is a generalization of the Fourier theorem, because $\Psi_n(\mathbf{R})$ forms a complete basis set. The variational principle results if we substitute the expansion for $\Phi(\mathbf{R})$ into the evaluation of the expectation value in Eq. (10.62) with the understanding that

$$E_n \geq E_0 \tag{10.64}$$

for $n > 0$ and that the eigenstates of a Hermitian operator can be made orthonormal with

$$\langle\Psi_n|\Psi_m\rangle = \delta_{nm}. \tag{10.65}$$

In order to obtain the optimized wavefunction, we can treat the variational parameters or specific functions in the variational wavefunction as independent variables in the Euler–Lagrange equation

$$\frac{\delta E[\alpha_i]}{\delta\alpha_j} = 0, \tag{10.66}$$

for $j = 1, 2, 3, \ldots$. To simplify our discussion, let us assume that the system is a continuum and that α_i is a set of variational parameters. So we can write the expectation value in the form of an integral

$$E[\alpha_i] = \frac{\int \Phi^\dagger(\mathbf{R})\mathcal{H}\Phi(\mathbf{R})\, d\mathbf{R}}{\int |\Phi(\mathbf{R}')|^2\, d\mathbf{R}'} = \int \mathcal{W}(\mathbf{R})\mathcal{E}(\mathbf{R})\, d\mathbf{R}, \tag{10.67}$$

where

$$\mathcal{W}(\mathbf{R}) = \frac{|\Phi(\mathbf{R})|^2}{\int |\Phi(\mathbf{R}')|^2\, d\mathbf{R}'} \tag{10.68}$$

can be interpreted as a distribution function and

$$\mathcal{E}(\mathbf{R}) = \frac{1}{\Phi(\mathbf{R})}\mathcal{H}\Phi(\mathbf{R}) \tag{10.69}$$

can be viewed as a local energy of a specific configuration $\mathbf{R}$. The expectation value can then be evaluated by the Monte Carlo quadrature if the expressions for $\mathcal{E}(\mathbf{R})$ and $\mathcal{W}(\mathbf{R})$ for a specific set of parameters α_i are given. In practice, $\Phi(\mathbf{R})$ is parameterized to have the important physics built in. The variational parameters α_i in the trial wavefunction $\Phi(\mathbf{R})$ are then optimized with the minimization of the expectation value $E[\alpha_i]$. It needs to be emphasized that the variational process usually cannot lead to any new physics and that the physics is in the variational wavefunction, which is constructed intuitively. However, in order to observe the physics of a nontrivial wavefunction, one usually needs to carry out the calculations.

A common choice of the trial wavefunction for quantum liquids has the general form

$$\Phi(\mathbf{R}) = D(\mathbf{R})e^{-U(\mathbf{R})}, \tag{10.70}$$

where $D(\mathbf{R})$ is a constant for boson systems and a Slater determinant of single-particle orbitals for fermion systems to meet the Pauli principle. Here $U(\mathbf{R})$ is the Jastrow correlation factor, which can be written in terms of one-body terms, two-body terms, and so on, with

$$U(\mathbf{R}) = \sum_{i=1}^{N} u_1(\mathbf{r}_i) + \sum_{i>j}^{N} u_2(\mathbf{r}_i, \mathbf{r}_j) + \sum_{i>j>k}^{N} u_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \cdots, \tag{10.71}$$

which is usually truncated at the two-body terms in most Monte Carlo simulations. Both $u_1(\mathbf{r})$ and $u_2(\mathbf{r}, \mathbf{r}')$ are parameterized according to the physical understanding of the source of these terms. We can show that when the external potential becomes a dominant term, $u_1(\mathbf{r})$ is uniquely determined by the form of the external potential, and when the interaction between the ith and jth particles dominates, the form of $u_2(\mathbf{r}_i, \mathbf{r}_j)$ is uniquely determined by $V(\mathbf{r}_i, \mathbf{r}_j)$. These are the important aspects used in determining the form of u_1 and u_2 in practice.

First let us consider the case of bulk helium liquids. Because the external potential is zero, we can choose $u_1(\mathbf{r}) = 0$. The two-body term is translationally invariant, $u_2(\mathbf{r}_i, \mathbf{r}_j) = u_2(r_{ij})$. The expression for $u_2(r)$ at the limit of $r \to 0$ can be obtained by solving a two-body problem. In the center-of-mass coordinate system, we have

$$\left[-\frac{\hbar^2}{2\mu}\nabla^2 + V(r)\right] e^{-u_2(r)} = E e^{-u_2(r)}, \tag{10.72}$$

where μ is the reduced mass $m/2$, $V(r)$ is the interaction potential that is given by $4\varepsilon(\sigma/r)^{12}$ at the limit of $r \to 0$, and the Laplacian can be decoupled in the angular momentum eigenstates as

$$\nabla^2 = \frac{d^2}{dr^2} + \frac{2}{r}\frac{d}{dr} - \frac{l(l+1)}{r^2}. \tag{10.73}$$

We can show that in order for the divergence of the potential energy to be canceled by the kinetic energy at the limit of $r \to 0$, we must have

$$u_2(r) = \left(\frac{a}{r}\right)^5. \tag{10.74}$$

The condition needed to remove the divergence of the potential energy with the kinetic energy term constructed from the wavefunction is called the *cusp condition*. This condition is extremely important in all quantum Monte Carlo simulations, because it is the major means of stabilizing the algorithms. The behavior of $u_2(r)$ at longer range is usually dominated by the density fluctuation or zero-point motion of phonons, which is proportional to $1/r^2$. We can show that the three-body term and the effects due to the backflow are also important and can be incorporated into the variational wavefunction. The Slater determinant for liquid ^{3}He can be constructed from plane waves. The key, as we have stressed earlier, is to find a variational wavefunction that contains the essential physics of the system, which is always nontrivial. In one study, Bouchaud and Lhuillier (1987) replaced the Slater determinant with a BCS (Bardeen–Cooper–Schrieffer) ansatz for liquid ^{3}He. This changed the structure of the ground state of the system from a Fermi liquid, which is characterized by a Fermi surface with a finite jump in the particle distribution, to a superconducting glass, which does not have a Fermi surface. For more details on the variational Monte Carlo simulations of helium systems, see Ceperley and Kalos (1986).

Electronic systems are another class of systems studied with the variational quantum Monte Carlo method. Significant results are obtained for atomic systems, molecules, and even solids. Typical approaches treat electrons and nuclei separately with the so-called Born–Oppenheimer approximation: that is, the electronic state is adjusted quickly for a given nuclear configuration so the potential due to the nuclei can be treated as an external potential of the electronic system. Assume that we can obtain the potential of the nuclei or ions with some other method. Then $u_1(\mathbf{r}_i)$ can be parameterized to ensure the cusp condition between an electron and a nucleus or ion. For an ion with an effective charge Ze, we have

$$u_1(r) = Zr/a_0 \tag{10.75}$$

as $r \to 0$. Here a_0 is the Bohr radius. $u_2(r_{ij})$ is obtained from the electron–electron interaction, and the cusp condition requires

$$u_2(r_{ij}) = -\frac{\sigma_{ij} r_{ij}}{2a_0} \tag{10.76}$$

as $r_{ij} \to 0$. Here $\sigma_{ij} = 1$ if the ith and jth electrons have different spin orientations; otherwise $\sigma_{ij} = 1/2$. The Slater determinant can be constructed from the local orbitals, for example, the linear combinations of the Gaussian orbitals from all nuclear sites. Details of how to construct the kinetic energy for the Jastrow ansatz with the Slater determinant can be found in Ceperley, Chester, and Kalos (1977).

There have been a lot of impressive variational quantum Monte Carlo simulations for electronic systems, for example, the work of Fahy, Wang, and Louie (1990) on carbon and silicon solids, the work of Umrigar, Wilson, and Wilkins (1988) on the improvement of variational wavefunctions; and the work of Umrigar (1993) on accelerated variational Monte Carlo simulations.

The numerical procedure for performing variational quantum Monte Carlo simulations is similar to that for the Monte Carlo simulations of statistical systems. The only difference is that for the statistical system the calculations are done for a given temperature, but for the quantum system they are done for a given set of variational parameters in the variational wavefunction. We can update either the whole configuration or just the coordinates associated with a particular particle at each Metropolis step.

10.6 Green's function Monte Carlo simulations

As we pointed out in the preceding section, the goal of the many-body theory is to understand the properties of a many-body Hamiltonian under given conditions. Assuming that we are dealing with an identical-particle system with an external potential $U_{\text{ext}}(\mathbf{r})$, the Hamiltonian is then given by

$$\mathcal{H} = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N} U_{\text{ext}}(\mathbf{r}_i) + \sum_{i<j}^{N} V(\mathbf{r}_i, \mathbf{r}_j), \tag{10.77}$$

where m is the mass of each particle and $V(\mathbf{r}_i, \mathbf{r}_j)$ is the interaction potential between the ith and jth particles. We will also assume that the interaction is spherically symmetric: that is, that $V(\mathbf{r}_i, \mathbf{r}_j) = V(r_{ij})$. As we discussed in the preceding section, the formal solution of the Schrödinger equation can be written as

$$\mathcal{H}\Psi_n(\mathbf{R}) = E_n\Psi_n(\mathbf{R}), \tag{10.78}$$

where $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ and $\Psi_n(\mathbf{R})$ and E_n are the nth eigenstate and its corresponding eigenvalue of $\mathcal{H}$. For convenience of discussion, we will set $\hbar = m = 1$.

It is impossible to solve the above Hamiltonian analytically for most cases. Numerical simulations are the alternatives in the study of many-body Hamiltonians. In the preceding section, we discussed the use of the variational quantum Monte Carlo simulation scheme to obtain the approximate solution for the ground state. However, the result is usually limited by the variational wavefunction selected. In this section, we show that we can go one step further to sample the ground-state wavefunction and for some systems we can even find the exact ground state. First we will discuss in more detail a version of the Green's function Monte Carlo scheme that treats the ground state of the Schrödinger equation as the stationary solution of a diffusion equation, the so-called imaginary-time Schrödinger equation. That is why this specific version of the Green's function Monte Carlo scheme is commonly referred to as the diffusion quantum Monte Carlo method. Later we will briefly discuss the original version of the Green's function Monte Carlo scheme, which actually samples Green's function. The imaginary-time Schrödinger equation is given by

$$\frac{\partial \Psi(\mathbf{R}, t)}{\partial t} = -(\mathcal{H} - E_{\rm c})\Psi(\mathbf{R}, t), \tag{10.79}$$

where t is the imaginary time, $\Psi(\mathbf{R}, t)$ is a time-dependent wavefunction, and $E_{\rm c}$ is an adjustable constant that is used later in the simulation to ensure that the overlap of $\Psi(\mathbf{R}, t)$ and $\Psi_0(\mathbf{R})$ is on the order of 1. We can show that at a later time the wavefunction is given by the convolution

$$\Psi(\mathbf{R}, t) = \int G(\mathbf{R}, \mathbf{R}'; t - t')\Psi(\mathbf{R}', t')\, d\mathbf{R}', \tag{10.80}$$

where Green's function $G(\mathbf{R}, \mathbf{R}'; t - t')$ is given by

$$\left[\frac{\partial}{\partial t} - (\mathcal{H} - E_{\rm c})\right] G(\mathbf{R}, \mathbf{R}'; t - t') = \delta(\mathbf{R} - \mathbf{R}')\delta(t - t'), \tag{10.81}$$

which can be expressed as

$$G(\mathbf{R}, \mathbf{R}'; t - t' = \langle \mathbf{R}| \exp[-(\mathcal{H} - E_{\rm c})(t - t')]|\mathbf{R}'\rangle. \tag{10.82}$$

Green's function can be used to sample the time-dependent wavefunction $\Psi(\mathbf{R}, t)$ as discussed by Anderson (1975; 1976). A more efficient way of sampling the wavefunction is by constructing a probability-like function (Reynolds *et al.*, 1982). Assume that we take a trial wavefunction for the ground state, such as the one obtained in a variational quantum Monte Carlo simulation, as the initial wavefunction $\Psi(\mathbf{R}, 0) = \Phi(\mathbf{R})$. We can easily show that the time-dependent wavefunction is given by

$$\Psi(\mathbf{R}, t) = e^{-(\mathcal{H} - E_{\rm c})t}\Phi(\mathbf{R}). \tag{10.83}$$

Then we can construct a probability-like function

$$F(\mathbf{R}, t) = \Psi(\mathbf{R}, t)\Phi(\mathbf{R}). \tag{10.84}$$

We can show that $F(\mathbf{R}, t)$ satisfies the diffusion equation

$$\frac{\partial F}{\partial t} = \frac{1}{2}\nabla^2 F - \boldsymbol{\nabla} \cdot F\mathbf{U} + [E_c - \mathcal{E}(\mathbf{R})]F, \tag{10.85}$$

where

$$\mathbf{U} = \boldsymbol{\nabla} \ln \Phi(\mathbf{R}) \tag{10.86}$$

is the drifting velocity and

$$\mathcal{E}(\mathbf{R}) = \Phi^{-1}(\mathbf{R})\mathcal{H}\Phi(\mathbf{R}) \tag{10.87}$$

is the local energy for a given $\mathbf{R}$. If we introduce a time-dependent expectation value

$$E(t) = \frac{\langle\Phi(\mathbf{R})|\mathcal{H}|\Psi(\mathbf{R}, t)\rangle}{\langle\Phi(\mathbf{R}')|\Psi(\mathbf{R}', t)\rangle} = \frac{\int F(\mathbf{R}, t)\mathcal{E}(\mathbf{R})\, d\mathbf{R}}{\int F(\mathbf{R}', t)\, d\mathbf{R}'} \tag{10.88}$$

at time t, we can obtain the true ground-state energy

$$E_0 = \lim_{t\to\infty} E(t). \tag{10.89}$$

The multidimensional integral in Eq. (10.88) can be obtained by means of the Monte Carlo quadrature with $F(\mathbf{R}, t)$ treated as a time-dependent distribution function if it is positive definite. In practice, the simulation is done by rewriting the diffusion equation for $F(\mathbf{R}, t)$ as an integral

$$F(\mathbf{R}', t + \tau) = \int F(\mathbf{R}, t)G(\mathbf{R}', \mathbf{R}; \tau)\, d\mathbf{R}, \tag{10.90}$$

where $G(\mathbf{R}', \mathbf{R}; \tau)$ is Green's function of the diffusion equation. If τ is very small, Green's function can be approximated as

$$G(\mathbf{R}', \mathbf{R}; \tau) \simeq \mathcal{W}(\mathbf{R}', \mathbf{R}; \tau)G_0(\mathbf{R}', \mathbf{R}; \tau), \tag{10.91}$$

where

$$G_0(\mathbf{R}', \mathbf{R}; \tau) = \left(\frac{1}{2\pi\tau}\right)^{3N/2} e^{-[\mathbf{R}'-\mathbf{R}-\mathbf{U}\tau]^2/2\tau} \tag{10.92}$$

is a propagator due to the drifting and

$$\mathcal{W}(\mathbf{R}', \mathbf{R}; \tau) = e^{-\{[\mathcal{E}(\mathbf{R})+\mathcal{E}(\mathbf{R}')]/2-E_c(t)\}\tau} \tag{10.93}$$

is a branching factor.

In order to treat $F(\mathbf{R}, t)$ as a distribution function or a probability, it has to be positive definite. A common practice is to use a *fixed-node approximation*, which forces the function $F(\mathbf{R}, t)$ to be zero in case it becomes negative. This fixed-node approximation still provides an upper bound for the evaluation of the ground-state energy and gives good molecular energies for small systems if the trial wavefunction is properly chosen. The fixed-node approximation can be removed by releasing the fixed nodes, but this requires additional computing efforts (Ceperley and Alder, 1984). Below is a summary of the major steps in the actual Green's function Monte Carlo simulation.

(1) We first perform variational quantum Monte Carlo simulations to optimize the variational parameters in the trial wavefunction.
(2) Then we can generate an initial ensemble with many independent configurations from the variational simulations.
(3) Each configuration is updated with a drifting term and a Gaussian random walk, with the new coordinate

$$\mathbf{R}' = \mathbf{R} + \mathbf{U}\tau + \chi, \tag{10.94}$$

where χ is a $3N$-dimensional Gaussian random number generated in order to have a variance of τ on each dimension.
(4) Not every step is accepted, and a probability

$$p = \min[1, w(\mathbf{R}', \mathbf{R}; \tau)] \tag{10.95}$$

is used to judge whether the updating move should be accepted or not. Here

$$w(\mathbf{R}', \mathbf{R}, \tau) = \frac{\Phi(\mathbf{R}')^2 G(\mathbf{R}, \mathbf{R}'; \tau)}{\Phi(\mathbf{R})^2 G(\mathbf{R}', \mathbf{R}; \tau)} \tag{10.96}$$

is necessary in order to have detailed balance between points $\mathbf{R}'$ and $\mathbf{R}$. Any move that crosses a node is rejected under the fixed-node approximation.
(5) A new ensemble is then created with branching: that is,

$$M = [\mathcal{W}(\mathbf{R}', \mathbf{R}; \tau_a) + \xi] \tag{10.97}$$

copies of the configuration $\mathbf{R}'$ are put in the new ensemble. Here ξ is a uniform random number between [0, 1] that is used to make sure that the fraction of $\mathcal{W}$ is properly taken care of; τ_a is the effective diffusion time, which is proportional to τ, with the coefficient being the ratio of the mean square distance of the accepted moves to the mean square distance of the attempted moves.
(6) The average local energy at each time step $\bar{\mathcal{E}}(\mathbf{R}')$ is then evaluated from the summation of $\mathcal{E}(\mathbf{R}')$ of each configuration and weighted by the corresponding probability $\mathcal{W}(\mathbf{R}', \mathbf{R}; \tau_a)$. The time-dependent energy $E_c(t)$ is updated at every step with

$$E_c(t) = \frac{\bar{\mathcal{E}}(\mathbf{R}) + \bar{\mathcal{E}}(\mathbf{R}')}{2} \tag{10.98}$$

to ensure a smooth convergence.

Before we can take the data for the calculation, we need to run the above steps enough times that the error is dominated by statistics. The data are typically taken with an interval of ten steps. The exact size of the interval can be determined from the autocorrelation function of the physical quantities evaluated in the simulation. We can then group and average the data to the desired accuracy. Several independent runs can be carried out for a better average.

Another scheme that samples Green's function directly does not have the problems of finite time step or fixed-node approximation. This method was originally proposed by Kalos (1962). It is based on the time-independent Green's function formalism. We can formally write the solution of the many-body Hamiltonian as an integral

$$\Psi^{(n+1)}(\mathbf{R}) = E_c \int G(\mathbf{R}, \mathbf{R}')\Psi^{(n)}(\mathbf{R}')\, d\mathbf{R}', \tag{10.99}$$

where E_c is a trial energy of the system and $G(\mathbf{R}, \mathbf{R}')$ is the Green's function of the Hamiltonian given by

$$\mathcal{H}G(\mathbf{R}, \mathbf{R}') = \delta(\mathbf{R} - \mathbf{R}'), \tag{10.100}$$

where $G(\mathbf{R}, \mathbf{R}')$ also satisfies the boundary condition. We can show that the ground state of the Hamiltonian is recovered for $n = \infty$ with

$$\Psi_0(\mathbf{R}) = \Psi^{(\infty)}(\mathbf{R}). \tag{10.101}$$

In order to have the desired convergence, E_c has to be a positive quantity, which can be ensured at least locally by adding a constant to the Hamiltonian. It can be shown for sufficiently large values of n that

$$\Psi^{(n+1)}(\mathbf{R}) \simeq \frac{E_c}{E_0}\Psi^{(n)}(\mathbf{R}). \tag{10.102}$$

The time-independent Green's function is related to the time-dependent Green's function by

$$G(\mathbf{R}, \mathbf{R}') = \int_0^\infty G(\mathbf{R}, \mathbf{R}'; t)\, dt, \tag{10.103}$$

which can be sampled exactly in principle. In practice, it will usually take a lot of computing time to reach a stable result. For more details on how to sample Green's function, see Ceperley and Kalos (1986) or Lee and Schmidt (1992).

10.7 Two-dimensional electron gas

In the nearly two decades since the bold claim (Anderson, 1987) that cuprate superconductors result from a resonating-valence-bond state, the physics community has agonized with excitement and frustration, searching for such a state and its consequences after doping. The properties of this state would be unique, as exemplified in the proposal of applying it to quantum computing (Ioffe *et al.* 2002). A simpler but related system is the two-dimensional electron gas formed in semiconductor structures, which is found to have many interesting properties, including a metal–insulator transition in the dilute limit (Pudalov *et al.*, 1993). The search for a proper description of the metallic state in the dilute two-dimensional electron gas is intensified because emerging experimental results are challenging the established theoretical pictures.

As an illustrative example, we present numerical evidence from diffusion quantum Monte Carlo simulation to show that the ground state of the dilute two-dimensional electron gas in the vicinity of the Wigner crystallization is best described by a resonating-pair liquid. With an explicit construction of the many-body wavefunction of such a liquid, we elucidate its relations to the resonating-valence-bond and gossamer superconducting states (Laughlin, 2002; Zhang, 2003) and highlight the significance of the disappearance of the Fermi surface in such a system.

The nature of a three-dimensional homogeneous electron gas is well understood in both the high- and low-density limits (Mahan, 2000). When the density is extremely low, the system is a quantum solid, known as the Wigner crystal, resulting from strong Coulomb interactions among the electrons. At extremely high density, the system behaves like a free electron gas because of the screening of the interaction potentials and is well described by the Fermi liquid theory. What happens in between is not entirely understood, and this makes the system interesting, especially when the dimensionality is reduced. Two-dimensional electron gas systems have been full of surprises on several fronts of condensed matter research. First there is the appearance of the quantum and fractional quantum Hall effects under a strong, perpendicular magnetic field. Then there is the discovery of the cuprate superconductors, which can be viewed as a highly correlated two-dimensional electron gas subject to a periodic potential near the Mott insulating state. The third there is the discovery of the metal–insulator transition in a dilute two-dimensional electron gas formed in a semiconductor structure; this is totally unexpected, in defiance of the scaling theory of localization (Abrahams *et al.*, 1979).

Below we outline a study (Pang, 2005) of the two-dimensional electron gas using diffusion quantum Monte Carlo simulation. A metallic state, consisting of a collection of resonating singlet pairs, was found to be the preferred state in the vicinity of the Wigner crystallization. These findings resolve several issues regarding the two-dimensional electron gas and provide important clues on how to proceed to have a better understanding of the electron–electron correlation in related materials. Following convention, we use the jellium model (Mahan, 2000) to describe the two-dimensional homogeneous electron gas:

$$\mathcal{H} = -\frac{1}{2}\sum_{i=1}^{N} \nabla_i^2 + \sum_{i>j=1}^{N} \frac{1}{r_{ij}} + NV_0, \tag{10.104}$$

where N is the total number of particles in the simulation cell, r_{ij} is the distance between the ith electron and the jth electron (or its image if it is closer) under the periodic boundary condition, and $V_0 = -\frac{N}{2}\frac{1}{\Omega}\int_\Omega \frac{d\mathbf{r}}{r}$ is an effective potential due to the charge images and the positive background with Ω being the area of the simulation cell. This model appears to be a reasonable approximation of the corresponding infinite system (Attaccalite *et al.*, 2002). Atomic units are used here for convenience.

The jellium model has been studied extensively, with calculations using quantum Monte Carlo simulations (Attaccalite *et al.*, 2002), which conclude that, in the vicinity of the Wigner crystallization, the system is a spin-polarized Fermi liquid. However, another experiment (Shashkin *et al.*, 2003) seems to suggest that this cannot be the case. Pang (2005) proposes that two-dimensional electron gas in the dilute limit is instead a resonating-pair liquid and introduces a variational wavefunction to describe the ground state of such a liquid. To support this view, variational and diffusion quantum Monte Carlo simulations based on this wavefunction are performed. It is found that the system prefers this liquid state over the Fermi liquid state and this opens a completely new ground for studying highly correlated two-dimensional electron gas systems.

The standard variational and diffusion quantum Monte Carlo simulation methods are applied to the above model with a unique trial/guide many-body wavefunction that contains resonating pairs of spin singlets:

$$|\Psi\rangle = F|\Phi\rangle, \tag{10.105}$$

where $F = \prod_{i\neq j} f(r_{ij})$ is the Jastrow projection factor that restricts the wavefunction according to the interaction when any two particles are approaching closely to each other, and $|\Phi\rangle$ is a determinant formed from matrix elements $\varphi(r_{ij})$ between a spin-up and a spin-down particle (Bouchaud and Lhuillier, 1987). We use $f(r) = e^{u(r)}$, with $u(r) = ar/(1 + br) + s$ for $r < r_{\mathrm{m}}$, $u(r) = \sum_{k=0}^{5} c_k(r - r_{\mathrm{m}})^k$ for $r_{\mathrm{m}} < r < r_{\mathrm{c}}$, and $u(r) = 0$ for $r > r_{\mathrm{c}}$, as has been done for the three-dimensional electron gas (Ortiz, Harris, and Ballone, 1999). In order to have the Coulomb repulsion canceled by the kinetic energy when two particles approach each other, we must have $a = 1$ if the spins are different and $a = 1/3$ if the spins are the same. Other parameters are selected to minimize the variational ground-state energy.

The resonating-pair wavefunction is taken as a modified Gaussian with $\varphi(r) = e^{-\alpha_1 r^2/(1+\beta_1 r^2)} - qe^{-\alpha_2 r^2/(1+\beta_2 r^2)}$, where $\alpha_i > 0$ determine the range of the resonating pair and $\beta_i \geq 0$ determine where the pairing ends with $\beta_i = 0$ being a special case of a bound pair (Bouchaud and Luillier, 1987). The parameter q is used to include the effect of the correlation hole and detailed structure of the pair. This choice of trial/guide wavefunction is based on the intuition that the two-dimensional electron gas near the Wigner crystallization is a liquid but possesses certain characteristics of the nearby insulating solid state. The Wigner crystal is a triangular lattice that has electrons forming resonating spin singlets. If we picture the melting process as one that first delocalizes the electrons while keeping the features of the spin singlets that can be either bound or in a resonating state, near the Wigner crystallization the system must be close to a resonating-valence-bond state or a quantum antiferromagnetic state, with singlet pairs that are close to but not quite bounded.

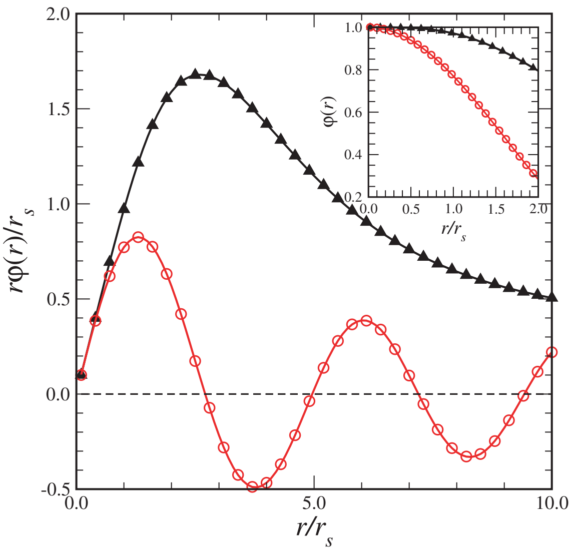


Fig. 10.2 Optimized resonating-pair wavefunction $\varphi(r)$ (triangles) compared with that of the noninteracting system $\varphi_0(r)$ (circles). The size of the resonating pair is clearly on the order of the average nearest neighbor distance ($2r_s$), marked by the peak in $r\varphi(r)$. The long-range oscillatory tail in $\varphi_0(r)$ around zero (indicated by a dashed line) reflects the sharp edge of the Fermi surface in the noninteracting system or the corresponding Fermi-liquid state. The wavefunctions are normalized by their maximum values in the plots and the inset shows the short-range difference between $\varphi(r)$ and $\varphi_0(r)$.

Several variational quantum Monte Carlo simulations on the model Hamiltonian of Eq. (10.104) with the trial wavefunction $|\Psi\rangle$ can be used in order to optimize it. The parameters, $\alpha_1 = 0.11/r_s^2$, $\beta_1 = 0.025/r_s^2$, $\alpha_2 = 0.425/r_s^2$, $\beta_2 = 0.1/r_s^2$, and $q = 0.25$, appear to optimize $|\Psi\rangle$ for any given density $\rho = N/\Omega = 1/(\pi r_s^2)$. Figure 10.2 shows the optimized resonating-pair wavefunction $\varphi(r)$ with that of the Fermi liquid, $\varphi_0(r) = \int_0^{k_F} J_0(kr)k\,dk$, where J_0 is the zeroth-order Bessel function of the first kind and $k_F = \sqrt{2\pi\rho}$ is the Fermi wave number of the noninteracting system.

The ground-state energy of the resonating-pair liquid is obtained through the standard fixed-node diffusion quantum Monte Carlo simulation, shown in Fig. 10.3, referred to the ground-state energy of the corresponding Fermi liquid state, which is calculated in the exactly same manner with the resonating-pair wavefunction $\varphi(r)$ replaced by $\varphi_0(r)$. The scheme provides the exact result if the nodal structure of the exact wavefunction is given. Otherwise the result obtained is variational in nature, representing a thorough search of an entire class of wavefunctions with the same nodal structure.

From the data shown, the resonating-pair liquid is not only plausible but is clearly preferable in the vicinity of the Wigner crystallization. The energy difference calculated in the region from $r_s = 10$ to $r_s = 45$ is great enough to rule out the possibility of a spin-polarized Fermi liquid, and pushes the transition to a Wigner crystal to a higher value than $r_s \simeq 37$. The differences presented should have reduced, if not entirely circumvented, the errors due to the finite size of the simulation cell, truncation of the Coulomb interaction, finite time steps ($\tau = r_s/100$) used, and certain biases created in the specific implementation of

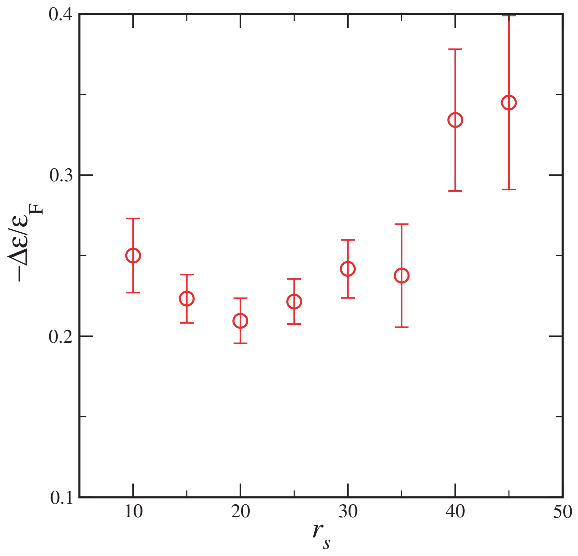


Fig. 10.3 The energy difference per particle $\Delta\varepsilon$ between the resonating-pair liquid and Fermi liquid measured in terms of the Fermi energy ($\varepsilon_F = k_F^2/2$) of the noninteracting system. The total number of particles used in all the simulations here is $N = 58$. The largest difference obtained in the region shown for a spin-polarized liquid or Wigner crystal is on the order of, or smaller than, 0.1, not enough to make one of them a preferable state in the region shown.

the fixed-nodal scheme. Any of these contributions should be much smaller than the energy difference shown and should not have altered the conclusions reached here.

In order to understand the structure of the proposed resonating-pair liquid better, snapshots of the electrons in this state and in the corresponding Fermi liquid state have been taken (Pang, 2005). Figure 10.4 shows two representative snapshots of the electrons with the same $r_s = 30$. As one can see, electrons with different spins are highly correlated in both liquids. They almost all appear in pairs in which one is a spin-up electron and the other a spin-down one. However, the electrons in the resonating-pair liquid are more evenly distributed with a smaller density fluctuation, close to a solid or an incompressible fluid. This is the result of the resonating-pair wavefunction $\varphi(r)$, which has a broad peak at a distance close to the lattice spacing around $2r_s$. For the Fermi liquid, the spin-up and spin-down electrons are still well paired but with a much larger (more liquid-like) density fluctuation. This is evident from the number of pairs that have a separation distance smaller than r_s, reflecting the structures of $\varphi(r)$ and $\varphi_0(r)$. We may speculate that the resonating-pair liquid is close to a glass state, similar to what was argued for the normal state of the ^{3}He liquid (Bouchaud and Luillier, 1987).

The resonating-pair state that we have introduced here is clearly a liquid because of the continuous translational symmetry preserved in the wavefunction. Nevertheless, the resonating-pair liquid does not possess a Fermi surface (or Fermi circle for the two-dimensional systems). This can be shown by comparing the resonating-pair state $\varphi(r)$ with its noninteracting counterpart $\varphi_0(r)$. The noninteracting state $\varphi_0(r)$ is formed from the superposition of plane waves within

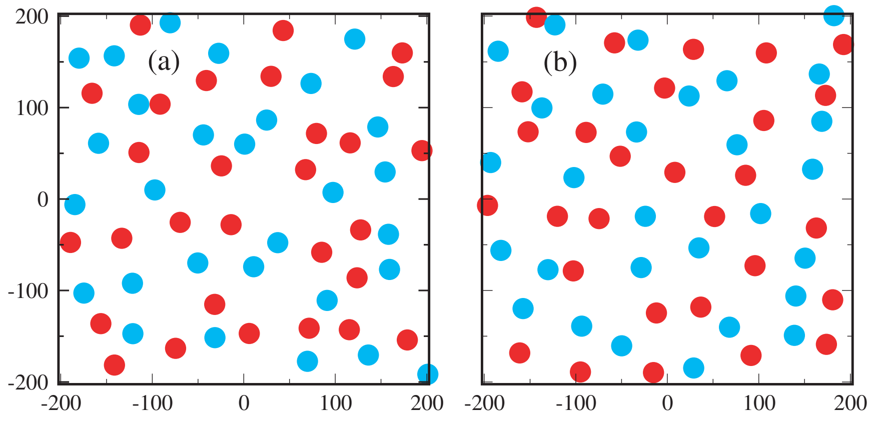


Fig. 10.4 Snapshots of the electrons in the two distinct liquid states with $r_s = 30$: (a) the resonating-pair liquid; (b) the Fermi liquid. Spin-up and spin-down electrons are distinguished in these snapshots. The total number of electrons in each case is $N = 58$ and the average distance between nearest neighbors is about $2r_s = 60$. The snapshots are for the entire simulation cell under the periodic boundary condition. The density fluctuation in the resonating-pair liquid is much smaller, as is evident from only one pair in (a) having a separation distance smaller than r_s, whereas there are nine in (b).

the Fermi surface:

$$\varphi_0(r) = \frac{1}{\Omega}\sum_{k<k_F} e^{i\mathbf{k}\cdot\mathbf{r}}, \tag{10.106}$$

which can be viewed as a Fourier transform with a unit Fourier coefficient for $k < k_F$ and zero otherwise. However, the resonating-pair state has a nonzero Fourier coefficient

$$c_{\mathbf{k}} = \frac{1}{2\pi}\int \varphi(r)e^{-i\mathbf{k}\cdot\mathbf{r}}\, d\mathbf{r} \tag{10.107}$$

for any given $\mathbf{k}$. Here $c_{\mathbf{k}}$ is obviously a continuous function of k across k_F. Therefore the resonating-pair liquid does not have a discontinuity in the single-particle spectrum in the momentum space. This is a significant departure from the conventional Fermi liquid description of the normal state of an interacting Fermi system. Because of the liquid nature of the resonating-pair state, this is certainly a candidate that defies the unique metallic state argument (Laughlin, 1998) based on quantum criticality. An interesting issue that can be investigated in the framework presented here is whether there is a density region in which both the Fermi liquid and the resonating-pair liquid can coexist. This may provide an explanation for the experiment observation (Gao *et al.*, 2002) that the metallic phase of the two-dimensional electron gas appears to be a mixture of two phases.

In general, the Fermi liquid would be sensitive to the disorder and behave as a weakly localized system whereas the resonating-pair liquid, with pairs resembling Cooper pairs within the scattering length, would not be significantly influenced by nonmagnetic impurities or disorders and therefore would remain metallic. The metal–insulator transition can then be viewed as the disappearance or localization of the resonating pairs. The localization of the resonating pairs could be a transformation into a resonating-valence-bond liquid if magnetic ordering is not preferred. This is believed to be the case for a triangular lattice with one electron per lattice site. Otherwise, it can lead to an antiferromagnetic ground state of the localized electrons, such as the case of a square lattice with one electron per lattice site. The third possibility is for the system to become a paired electron solid, a state explored for three-dimensional electron gas (Moulopoulos and Ashcroft, 1993). Another possibility is that a small fraction of resonating pairs become bound, and the system then is a gossamer superconductor. In general, one expects that all the resonating pairs become bound if the interaction is modified to include an attractive tail, such as the case of the ^{3}He liquid (Bouchaud and Luillier, 1987).

10.8 Path-integral Monte Carlo simulations

So far we have been discussing the study of the ground-state properties of quantum many-body systems. In reality, the properties of excited states are as important as those of the ground state, especially when we are interested in the temperature-dependent behavior of the system. In this section, we briefly discuss a finite-temperature quantum Monte Carlo simulation scheme introduced by Pollock and Ceperley (1984). If the system is in equilibrium, the properties of the system are generally given by the density matrix

$$\rho(\mathbf{R}, \mathbf{R}'; \beta) = \langle \mathbf{R}' | e^{-\beta\mathcal{H}} | \mathbf{R} \rangle = \sum_n e^{-\beta E_n} \Psi_n^\dagger(\mathbf{R}') \Psi_n(\mathbf{R}), \tag{10.108}$$

where $\beta = 1/k_B T$ is the inverse temperature and $\mathcal{H}$, E_n, and Ψ_n are related through the Schrödinger equation

$$\mathcal{H}\Psi_n(\mathbf{R}) = E_n \Psi_n(\mathbf{R}) \tag{10.109}$$

of the many-body system. The thermodynamic quantities of the system are related to the density matrix through the partition function

$$\mathcal{Z} = \mathrm{Tr}\rho(\mathbf{R}, \mathbf{R}'; \beta), \tag{10.110}$$

which is the sum of all diagonal elements of the matrix. The density matrix, as well as the partition function, satisfies the convolution

$$\rho(\mathbf{R}, \mathbf{R}'; \beta) = \int \rho(\mathbf{R}, \mathbf{R}_1; \tau) \cdots \rho(\mathbf{R}_M, \mathbf{R}'; \tau)\, d\mathbf{R}_1 \cdots d\mathbf{R}_M, \tag{10.111}$$

with $\tau = \beta/(M+1)$. If M is a very large number, $\rho(\mathbf{R}_i, \mathbf{R}_j; \tau)$ approaches the high-temperature limit with the effective temperature as $T_\tau = 1/k_B\tau \simeq MT$. We can formally show that the quantum density matrix written in the convolution is equivalent to a classical polymer system, which is one dimension higher in geometry than the corresponding quantum system. For example, the point particles of a quantum system therefore behave like classical polymer chains with the bonds and beads connected around each true particle. This can be viewed from the path-integral representation for the partition function

$$\mathcal{Z} = \int e^{-S[\mathbf{R}(t)]} \mathcal{D}\mathbf{R}(t) \tag{10.112}$$

at the limit of $M \to \infty$, where

$$S[\mathbf{R}(t)] = \int_0^\beta \left[\frac{1}{2} \left(\frac{d\mathbf{R}}{dt} \right)^2 + V(\mathbf{R}) \right] dt \tag{10.113}$$

and

$$\mathcal{D}\mathbf{R}(\tau) = \lim_{M\to\infty} \frac{1}{(2\pi\tau)^{3N(M+1)/2}} d\mathbf{R}_1 d\mathbf{R}_2 \cdots d\mathbf{R}_M. \tag{10.114}$$

Note that the path integral in Eq. (10.112) is purely symbolic. For more details on the path-integral representation, especially its relevance to polymers, see Wiegel (1986).

If we can obtain the density matrix of the system at the high-temperature limit, the density matrix and other physical quantities at the given temperature are given by $3N \times M$-dimensional integrals, which can be sampled, in principle, with the Metropolis algorithm. However, in order to have a reasonable speed of convergence, we cannot simply use

$$\rho(\mathbf{R}, \mathbf{R}'; \tau) = \frac{1}{(2\pi\tau)^{3N/2}} \exp\left[-\frac{(\mathbf{R} - \mathbf{R}')^2}{2\tau} - \tau V(\mathbf{R}) \right], \tag{10.115}$$

which is the density matrix at the limit of $M \to \infty$ or $\tau = \beta/(M+1) \to 0$ under the Feynman path-integral representation for simulation purposes. A good choice seems to be the pair-product approximation (Barker, 1979)

$$\rho(\mathbf{R}, \mathbf{R}'; \tau) \simeq \rho_0(\mathbf{R}, \mathbf{R}'; \tau) e^{-U(\mathbf{R}, \mathbf{R}'; \tau)}, \tag{10.116}$$

with

$$U(\mathbf{R}, \mathbf{R}'; \tau) = \sum_{i<j} u(r_{ij}, r'_{ij}; \tau), \tag{10.117}$$

which is the pair correction to the free particle density matrix,

$$\rho_0(\mathbf{R}, \mathbf{R}'; \tau) = \frac{1}{(2\pi\tau)^{3N/2}} \exp\left[-\frac{(\mathbf{R} - \mathbf{R}')^2}{2\tau} \right]. \tag{10.118}$$

Another important issue is that each quantum many-body system satisfies a certain quantum statistics. Fermions and bosons behave totally differently when two particles in the system are interchanged. The general expression of the symmetrized or antisymmetrized density matrix can be written as

$$\rho_\pm(\mathbf{R}, \mathbf{R}'; \beta) = \frac{1}{N!} \sum_P (\pm 1)^{\mathrm{P}} \rho(\mathbf{R}, \mathrm{P}\mathbf{R}'; \beta), \tag{10.119}$$

where P is an indicator of permutations. This matrix with permutations has been used to study liquid ^{4}He (Ceperley and Pollock, 1986), liquid ^{3}He (Ceperley, 1992), and liquid helium mixtures (Boninsegni and Ceperley, 1995). For a review of the subject see Ceperley (1995). The path-integral Monte Carlo method has been used to study the interacting bosons trapped in a potential well (Pearson, Pang, and Chen, 1998). An unexpected symmetry-breaking phenomenon shows up in a trapped boson mixture (Ma and Pang, 2004).

Different Metropolis sampling schemes have also been developed for path-integral Monte Carlo simulations. Interested readers should consult Schmidt and Ceperley (1992). We can also find the discussion of a new scheme devised specifically for many-fermion systems in Lyubartsev and Vorontsov-Velyaminov (1993).

10.9 Quantum lattice models

Various quantum Monte Carlo simulation techniques discussed in the last few sections have also been applied in the study of quantum lattice models, such as the quantum Heisenberg, Hubbard, and t–J models. These studies are playing a very important role in the understanding of highly correlated materials systems that have great potential for future technology.

In this section, we will give a very brief introduction to the subject: interested readers can refer the vast literature on the subject. We will use the Hubbard model as an illustrative example to highlight the key elements in these approaches and also point out the difficulties encountered in current research. We have used the Hubbard model in describing the electronic structure of H_3^+ in Chapter 5. The single-band Hubbard model for a general electronic system is given by the Hamiltonian

$$\mathcal{H} = -t \sum_{\langle ij \rangle}^{L} (a_{i\uparrow}^{\dagger} a_{j\uparrow} + a_{i\downarrow}^{\dagger} a_{j\downarrow}) + U \sum_{i=1}^{L} n_{i\uparrow} n_{i\downarrow}, \tag{10.120}$$

where $a_{i\sigma}^{\dagger}$ and $a_{j\sigma}$ are creation and annihilation operators of an electron with either spin up ($\sigma = \uparrow$) or down ($\sigma = \downarrow$), $n_{i\sigma} = a_{i\sigma}^{\dagger} a_{i\sigma}$ is the corresponding occupancy at the ith site, t is the hopping integral of an electron between the nearest sites (notated as$\langle ij \rangle$), U is the on-site interaction of two electrons with opposite spin orientations, and L is the total number of lattice sites.

The Hubbard model is extremely appealing, because it is very simple but contains almost all the information in the highly correlated systems. An exact solution for this model is only available for one dimension. The typical quantities to be studied are the spin correlation function, the particle correlation function, and the excitation spectra. The spin–spin correlation function is defined as

$$C_s(l) = \langle \mathbf{s}_i \cdot \mathbf{s}_{i+l} \rangle, \tag{10.121}$$

where the spin operator is given by

$$\mathbf{s}_i = \frac{\hbar}{2} \sum_{\sigma\nu} a_{i\sigma}^{\dagger} \mathbf{p}_{\sigma\nu} a_{i\nu}, \tag{10.122}$$

with $\mathbf{p}_{\sigma\nu}$ being the Pauli matrices,

$$p^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; p^y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \; p^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{10.123}$$

The particle–particle correlation function is defined as

$$C_{\mathrm{p}}(l) = \langle \Delta n_i \Delta n_{i+l} \rangle, \tag{10.124}$$

where $\Delta n_i = n_i - n_0$ is the difference in the occupancy of the ith site and the average occupancy n_0. Note that the index l is used symbolically for pairs of sites that are separated by the same distance. These and other correlation functions are very useful in understanding the physical properties of the system. For example, a solid state is formed if the particle correlation function has a long-range order and the (staggered) magnetization m is given by the long-distance limit of the spin correlation function as $m^2 = \frac{1}{3}C_{\mathrm{s}}(l \to \infty)$. These correlation functions can be evaluated up to a point by quantum Monte Carlo simulations.

Variational simulation

The variational quantum Monte Carlo simulation is a combination of the variational scheme and the Metropolis algorithm. We first need construct a trial wavefunction that contains as much of the relevant physics as possible. It is usually very difficult actually to come up with a state that is concise but contains all the important physics. It appears only to have happened three times in the entire history of the many-body theory: in 1930, when Bethe proposed the Bethe ansatz for the spin-1/2 one-dimensional Heisenberg model; in 1957, when Bardeen, Cooper, and Schrieffer proposed the BCS ansatz for the ground state of superconductivity; and in 1983, when Laughlin proposed the Laughlin ansatz for the fractional quantum Hall state. It turns out that the Bethe ansatz is exact for the spin-1/2 chain and has since been applied to many other one-dimensional systems.

Because there is no such extraordinary trial state for the two- or three-dimensional Hubbard model, we will take the best-known trial state so far, the Gutzwiller ansatz

$$|\Phi\rangle = \prod_{i=1}^{L} [1 - (1-g) n_{i\uparrow} n_{i\downarrow}] |\Phi_0\rangle, \tag{10.125}$$

to illustrate the scheme. Here g is the only variational parameter in the trial wavefunction and $|\Phi_0\rangle$ is a reference state, which is usually chosen as the uncorrelated state of the Hamiltonian, that is, the solution with $U = 0$, the single-band tight-binding model. This trial state is used by Vollhardt (1984) in the description of the behavior of the normal state of liquid ^{3}He. Some other variational wavefunctions have also been developed for the Hubbard model, but so far there has not been a unique description because of the lack of a fundamental understanding of the model in two dimensions. We can show that the Gutzwiller ansatz can be rewritten as

$$|\Phi\rangle = g^D|\Phi_0\rangle, \tag{10.126}$$

where D is the number of sites with double occupancy. The above variational wavefunction can be generalized into a Jastrow type of ansatz

$$|\Phi\rangle = e^{-U[n_i]/2}|\Phi_0\rangle, \tag{10.127}$$

with the correlation factor given by

$$U[n_i] = \sum_{j,k}^{L} \alpha_l(j,k) n_j n_k, \tag{10.128}$$

where $\alpha_l(j,k)$ is a variational parameter between the jth and kth sites. The spin index is combined with the site index for convenience, and the summation over j and k should be interpreted as also including spin degrees of freedom. The index $l = 1, 2$ is for the different spin configuration of the ith and jth sites with two distinct situations: spin parallel and spin antiparallel. The above Jastrow type of ansatz reduces back to the Gutzwiller ansatz when $\alpha_l(j,k)$ is only for the on-site occupancy with $\alpha = -\ln(1/g)$.

The simulation is performed with optimization of the energy expectation

$$E = \frac{\langle\Phi|\mathcal{H}|\Phi\rangle}{\langle\Phi|\Phi\rangle}. \tag{10.129}$$

Several aspects need special care during the simulation. First, the total spin of the system is commutative with the Hamiltonian, so we cannot change the total spin of the system, that is, we can only move the particles around but not flip their spins, or flip a spin-up and spin-down pair at the same time. The optimized g is a function of t/U and n_0, so we can choose t as the unit of energy, that is, $t = 1$, and then g will be a function of U and n_0 only. Each site cannot be occupied by more than two particles with opposite spins. So we can move one particle to its nearest site if allowed, and the direction of the move can be determined by a random number generator. For example, if we label the nearest sites $1, 2, \ldots, z$, we can move the particle to the site labeled int$(\eta_i z + 1)$ if allowed by the Pauli principle. Here η_i is a uniform random number in the region [0, 1]. The move is accepted by comparing

$$p = \frac{|\Phi_{\text{new}}|^2}{|\Phi_{\text{old}}|^2} = \frac{\exp\left(-U\left[n_i^{\text{new}}\right]\right)}{\exp\left(-U\left[n_i^{\text{old}}\right]\right)} \tag{10.130}$$

with another uniform random number w_i. Other aspects are similar to the variational quantum Monte Carlo simulation discussed earlier in this chapter for continuous systems.

Green's function Monte Carlo simulation

We can also use the Green's function Monte Carlo simulation to study quantum lattice systems. As we discussed earlier, the solution of the imaginary-time Schrödinger equation can be expressed in terms of an integral equation

$$\Psi(\mathbf{R}, \mathbf{R}', t + \tau) = \int G(\mathbf{R}, \mathbf{R}', \tau)\Psi(\mathbf{R}', t)\, d\mathbf{R}', \tag{10.131}$$

where $\mathbf{R}$ or $\mathbf{R}'$ is a set of locations and spin orientations of all the particles.

So far the Green's function Monte Carlo simulation has been performed for only lattice spin problems, especially the spin-$\frac{1}{2}$ antiferromagnetic Heisenberg model on a square lattice, which is relevant to the normal state of cuprates, which become superconducting with high transition temperatures after doping. We can show formally that this model is the limit of the Hubbard model with half-filling and infinite on-site interaction. The Heisenberg model is given by

$$\mathcal{H} = J \sum_{\langle ij \rangle}^{L} \mathbf{s}_i \cdot \mathbf{s}_j, \tag{10.132}$$

with $J = 4t^2/U$ a positive constant for the antiferromagnetic case. This model can formally be transformed into a hard-core lattice boson model with

$$\mathcal{H} = -J \sum_{\langle ij \rangle}^{L} b_i^{\dagger} b_j + J \sum_{\langle ij \rangle}^{L} n_i n_j + E_0, \tag{10.133}$$

where $b_i^{\dagger} = s_i^{+} = s_i^{x} + i s_i^{y}$, $b_i = s_i^{-} = s_i^{x} - i s_i^{y}$, and $n_i = b_i^{\dagger} b_i = 1/2 - s_i^{z}$. The constant $E_0 = -Jz(L - N)/4$, where L is the total number of sites in the system, $N = L/2 - S^z$ is the number of occupied sites, and z is the number of nearest neighbors of each lattice point. S^z is the total spin along the z direction. As we discussed earlier in this chapter, if we choose the time step τ to be very small, we can use the short-time approximation for Green's function

$$G(\mathbf{R}, \mathbf{R}', \tau) = \langle \mathbf{R} | e^{-\tau(\mathcal{H} - E_{\mathrm{c}})} | \mathbf{R}' \rangle \simeq \langle \mathbf{R} | [1 - \tau(\mathcal{H} - E_{\mathrm{c}})] | \mathbf{R}' \rangle. \tag{10.134}$$

This has a very simple form

$$G(\mathbf{R}, \mathbf{R}', \tau) = \begin{cases} 1 - \tau[V(\mathbf{R}) - E_{\mathrm{c}}] & \text{for } \mathbf{R} = \mathbf{R}', \\ \tau J/2 & \text{for } \mathbf{R} = \mathbf{R}' + \Delta\mathbf{R}', \\ 0 & \text{otherwise}, \end{cases} \tag{10.135}$$

under the boson representation, where $\mathbf{R}' + \Delta\mathbf{R}'$ is a position vector with one particle moved to the nearest neighbor site in $\mathbf{R}'$ and $V(\mathbf{R})$ is the interaction part, that is, the second term in the Hamiltonian with a configuration $\mathbf{R}$. A procedure similar to that we discussed for continuous systems can be devised. Readers who are interested in the details of the Green's function Monte Carlo simulation for quantum lattice systems should consult the original articles, for example, Trivedi and Ceperley (1990).

Finite-temperature simulation

If we want to know the temperature dependence of the system, we need to simulate it at a finite temperature. The starting point is the partition function

$$\mathcal{Z} = \mathrm{Tr}e^{-\beta\mathcal{H}}, \tag{10.136}$$

where $\mathcal{H}$ is the lattice Hamiltonian, for example, the Hubbard model, and $\beta = 1/k_B T$ is the inverse temperature. The average of a physical quantity A is measured by

$$\langle A \rangle = \frac{1}{\mathcal{Z}}\mathrm{Tr}Ae^{-\beta\mathcal{H}}. \tag{10.137}$$

If we divide the inverse temperature into $M+1$ slices with $\tau = \beta/(M+1)$, we can rewrite the partition function as a product

$$\mathcal{Z} = \mathrm{Tr}(e^{-\tau\mathcal{H}})^{M+1}. \tag{10.138}$$

In the limit of $\tau \to 0$, we can show that

$$e^{-\tau\mathcal{H}} \simeq e^{-\tau V}e^{-\tau\mathcal{H}_0}, \tag{10.139}$$

where

$$V = U\sum_{i=1}^{L} n_{i\uparrow}n_{i\downarrow} \tag{10.140}$$

is the potential part of the Hamiltonian and

$$\mathcal{H}_0 = -t\sum_{\langle ij\rangle}^{L}(a_{i\uparrow}^{\dagger}a_{j\uparrow} + a_{i\downarrow}^{\dagger}a_{i\downarrow}) \tag{10.141}$$

is the contribution of all the hopping integrals. Hirsch (1985a; 1985b) has shown that the discrete Hubbard–Stratonovich transformation can be used to rewrite the quartic term of the interaction as a quadratic form, that is,

$$e^{-\tau U(n_{i\uparrow}-1/2)(n_{i\downarrow}-1/2)} = e^{-\tau U/4}\sum_{\sigma_l=\pm 1} e^{-\tau\sigma_l\lambda(n_{i\uparrow}-n_{i\downarrow})}, \tag{10.142}$$

with λ given by

$$\cosh(\tau\lambda) = e^{\tau U/2}. \tag{10.143}$$

Because the kinetic energy part $\mathcal{H}_0$ has only the quadratic term, it is ready now to have the partition function simulated with the Metropolis algorithm. Readers who are interested in more details can find a good discussion of the method in Sugar (1990).

There is a fundamental problem for the Fermi systems, called the *fermion sign problem*, which appears in most Fermi systems. It is the result of the fundamental properties of the nonlocal exchange interaction or the Pauli principle. It may appear in different forms, such as the nodal structure in the wavefunction or negative probability in the stochastic simulation. However, the challenge is to find an approximation that preserves the essential physics of a given many-body fermion system. Interested readers will be able to find many discussions, but not yet a real solution, in the literature.

Exercises

10.1 Show that the Monte Carlo quadrature yields a standard deviation

$$\Delta^2 = \langle A^2 \rangle - \langle A \rangle^2 \propto \frac{1}{M},$$

where A is a physical observable and M is the total number of Monte Carlo points taken. Demonstrate it numerically by sampling the average of a set data $x_i \in [0, 1]$, drawn uniformly from a random-number generator.

10.2 Show that the Metropolis algorithm applied to statistical mechanics does satisfy the detailed balance and sample the points according to the distribution function. Demonstrate it by sampling the speed of a particle in an ideal gas.

10.3 Calculate the integral

$$S = \int_{-\infty}^{\infty} e^{-r^2/2}(xyz)^2 \, d\mathbf{r}$$

with the Metropolis algorithm and compare the Monte Carlo result with the exact result. Does the Monte Carlo errorbar decrease with the total number of points as expected?

10.4 Calculate the autocorrelation function of the data sampled in the evaluation of the integral in Exercise 10.3. From the plot of the autocorrelation function, determine how many points need to be skipped between any two data points in order to have nearly uncorrelated data points.

10.5 Develop a Monte Carlo program for the ferromagnetic Ising model on a square lattice. For simplicity, the system can be chosen as an $N \times N$ square with the periodic boundary condition imposed. Update the spin configuration site by site. Study the temperature dependence of the magnetization. Is there any critical slowing down in the simulation when the system is approaching the critical point?

10.6 Implement the Swendsen–Wang algorithm in a Monte Carlo study of the ferromagnetic Ising model on a triangular lattice. Does the Swendsen–Wang scheme cure the critical slowing down completely?

10.7 Implement the Wolff algorithm in a Monte Carlo study of the ferromagnetic Heisenberg model on a cubic lattice. Does the Wolff scheme cure the critical slowing down completely?

10.8 Study the antiferromagnetic Ising model on a square lattice using both the Swendsen–Wang and Wolff algorithms. Find the temperature dependence of the staggered magnetization. Which of the algorithms handles the critical slowing down better? What happens if the system is a triangular lattice?

10.9 Carry out a Monte Carlo study of the anisotropic Heisenberg model

$$\mathcal{H} = -J \sum_{\langle ij \rangle}^{L} \left(\lambda s_i^z s_j^z + s_i^x s_j^x + s_i^y s_j^y \right)$$

on a square lattice, where $J > 0$ and the spins are classical, each with a magnitude S. Find the λ dependence of the critical temperature for a chosen value of S. What happens if $J < 0$?

10.10 Study the electronic structure of the helium atom using the variational quantum Monte Carlo method. Assume that the nucleus is fixed at the origin of the coordinates. The key is to find a good parameterized variational wavefunction with proper cusp conditions built in.

10.11 Find the variational ground-state energy per particle, the density profile, and the pair correlation function of a ^{4}He cluster. How sensitive are the values to the size of the cluster? Assume that the interaction between any two atoms is given by the Lennard-Jones potential and express the results in terms of the potential parameters. What happens if the system is a ^{3}He cluster?

10.12 Implement the Green's function Monte Carlo algorithm in a study of a hydrogen molecule. The two-proton and two-electron system should be treated as a four-body system. Calculate the ground-state energy of the system and compare the result with the best of the known calculations.

10.13 The structure of the liquid ^{4}He can be studied using the variational quantum Monte Carlo method. Develop a program to calculate the ground-state energy per particle and the pair correlation function with the selected variational wavefunction. Assume that the system is in a cubic simulation cell under the periodic boundary condition and use the density measured at the lowest temperature possible and the state-of-the-art calculation of the interaction potential for the simulation.

10.14 Perform the diffusion or Green's function Monte Carlo simulation for liquid ^{4}He. Is there any significant improvement in the calculated ground-state energy over that of the variational Monte Carlo calculation?

10.15 Carry out path-integral quantum Monte Carlo simulation of a hard-sphere boson cluster in an anisotropic harmonic trap with a trapping potential

$$V(\mathbf{r}) = \frac{m\omega^2}{2}\left(\lambda z^2 + x^2 + y^2\right),$$

where m is the mass of a particle and ω and λ are parameters of the trap. Find the condensation temperature of the system for a set of different values of the total number of particles, hard-sphere radius, ω, and λ. What happens if there is more than one species of bosons in the trap?

Chapter 11
Genetic algorithm and programming

From the relevant discussions on function optimization covered in Chapters 3, 5, and 10, we by now should have realized that to find the global minimum or maximum of a multivariable function is in general a formidable task even though a search for an extreme of the same function under certain circumstances is achievable. This is the driving force behind the never-ending quest for newer and better schemes in the hope of finding a method that will ultimately lead to the discovery of the shortest path for a system to reach its overall optimal configuration.

The *genetic algorithm* is one of the schemes obtained from these vast efforts. The method mimics the evolution process in biology with inheritance and mutation from the parents built into the new generation as the key elements. Fitness is used as a test for maintaining a particular genetic makeup of a chromosome. The scheme was pioneered by Holland (1975) and enhanced and publicized by Goldberg (1989). Since then the scheme has been applied to many problems that involve different types of optimization processes (Bäck, Fogel, and Michalewicz, 2003). Because of its strength and potential applications in many optimization problems, we introduce the scheme and highlight some of its basic elements with a concrete example in this chapter. Several variations of the genetic algorithm have emerged in the last decade under the collective name of evolutionary algorithms and the scope of the applications has also been expanded into multi-objective optimization (Deb, 2001; Coello Coello, van Veldhuizen, and Lamont, 2002). The main purpose here is to introduce the practical aspects of the method. Readers interested in its mathematical foundations can find the relevant material in Mitchell (1996) and Vose (1999).

We can even take one further step. Instead of encoding the possible configurations into chromosomes, possible computing operations can be selected or altered, resulting in the program encoding itself. This direct manipulation or creation of optimal programs based on the evolution concept is called *genetic programming*, which was conceptualized 40 years ago (Fogel, 1962) and matured more recently (Koza, 1992). We will not be able cover all the details of genetic programming here but will merely highlight the concept toward to the end of the chapter.

11.1 Basic elements of a genetic algorithm

The basic idea behind a genetic algorithm is to follow the biological process of evolution in selecting the path to reach an optimal configuration of a given complex system. For example, for an interacting many-body system, the equilibrium is reached by moving the system to the configuration that is at the global minimum on its potential energy surface. This is single-objective optimization, which can be described mathematically as searching for the global minimum of a multivariable function $g(r_1, r_2, \ldots, r_n)$. Multiobjective optimization involves more than one equation, for example, a search for the minima of $g_k(r_1, r_2, \ldots, r_n)$ with $k = 1, 2, \ldots, l$. Both types of optimization can involve some constraints. We limit ourselves to single-objective optimization here. For a detailed discussion on multi-objective optimization using the genetic algorithm, see Deb (2001).

In this section, we describe only a binary version of the genetic algorithm that closely follows the evolutionary processes. The advantage of the binary genetic algorithm lies in its simplicity, and it articulates the evolution in the forms of binary chromosomes. Later in the chapter, we will also introduce the algorithm that uses real numbers in constructing a chromosome. The advantage of the real genetic algorithm is that the search is done with continuous variables, which better reflects the nature of a typical optimization problem.

In the binary algorithm, each configuration of variables $(r_1, r_2, \ldots, r_n)$ is represented by a binary array, This array can be stored on a computer as an integer array with each element containing a decimal number 1 or 0, or as a boolean array with each element containing a bit that is set to be true (1) or false (0). We will use boolean numbers in actual computer codes but use decimal 0s and 1s when writing equations for convenience.

Several steps are involved in a genetic algorithm. First we need to create an initial population of configurations, which is called the initial gene pool. Then we need to select some members to be the parents for reproduction. The way to mix the genes of the two parents is called crossover, which reflects how the genetic attributes are passed on. In order to produce true offspring, each of the parent chromosomes is cut into segments that are exchanged and joined together to form the new chromosomes of the offspring. After that we allow a certain percentage of bits in the chromosomes to mutate. In the whole process, we use the fitness of each configuration based on the cost (the function to be optimized) $g(r_1, r_2, \ldots, r_n)$ as the criterion for selecting parents and sorting the chromosomes for the next generation of the gene pool. In each of the three main operations (selection, crossover, and mutation) in each generation, we make sure that the elite configurations with the lowest costs always survive.

Creating a gene pool

The initial population of the gene pool is typically created randomly. A sorting scheme is used to rank each of the chromosomes according to its fitness. The following method shows an example of how to create the initial population of the gene pool.

```
// Method to initialize the simulation by creating the
// zeroth generation of the gene population.

  public static void initiate(){
    Random rnd = new Random();
    boolean d[][] = new boolean[ni][nd];
    boolean w[] = new boolean[nd];
    double r[] = new double[nv];
    double e[] = new double[ni];
    int index[] = new int[ni];

    for (int i=0; i<ni; ++i) {
      for (int j=0; j<nv; ++j) r[j] = rnd.nextDouble();
      e[i] = cost(r);
      index[i] = i;
      w = encode(r,nb);
      for (int j=0; j<nd; ++j) d[i][j] = w[j];
    }
    sort(e,index);
    for (int i=0; i<ng; ++i){
      f[i] = e[i];
      for (int j=0; j<nd; ++j) c[i][j] = d[index[i]][j];
    }
  }
```

There are several issues that need to be addressed during the initialization of the gene pool. The first is the size of the population. Even though each individual configuration in the gene pool is represented by a distinct chromosome, a good choice of the population size optimizes the convergence of the simulation. If the population is too small, it will require more time to sample the entire possible configuration space; but if the population is too large, it takes more time to create a new generation each time. The second issue concerns the quality of the initial gene pool. If we just used all the configurations created randomly, the quality of the initial gene pool would be low and that means a longer convergence time. Instead, as shown in the above example, we usually create more chromosomes initially in order to give us a choice. For example, if we want to have a population of n_g chromosomes in the gene pool, we can randomly generated a larger number of chromosomes and then select the best n_g chromosomes that have the lowest costs. A typical choice is to have $n_i = 2n_g$ chromosomes from which to choose. We have to encode each configuration into a chromosome and also evaluate its corresponding cost. These two issues will be discussed later in the section. After we obtain the costs of all the configurations, we can rank them accordingly through a sorting scheme. Because we want the ranking recorded and used to relabel the chromosomes, we need to assign a ranking index in the sorting process. The following method shows how to achieve such a sorting.

```
// Method to sort an array x[i] from the lowest to the
// highest with the original order stored in index[i].

  public static void sort(double x[], int index[]){
    int m = x.length;
    for (int i = 0; i<m; ++i) {
      for (int j = i+1; j<m; ++j) {
        if (x[i] > x[j]) {
          double xtmp = x[i];
          x[i] = x[j];
          x[j] = xtmp;
          int itmp = index[i];
          index[i] = index[j];
          index[j] = itmp;
        }
      }
    }
  }
```

Note that the index used here in the input is in a natural order and that in the output is in increasing order of the associated costs of all the configurations. This sorting scheme is also used in rearranging the chromosomes in other places when needed.

Encoding a configuration

Because we want to model the genetic process accurately, we need to convert each configuration $(r_1, r_2, \ldots, r_n)$ into a chromosome through an encoding process. We will assume that the variables r_i are in the region [0, 1], that is, $0 \leq r_i \leq 1$ for $i = 1, 2, \cdots, n$. This does not affect the generality of the discussion as long as the variables are bound in a finite region so they can always be cast back into the region [0, 1] by a linear transformation.

We can represent any variable $r_i \in [0, 1]$ by a binary string in which each bit is set to a true or false value. If the kth bit is true, there is a fraction $1/2^k$ contributed to the variable, which can then be expressed as

$$r_i = \frac{y_{i1}}{2} + \frac{y_{i2}}{4} + \frac{y_{i3}}{8} + \cdots = \sum_{j=1}^{\infty} \frac{y_{ij}}{2^j}, \tag{11.1}$$

where y_{ij} is a decimal integer equal to either 0 or 1. We can truncate the binary string at a selected number m, whose value depends on how accurate we want r_i to be. Then we have

$$r_i \simeq \sum_{j=1}^{m} \frac{y_{ij}}{2^j}. \tag{11.2}$$

For example, if $r_i = 0.93$, We have $y_{i1} = y_{i2} = y_{i3} = 1$ and $y_{i4} = 0$, and if $r_i = 0.6347$, we have $y_{i1} = y_{i3} = 1$ and $y_{i2} = y_{i4} = 0$, for $m = 4$. The maximum error created in r_i is $\pm 1/2^{m+1}$. The process of generating y_{ij} is called *encoding*, which is accomplished with

$$y_{ij} = \text{int}\left[2^{j-1} r_i - \sum_{k=1}^{j-1} \left(2^{j-k-1} y_{ik}\right)\right], \tag{11.3}$$

for $j = 2, 3, \ldots, m$, where the operation int rounds the value inside the square bracket to the nearest decimal integer (either 0 or 1), with $i = 1, 2, \ldots, n$. Note that $y_{i1} = \text{int}[r_i]$.

The encoding is a process of representing r_i by a binary array with each element containing a bit that is set to be true or false, corresponding to a decimal integer 1 or 0. We call this binary array y_{ij} for $j = 1, 2, \ldots, m$ the ith *gene* of the *chromosome* that is a binary representation of the entire real array $(r_1, r_2, \ldots, r_n)$. The following method encodes the array r_i for $i = 1, 2, \ldots, n$ into a binary chromosome $w = \{y_{11} \ldots y_{1m}\, y_{21} \ldots y_{2m}\, y_{n1} \ldots y_{nm}\}$.

```
// Method to encode an array of n real numbers r[i] in
// [0,1] into an n*m binary representation w[j].

  public static boolean[] encode(double r[], int m) {
    int n = r.length;
    boolean w[] = new boolean[n*m];
    for (int i = 0; i<n; ++i) {
      double sum = r[i];
      w[i*m] = false;
      if((int)(0.5+sum) == 1) w[i*m] = true;
      double d = 2;
      for (int j = 1; j<m; ++j) {
        if(w[i*m+j-1]) sum -= 1/d;
        w[i*m+j] = false;
        if((int)(0.5+d*sum) == 1) w[i*m+j] = true;
        d *= 2;
      }
    }
    return w;
  }
```

Of course, the reverse of the encoding process is also necessary when we need to use the configuration information in the evaluation of the cost function or output the final configurations. To decode a chromosome, we can use Eq. (11.2). For example, if we have a chromosome $w = \{1010111001\ldots\}$ for $m = 10$, the corresponding variable

$$r_1 \simeq \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^{10}} \simeq 0.6807. \tag{11.4}$$

Note that the uncertainty in r_1 here is determined by the choice of m, given by $\pm 1/2^{m+1} \simeq \pm 0.0005$ in this case. This decoding process can be achieved quite easily in a program. The following is an example for decoding a binary array.

```
// Method to decode an array of n*m binary numbers w[j]
// into an array of n real numbers r[i].

  public static double[] decode(boolean w[], int m) {
    int n = w.length/m;
    double r[] = new double[n];
    for (int i = 0; i<n; ++i) {
      double d = 2;
      double sum = 0;
      for (int j = 0; j<m; ++j) {
        if(w[i*m+j]) sum += 1/d;
        d *= 2;
      }
      r[i] = sum + 1/d;
    }
    return r;
  }
```

Now we know how to create a chromosome w for a real array $(r_1, r_2, \ldots, r_n)$. We can then use the encoding scheme to create an initial population that is sorted with the cost function. After we have all these tools, we can accomplish the three main operations, selection, crossover, and mutation, in a genetic algorithm.

Selection operation

We need to able to select a fraction of the chromosomes from the given gene pool to pass on their genes. This models the natural reproduction process. If we take Darwin's concept literally, the chromosomes with better costs are the ones most likely to survive. Several methods have been designed to follow this concept in selecting the parents. A simple choice is to use the best half of the chromosomes from the entire population according to their costs. Then we can randomly select one pair after another from this parent pool to create a certain number of offspring. Another choice is to select a chromosome from the entire population with a probability based on a weight assigned according to either its ranking in cost or its relative cost. For a comparative study of these choices, see Goldberg and Deb (1991).

The most popular method for creating the parents is to hold tournaments, in each of which the two or more participants are selected at random with the winner of each tournament being the participant with the best cost. The following method illustrates how to select the winners from one-on-one matches.

```
// Method to run tournaments for selecting the parents.

  public static void select() {
    int index[] = new int[ng];
    boolean d[][] = new boolean[ng][nd];
    double e[] = new double[ng];

    for (int i=0; i<ng; ++i){
      for (int l=0; l<nd; ++l) d[i][l] = c[i][l];
      e[i] = f[i];
      index[i] = i;
    }
    shuffle(index);
    int k = 0;
    for (int i=0; i<nr; ++i) {
      if (e[index[k]] < e[index[k+1]]){
        for (int l=0; l<nd; ++l) c[i][l]=d[index[k]][l];
        f[i] = e[index[k]];
      }
      else {
        for (int l=0; l<nd; ++l) c[i][l]=d[index[k+1]][l];
        f[i] = e[index[k+1]];
      }
      k += 2;
    }
  }
```

Note that we have allowed each member in the pool to participate, but in only one match. Half of the chromosomes in the pool ($n_\text{r} = n_\text{g}/2$) are selected to be the parents without any duplication. The above method will always result in a copy of the best chromosome being and retained the worst being eliminated. Some other tournament schemes do allow duplications, for example, to have two copies of the best chromosome in the parent pool by letting each chromosome participate in two matches. Shuffling is used to mix up the indices before the matches so that the participants are randomly drawn for a match with an equal probability. Here is how we shuffle the indices.

```
// Method to shuffle the index array.

  public static void shuffle(int index[]){
    int k = index.length;
    Random rnd = new Random();
    for (int i = 0; i<k; ++i) {
      int j = (int)(k*rnd.nextDouble());
      if (j!=i) {
        int itmp = index[i];
        index[i] = index[j];
        index[j] = itmp;
      }
    }
  }
```

After we have completed the selection, we are ready to make some new chromosomes.

Crossover operation

So far we have found ways of creating a gene pool based on randomly assigned bits and then selecting certain genes to be the parents. The next step is to devise a way of exploring the cost surface. There are two operations in the genetic algorithm that effectively look over the entire variable space. The first operation is called *crossover*, which is achieved by mimicking the reproduction processes in Nature. We can choose a pair of parents from the parent pool, cut each chromosome of the parents into two segments at a selected point, and then join the right segment of one parent to the left segment of the other, and vice versa, to form two new chromosomes for the offspring. For example, if the chromosomes of two parents are $w_1 = \{01101010\}$ and $w_2 = \{10101101\}$, the corresponding chromosomes of the offspring are $w_3 = \{01101101\}$ and $w_4 = \{10101010\}$, respectively, with the crossover point taken as the middle of the chromosomes. Crossover is one the most effective ways of exploring the cost surface of all the possible configurations. What we have described here is the single-point crossover scheme. There are other types of crossover schemes and the readers can find them in the related literature (Spears, 1998). The following method is an implementation of the single-point crossover.

```
// Method to perform single-point crossover operations.

  public static void cross() {
    Random rnd = new Random();

    int k = 0;
    for (int i=nr; i<nr+nr/2; ++i) {
      int nx = 1 + (int)(nd*rnd.nextDouble());
      for (int l=0; l<nx; ++l){
        c[i][l] = c[k][l];
        c[i+nr/2][l] = c[k+1][l];
      }
      for (int l=nx; l<nd; ++l){
        c[i][l] = c[k+1][l];
        c[i+nr/2][l] = c[k][l];
      }
      k += 2;
    }
  }
```

Note that each time we have taken two members from the parent pool in order to create two offspring, who are made from single-point crossover operations. We can also choose the parents differently from the pool (Goldberg, 1989). After we complete the reproduction of $n_r = n_g/2$ offspring, we need to rearrange all the chromosomes, parents and offspring, into an increasing order of the cost. This is necessary in preparing the gene pool before any mutation takes place. The following method shows how to rearrange the chromosomes.

```
// Method to rank chromosomes in the population.

  public static void rank() {
    boolean d[][] = new boolean[ng][nd];
    boolean w[] = new boolean[nd];
    double r[] = new double[nv];
    double e[] = new double[ng];
    int index[] = new int[ng];
    for (int i=0; i<ng; ++i) {
      for (int j=0; j<nd; ++j){
        w[j] = c[i][j];
        d[i][j] = w[j];
      }
      r = decode(w,nb);
      e[i] = cost(r);
      index[i] = i;
    }
    sort(e,index);
    for (int i=0; i<ng; ++i){
      f[i] = e[i];
      for (int j=0; j<nd; ++j) c[i][j] = d[index[i]][j];
    }
  }
```

Note specifically how the index array is used to help relabel the chromosomes according to the cost of each configuration.

Mutation operation

Another effective way of exploring the cost surface in the genetic algorithm is through the mutation process. This is achieved by randomly reversing the bits in the randomly selected chromosomes. There are two issues related to mutation.

First is the percentage of bits in the entire gene pool that are to be mutated in each generation. A typical case is to select about 1% of bits randomly for mutation, 0 (false) to 1 (true) and 1 (true) to 0 (false). The higher the percentage the bigger the fluctuation. However, if a larger fraction of bits is mutated in each generation, the cost surface can be explored faster, but it may mean that the best cost is skipped in the process. So in an actual calculation, we need to experiment with different percentages for a specific given problem (cost function) in order to find a moderate percentage that will allow us to explore the cost surface fast enough without missing the best cost configuration.

Another issue is the number of configurations that we would like to keep immune from mutation. Under any mutation scheme, the best chromosome is always kept unchanged. But we may also want a few of the next-best chromosomes to be immune from mutation. This slows down the exploration of the cost surface by mutation but increases the rate of convergence because those configurations may have already contain a large fraction of excellent genes in their chromosomes. In practice, we need to experiment in order to find the most suitable percentage of bits to be mutated for the specific problem. The following method is an example of performing mutation on a population with a given percentage of bits to be reversed.

```
// Method to mutate a percentage of bits in the selected
// chromosomes except the best one.

  public static void mutate() {
    int mmax = (int)(ng*nd*pm+1);
    Random rnd = new Random();
    double r[] = new double[nv];
    boolean w[] = new boolean[nd];

 // Mutation in the elite configurations
    for (int i=0; i<ne; ++i) {
      for (int l=0; l<nd; ++l) w[l] = c[i][l];
      int mb = (int)(nd*pm+1);
      for (int j=0; j<mb; ++j){
        int ib = (int)(nd*rnd.nextDouble());
        w[ib] = !w[ib];
      }
      r = decode(w,nb);
      double e = cost(r);
      if (e<f[i]){
        for (int l=0; l<nd; ++l) c[i][l] = w[l];
        f[i] = e;
      }
    }

 // Mutation in other configurations
    for (int i=0; i<mmax; ++i) {
       int ig = (int)((ng-ne)*rnd.nextDouble()+ne);
       int ib = (int)(nd*rnd.nextDouble());
       c[ig][ib] = !c[ig][ib];
    }

 // Rank the chromosomes in the population
    rank();
  }
```

Note that we have kept a certain number of elite configurations immune from mutation unless the attempted mutation improves their costs. All the configurations are rearranged according to their costs after each round of mutation operation. Now we are ready to see how the algorithm works in actual problems.

11.2 The Thomson problem

In electrostatics, charges will distribute to minimize the electrostatic energy under the equilibrium condition implied. This is the so-called Thomson theorem. For example, we can show that in a system of conductors, each forms an equipotential surface if the charge placed on each conductor is fixed and the total electrostatic energy of the system is minimized. However, a problem arises when the equilibrium configuration of a significant number of discrete charges is sought, for example, the stable geometry of n_c identical point charges confined on the surface of a unit sphere. This problem, known as the Thomson problem, originates from two complexities: the nonlinearity in the behavior of the system and a large number of low-lying energy levels.

We can obtain definite answers for some limited cases. For example, we can prove that the charges will cover the entire surface uniformly if $n_c \to \infty$ and that the symmetric geometries to keep all the charges far apart are the stable configurations for small n_c, such as an equilateral triangle for $n_c = 3$, a tetrahedron for $n_c = 4$, a twisted and stretched cube for $n_c = 8$, and so forth. However, the problem becomes increasingly complicated as n_c increases. For example, when n_c reaches 200, the number of nearly degenerate low-lying energy levels is about 8000. The problem with a large n_c is as yet still considered unsolved. Here we want to demonstrate the application of the genetic algorithm to the Thomson problem to demonstrate the strength of the scheme. Furthermore, the calculations achieved with the genetic algorithm for $n_c \leq 200$ are the best results obtained so far (Morris, Deaven, and Ho, 1996).

Mathematically, the Thomson problem is to find the configuration that minimizes the electrostatic energy

$$U = \frac{q^2}{4\pi\epsilon_0} \sum_{i>j=1}^{n_c} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}, \tag{11.5}$$

where q is the charge on each particle, ϵ_0 is the electric permittivity of free space, and $\mathbf{r}_i$ is the position vector of the ith charge. Because all the charges are confined on the surface of the unit sphere, $r_i \equiv 1$. We will take $q^2/4\pi\epsilon_0 = 1$, reflecting a special choice of units, for convenience. If we represent the Cartesian coordinates in terms of the polar and azimuthal angles, we have

$$\begin{aligned} x_i &= \sin\theta_i \cos\phi_i, \\ y_i &= \sin\theta_i \sin\phi_i, \\ z_i &= \cos\theta_i. \end{aligned}$$

Here we have taken the radius of the sphere to be unit. The following method is an implementation of the evaluation of the cost (total electrostatic energy of the system) in the Thomson problem.

```
// Method to evaluate the cost for a given variable array.

  public static double cost(double r[]) {
    double g = 0;
    double theta[] = new double[nc];
    double phi[] = new double[nc];

    for (int i=0; i<nc; ++i){
      theta[i] = Math.PI*r[i];
      phi[i] = 2*Math.PI*r[i+nc];
    }

    for (int i=0; i<nc-1; ++i){
      double ri = Math.sin(theta[i]);
      double xi = ri*Math.cos(phi[i]);
      double yi = ri*Math.sin(phi[i]);
      double zi = Math.cos(theta[i]);
      for (int j=i+1; j<nc; ++j){
        double rj = Math.sin(theta[j]);
        double dx = xi - rj*Math.cos(phi[j]);
        double dy = yi - rj*Math.sin(phi[j]);
        double dz = zi - Math.cos(theta[j]);
        g += 1/Math.sqrt(dx*dx+dy*dy+dz*dz);
      }
    }
    return g;
  }
```

Fig. 11.1 The stable structures of five and eight charges on a unit sphere.

Note that the input variables are assumed to be in the region [0, 1] and we therefore convert them to the correct regions with $\theta_i \in [0, \pi]$ and $\phi \in [0, 2\pi]$, for $i = 1, 2, \ldots, n_c$. To test the validity of each part of the computer code that we have developed for the Thomson problem, we search first for the geometric configurations of small systems. In Fig. 11.1 we show the stable configurations that we have obtained for $n_c = 5$ and $n_c = 8$. We obtain these stable configurations in just about 10 000 generations with the total energies $E_5 = 6.4747$ and $E_8 = 19.675$, respectively. The energies obtained are fully converged for both cases (Wille, 1986).

Because the search in the genetic algorithm is nonlinear, we do not need to worry about the degeneracy of the global rotation of the entire system. When we performed the search for the stable configuration of the ionic cluster $(\mathrm{Na}^+)_n(\mathrm{Cl}^-)_m$ in Chapter 5, we had to remove the motion of the center of mass and the global rotation about an axis through the center of mass, because the search there was in fact a linear search. We can, of course, modify the above method to have one charge sitting at the north pole and another in the xz plane at a variable latitude. The following part of the code shows the assigning of the angles in such a manner.

```
theta[0] = 0;
phi[0] = 0;
theta[1] = Math.PI*r[nv-1];
phi[1] = 0;
int k = 0;
for (int i=2; i<nc; ++i){
  theta[i] = Math.PI*r[k];
  phi[i] = 2*Math.PI*r[k+1];
  k += 2;
}
```

If we replace the corresponding part in the method for the evaluation of the cost, we have removed any global rotation in the simulation.

The Thomson problem is interesting because the number of low-lying excited states increases with the number of charges exponentially. However, we know that the solution is the configuration that spreads out the charges in the most uniform manner; the charges try to avoid each other as much as they can, but confinement on a finite surface forces them to find a compromise. This type of competition is at the heart of the modern theory of the quantum many-body systems. The

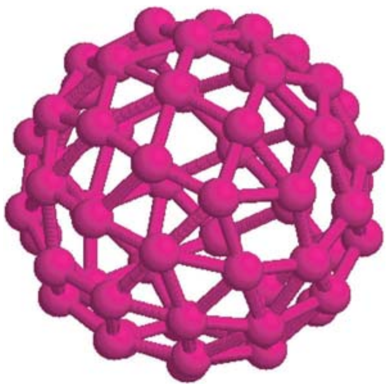


Fig. 11.2 The final configuration of 60 charges on a unit sphere after 60 000 generations of the search under genetic algorithm.

compromise between two conflicting effects drives a system into an exotic state that is typically beyond intuitive thinking or creates surprises. This is perhaps the reason why the Thomson problem is so fascinating. In Fig. 11.2 we show the final configuration of a system of 60 charges, after 60 000 generations. Even though the system is still not fully converged, we can already see the clear distinction between this system and the buckminsterfullerene structure of the molecule C_{60}.

We can, of course, use the same program to search for the stable configuration of even larger systems. The computing time required to obtain a stable configuration will increase, but in principle, we can obtain the stable configuration of multicharges located on the surface of a unit sphere. There are still many interesting issues related to the Thomson problem that are under constant debate, interested readers can find some of these discussions in Pérez-Garrido and Moore (1999) and Morris, Deaven, and Ho (1996).

11.3 Continuous genetic algorithm

The advantage of the binary genetic algorithm lies in its clarity and the transparency of the evolutionary mechanism built in through the three basic operations, selection, crossover, and mutation. However, sometimes the encoding scheme can become unmanageable if accuracy in the array is greatly desired. Examples include cases in which a significant number of local minima are located close together. It then becomes desirable for the variables to remain continuous real parameters and to utilize the machine or language accuracy for the variables.

An alternative to the binary scheme is to code the arrays as real parameters directly. So a chromosome is simply an array of real variables involved in the cost function. The three operations, selection, crossover, and mutation, have to be adjusted in order to work with the real parameters directly instead of operating on the binary strings.

The selection is done in a nearly identical manner because the cost function corresponding to each individual chromosome is always used in any selection scheme. For example, if we still use the tournament method, we can implement the selection using the following method as in the binary scheme.

```
// Method to run tournaments for selecting the parents.
  public static void select() {
    int index[] = new int[ng];
    double d[][] = new double[ng][nv];
    double e[] = new double[ng];
    for (int i=0; i<ng; ++i){
      for (int l=0; l<nv; ++l) d[i][l] = c[i][l];
      e[i] = f[i];
      index[i] = i;
    }
    shuffle(index);
    int k = 0;
    for (int i=0; i<nr; ++i) {
      if (e[index[k]] < e[index[k+1]]){
        for (int l=0; l<nv; ++l) c[i][l]=d[index[k]][l];
        f[i] = e[index[k]];
      }
      else {
        for (int l=0; l<nv; ++l) c[i][l]=d[index[k+1]][l];
        f[i] = e[index[k+1]];
      }
      k += 2;
    }
  }
```

Note that the only difference between the above selection and that of the binary code is that the chromosomes now are stored as real variables instead of binaries.

The crossover can be achieved in several different ways. The simplest is to swap a part of the real arrays. For example, the single-point crossover can be implemented as in the following method.

```
// Method to perform single-point crossover operations.
  public static void cross() {
    Random rnd = new Random();
    int k = 0;
    for (int i=nr; i<nr+nr/2; ++i) {
      int nx = 1 + (int)(nv*rnd.nextDouble());
      for (int l=0; l<nx; ++l){
        c[i][l] = c[k][l];
        c[i+nr/2][l] = c[k+1][l];
      }
      for (int l=nx; l<nv; ++l){
        c[i][l] = c[k+1][l];
        c[i+nr/2][l] = c[k][l];
      }
      k += 2;
    }
  }
```

Note again that the code appears to be virtually identical to that of the binary code, but the crossover is not happening at an arbitrary point that can be the middle of a binary string. Instead, it is at a selected location of the real array that represents a chromosome. The crossover in the binary code allows the change to happen in the middle of a binary segment that represents a single real variable. But the real-parameter version above swaps the variables themselves. There are other ways to implement crossover that allow a partial change of a real variable (Goldberg, 1989).

The most significant departure from the binary scheme is in the mutation operation. When we have the binary representation of a chromosome, binary bits are natural and we expect a bit to change from true to false or vice versa if mutation occurs. But when the chromosomes are given in real values, it is not entirely clear what change a variable will be subject to if mutation occurs. One way to implement mutation is with a random number. For example, if a real number is chosen to mutate according to the percentage of mutation specified, we can replace it with a uniform random number to the region [0, 1]. Note that all the variables are restricted to the region [0, 1]. The replacement with a random number would be equivalent to having a percentage of bits altered if the variable were represented by a segment of binaries. The following method is an implementation of such a mutation scheme.

```
// Method to mutate a percentage of bits in the selected
// chromosomes except the best one.

  public static void mutate() {
    Random rnd = new Random();
    double r[] = new double[nv];

 // Mutation in the elite configurations
    for (int i=0; i<ne; ++i) {
      for (int l=0; l<nv; ++l) r[l] = c[i][l];
      int mb = (int)(nv*pm+1);
      for (int j=0; j<mb; ++j){
        int ib = (int)(nv*rnd.nextDouble());
        r[ib] = rnd.nextDouble();
      }
      double e = cost(r);
      if (e<f[i]){
        for (int l=0; l<nv; ++l) c[i][l] = r[l];
        f[i] = e;
      }
    }

 // Mutation in other configurations
    int mmax = (int)((ng-ne)*nv*pm+1);
    for (int i=0; i<mmax; ++i) {
      int ig = (int)((ng-ne)*rnd.nextDouble()+ne);
      int ib = (int)(nv*rnd.nextDouble());
      c[ig][ib] = rnd.nextDouble();
    }

 // Rank the chromosomes in the population
    rank();
  }
```

When the above changes were implemented in a real-parameter genetic algorithm and applied to the Thomson problem, it was found that the scheme was as powerful as the binary version. The advantage of the real-parameter code is that it avoids having to encode and decode the chromosomes back and forth every time the variables are used or stored and therefore saves a large portion of computing time.

11.4 Other applications

The potential of the genetic algorithm was not realized immediately when the scheme was first introduced (Holland, 1975), but it picked up the pace afterwards (Goldberg, 1989). Now the scheme is applied in many fields, including business, economics, social studies, engineering, biology, physical sciences, computer science, and mathematics. In order to show its importance, we highlight a few applications in this section. The discussion is far from being complete; interested readers should search the current literature to obtain a better glimpse of what is going on with the genetic algorithm.

Molecules, clusters, and solids

The Thomson problem can be generalized in the search for stable structures of other small clusters of atoms and molecules that can be well described by a classical n-body interaction potential $V(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n)$, which is usually obtained from some types of first-principles calculations, such as the glue potential for gold clusters (Ercolessi, Tosatti, and Parrinello, 1986).

The simplest n-body interaction is an isotropic pairwise (two-body) interaction, such as the Coulomb interaction involved in the Thomson problem. For example, the interaction between two inert gas atoms is well described by the (two-body) Lennard–Jones potential

$$V(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \tag{11.6}$$

which we introduced in Chapter 8 to illustrate the molecular dynamics simulation. We can follow what we did with the Thomson problem to find the stable geometric configurations of clusters of inert gas atoms. However, there is one change that must be made in order to accommodate these unrestricted clusters. The particles are allowed to move along the r direction as well as the θ and ϕ directions. We can still keep all the variables finite by introducing a cut-off radius r_0, which can be viewed as the largest distance the particle can reach from the center of the clusters. Unless the cluster is falling apart, this choice of cut-off radius does not affect the stable configuration of the clusters. The following method shows how the cost is calculated in the program.

Fig. 11.3 The stable structures of five and eight particles that interact with each other through the Lennard–Jones potential.

```
// Method to evaluate the cost for a given variable array.

  public static double cost(double r[]) {
    double g = 0;
    double ri[] = new double[nc];
    double theta[] = new double[nc];
    double phi[] = new double[nc];

    int k = 0;
    for (int i=0; i<nc; ++i){
      ri[i] = r0*r[k];
      theta[i] = Math.PI*r[k+1];
      phi[i] = 2*Math.PI*r[k+2];
      k += 3;
    }

    for (int i=0; i<nc-1; ++i){
      double rhoi = ri[i]*Math.sin(theta[i]);
      double xi = rhoi*Math.cos(phi[i]);
      double yi = rhoi*Math.sin(phi[i]);
      double zi = ri[i]*Math.cos(theta[i]);
      for (int j=i+1; j<nc; ++j){
        double rhoj = ri[j]*Math.sin(theta[j]);
        double dx = xi - rhoj*Math.cos(phi[j]);
        double dy = yi - rhoj*Math.sin(phi[j]);
        double dz = zi - ri[j]*Math.cos(theta[j]);
        double r6 = Math.pow((dx*dx+dy*dy+dz*dz),3);
        double r12 = r6*r6;
        g += 1/r12-2/r6;
      }
    }
    return g;
  }
```

Note that the variables are still kept in the region of [0, 1] for convenience. Because of this choice, the main methods for selection, crossover, and mutation are unchanged. In Fig. 11.3, we show the stable configurations of the Lennard–Jones clusters with five and eight particles. We have used the real-parameter version of the genetic algorithm discussed in the preceding section in the search, and ε and $2^{1/6}\sigma$ as the units of energy and length, respectively. The potential energies per particle are $E_5/5 = -1.8208$ and $E_8/8 = -2.4776$. Note that even though the clusters studied here are extremely small, they already appear to be a stack of tetrahedrons that are the building blocks of a face-centered cubic or hexagonal close-packed lattice.

The search can be enhanced with a better design of the crossover operation, especially for large clusters. One type of such is called basin hopping, in which the potential energy surface has many local minima and the new configurations are created to overcome the barriers between these basins by swapping certain particles from different configurations. For example, we can cut two different configurations of the cluster through a randomly chosen plane that goes through or is near the center of mass of the cluster, splitting the cluster into two equal halves of compensated parts, and then exchange the parts to create two offspring (Deaven and Ho, 1995). This catchment basin transformation of the potential energy surface can also be built into other optimization schemes (Wales and Scheraga, 1999).

We can also use the genetic algorithm to search for the stable lattice structure of a solid. There is an additional issue that needs to be addressed. In practice, we cannot deal with an infinite system directly. However, we can always put the particles in a box and then impose the periodic boundary condition on the system under study. Because we are searching for a crystal structure, we also need to find ways to relax the size and the shape of the simulation box, as we did in Chapter 8 for the molecular dynamics simulation of an infinite system. In fact, we can combine the genetic algorithm with a typical simulation technique, such as molecular dynamics, or an ab initio total energy calculation from, for example, density function theory in the study of the phase transition of bulk materials. For example, a combination of the genetic algorithm and a solution of the Ornstein–Zernike equation provides an effective exploration of the phase diagram of spherical polyelectrolyte microgels (Gottwald *et al.*, 2004), and a combination of the genetic algorithm and the density functional theory allows an extensive search for the most stable four component alloys out of 192 016 possible fcc and bcc structures formed in 32 different metals (Jóhannesson *et al.*, 2002).

Chaos forecast

There have been many applications of genetic algorithm in all sorts of nonlinear processes, including in the study of dynamical systems. For example, if there is a given time sequence $(y_1, y_2, \dots, y_n)$ that can be either a data set from an experimental measurement or one from a computer simulation, we can use the genetic algorithm to obtain the information hidden in the sequence. A successful analysis of the sequence would provide a good description of the nonlinear terms involved and the nature of the sequence, linear, periodic, random, or chaotic.

The fundamental problem here is to find the appropriate function form of the time dependence in the dynamic variable

$$y_i = f(y_{i-1}, y_{i-2}, \dots, y_{i-k}) \tag{11.7}$$

from the given time sequence. Here k is an assumed number of previous steps that determine the current dynamics. The idea is to decompose $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$ into all possible building blocks and let a selection process that follows the genetic concept take place. After generations of evolution, we expect good blocks to be amplified and bad blocks eliminated (Szpiro, 1997; López, Álvarez, and Hernández-García, 2000). As soon as we find the exact form of the function, or the closest one possible, we can forecast the future behavior of the dynamical system, including chaos.

Depending on how much information in $f(y_{i-1}, y_{i-2} \ldots, y_{i-k})$ is known, we can construct an approximate cost for the search process. If we already know the model, namely, the function form of $f(y_{i-1}, y_{i-2} \ldots, y_{i-k})$, we can find the precise values of the parameters involved in the model by constructing a chromosome that represents a specific choice of the parameter set. For example, if we have l parameters, $v_1, v_2, \ldots, v_l$, in the model function $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$, the cost can be chosen as

$$g(v_1, v_2, \ldots, v_l) = \sum_{i=k+1}^{n} (y_i' - y_i)^2, \tag{11.8}$$

where y_i' is the predicted value based on the given set of parameters in the whole sequence and y_i is the actual value. Then we can follow the three operations, selection, crossover, and mutation, in the genetic algorithm to tune the parameter set in order to find the appropriate ones.

When we do not know the exact form of $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$, we can create chromosomes from the results of certain mathematical operations on the variables $y_{i-1}, y_{i-2}, \ldots, y_{i-k}$. Following the evolution of the chromosomes, we can at least find an approximate form of $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$ that can describe certain aspects of the system, even if it is chaotic (Szpiro, 1997). The mathematical operations can be the four basic operations, addition, subtraction, multiplication, and division, or more advanced operations, such as logarithmic, trigonometric, or logic operations. Care must be taken to avoid forbidden operations, such as division by a zero or taking the square root of a negative quantity in the case of real operations only. The selection can be achieved in the manner we have discussed, but the crossover can only be done by swapping certain building blocks or operations. This is quite similar to the real-parameter version of the genetic algorithm scheme. Mutation is achieved in a closely related manner by randomly swapping a given percentage of two operations or two building blocks that are also randomly selected.

Best strategy in a game

It is commonplace for a corporation to apply the concept of game theory in developing its business strategies. Game theory is the study of all the possible responses that each individual can make in a game (a process that involves mutually interacting rational players) and the consequence (gain or loss) of each response (Osborne, 2004). The essential part of a successful analysis of a game is figuring out the strategy for an individual player to obtain the maximum gain in the long run, based on the available record.

Here we use a simple example of the adaptive minority game (Sysi-Aho, Chakraborti, and Kaski, 2004) in order to illustrate the potential of applying the concepts of the genetic algorithm. The simple minority game has an odd number of N players with only two possible actions that can be recorded as a binary 0 or 1. A player wins a point by ending up in the minority group. Assuming that all the players have access to the M most recent results with each recorded also as a binary, 0 for the winning of 0's group and 1 for the winning of 1's group. So the record is one of the 2^M possible combinations. For each combination, a player can choose either 0 or 1, that makes a total of 2^{2^M} possible strategies for each player. However, each player is only allowed to have a finite number of S strategies in his strategy pool from which to choose. The adaptive minority game allows a player to modify his strategy pool. This modification can be constructed according to the genetic algorithm. For example, we can map each strategy into a gene that contains a binary string of a possible record and action. The chromosome is a binary string of the genes of all the agents at a given time t, which is increased by a unit each time the game is played. Among different strategies, we can create a population at a given time. The performance of the system is measured by the time-dependent cost (called the utility)

$$U(x) = \frac{1}{x_0}\left[x + \Theta(x - 2x_0)(N - x)\right], \tag{11.9}$$

where $x \in [1, N]$ is the number of agents that take a specific action (1 or 0) at time t, $x_0 = (N-1)/2$ is the maximum number of possible winners, and $\Theta(y) = 1$ for $y > 0$, zero otherwise. The maximum cost is normalized to 1 and decreases if x deviates from the average $N/2$. It has been shown that a single-point crossover operation can improve the performance of an individual agent as well as the system as a whole (Sysi-Aho, Chakraborti, and Kaski, 2004). It would be interesting to see whether the selection and mutation operations are able to do the same.

The minority game is not unique in the sense of having the characteristic that there is a gain for each individual and an overall efficiency of the system. We can identify many economical and social phenomena that carry such a generic feature. More applications of the concepts of genetic algorithm are expected to emerge in near future.

11.5 Genetic programming

The problems involved in game theory and chaos forecasting are more complicated than that of finding the stable configuration of an atomic cluster; they require optimization of the processes involved. If we imagine that each of the strategies taken by an agent in the minority game is a possible computer operation (represented there by a binary string), the entire sequence of strategies can be viewed as a piece of symbolic computer program and the sequence that receives the best overall performance is the best available computer program to achieve the task of the minority game.

The scheme that selects the best set of operations (best program), based on the concept of evolution, from all the possible sets of operations (possible programs) for solving a certain problem or achieving a certain task is called genetic (or evolutionary) programming, which was first introduced in the early 1960s (Fogel, 1962; Fogel, Owens, and Walsh, 1966). The scheme has experienced much renovation since then (Koza, 1992; 1994). For an introduction to the subject, see Jacob (2001). The main difference between genetic programming and the genetic algorithm is that the genetic algorithm tries to find the configuration (chromosome) that optimizes the cost, but genetic programming tries to find the best process among all the possible processes according to the assigned fitness. So genetic programming creates computer codes for certain objectives and modifies them according to evolutionary principles. This is much more proactive in the sense of creating a certain artificial intelligence in the programs.

For example, if we want to design software that can interpret the Chinese characters written with an electronic pen on a pat that is formed with a matrix of capacitors and resistors, the cost (the measure of fitness) function could be the percentage of characters that have been interpreted wrongly since the beginning of the writing. This is a difficult task because the software must be able to recognise the characters even when there are differences in the hand-writing. But this is the type of the problem for which genetic programming works the best, with a simple fitness function that depends on a large number of variables from different handwriting styles.

Genetic programming involves roughly the same pieces of ingredients as in the genetic algorithm. First, an initial pool of computer programs, formed with *functions* (computational operations) and *terminals* (elements to be operated on) of the possible solutions of the problem, is created. Parents are then selected according to the fitness of each program. Finally, new generations are created with crossover and mutation. The key difference between the genetic algorithm and genetic programming is what is being modified – configurations of the variables in a genetic algorithm but computational operations or elements in a genetic program.

A typical function in genetic programming is an arithmetic operation, such as addition, division, or taking a square root, but it can also be a more sophisticated operation, like a function operation $z = f(x, y)$ on the two terminals (elements) x and y. The simplest terminal is typically a variable or parameter; a general terminal is a combination of many variables and parameters through functions. For example, if the function is $z = e^x \ln(y - 1)$, we can consider the terminals to be x and y, which can be combinations of some other variables and parameters,

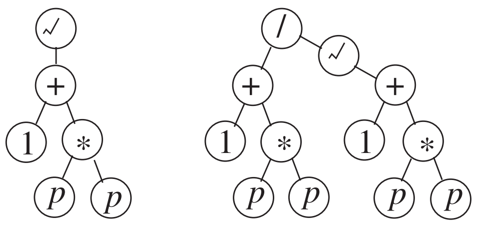


Fig. 11.4 Two different tree diagrams that represent the same energy equation.

such as $x = a^2 + 2b - \sqrt{c}$ and $y = 3a + b^2 - \ln c$. Of course, $y > 1$ and $c > 0$ if we are dealing with real quantities only.

One of the difficulties in designing a genetic program is in figuring out an appropriate fitness (cost) associated with each possible program. The cost can only be evaluated through running the program. In some cases in order to obtain a fair estimate of the cost, a program must be run multiple times, resulting in a time consuming process.

Let us take an extremely simple example in physics to highlight how genetic programming works. Consider that we are searching for the relation between the energy E and momentum p of a relativistic particle of mass m, which is given as

$$E = c\sqrt{m^2c^2 + p^2}, \tag{11.10}$$

where c is the speed of light in vacuum. If we use mc^2 as the unit of energy and mc as the unit of momentum, we have $E = \sqrt{1 + p^2}$. We can formally use a *tree diagram* to represent the actual equation. Note that each function in this problem can only have one or two terminals. When there are two terminals, the operation of the function is binary, from left to right. There are many possible representations (tree diagrams) for the equation, two of which are shown in Fig. 11.4.

For simplicity, we assume that there is a cut-off for the momentum $p \leq p_c$ and the functions are five elementary operations: addition (+), subtraction (−), multiplication (∗), division (/), and taking the square root ($\sqrt{}$). We further assume that the equation is given at n discrete points with $E_i = E(p_i)$, for $i = 1, 2, \ldots, n$. Now we can create the initial pool of equations by treating p and 1 as the only variable and parameter that can be used repeatedly. We can limit the number of the uninterrupted segments in the longest branch in the tree diagram to four. Note that we also must avoid dividing by a zero or having a negative value within a square root. We can simply eliminate that diagram if it is the case.

After we have the initial pool, we can assign a fitness (cost) to each of the diagrams based on

$$f = \sum_{i=1}^{n} \Delta E_i^2, \tag{11.11}$$

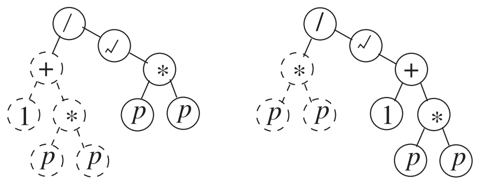


Fig. 11.5 Two possible parents that can be used to create offspring by swapping the dashed branches.

where ΔE_i is the energy difference between a tree diagram in the pool and that of actual value. Based on the cost, we can run tournaments to select parents.

Assume that we have found two parents as given in Fig. 11.5. We can perform crossover to create offspring. The crossover is performed between the two parents at similar functions, two-terminal or one-terminal. For example, if the crossover is done by swapping the dashed branches, we obtain two offspring, with one of them being the exact equation sought. Note that a crossover can also be performed on two identical parents with different parts swapped, which is equivalent to swapping different segments of two identical chromosomes in a genetic algorithm.

Mutation can be performed at each vertex. For example, if a function is selected randomly to be mutated, we can use one of the five operations to replace it. If the variable is selected to be mutated, we can replace it with the parameter, or vice versa. Note that we still have to keep the equation valid, that is avoid having a negative value inside a square root or dividing a quantity by a zero.

There have been much activity in genetic programming, including progress in different methods of assigning cost, performing crossover, and carrying out mutation. The field is still fast growing, especially in the areas of artificial intelligence and machine learning. Interested readers can find detailed discussions on these subjects in Ryan (2000), Langdon and Poli (2002), and Koza *et al.* (1999; 2003).

Exercises

11.1 Assuming that the interaction between Na^+ and Cl^- in a NaCl molecule is given by

$$V(r) = -\frac{e^2}{4\pi\epsilon_0 r} + V_0 e^{-r/r_0},$$

with $V_0 = 1.09 \times 10^3$ eV and $r_0 = 0.330$ Å, find the bond length of the molecule with the genetic algorithm.

11.2 Find the minimum of $f(x, y, z) = y \sin(4\pi x) + 2x \cos(8\pi y)$ in the region of $x, y \in [-1, 1]$ with the genetic algorithm.

11.3 Find the stable geometric structures of clusters of ions $(\text{Na}^+)_n(\text{Cl}^-)_m$ with small integers n and m by a genetic algorithm search.

11.4 Search for the stable geometric structures of small Lennard–Jones clusters. Specifically, find the Mackay icosahedron for 55 particles and decahedron for 75 particles. Are the Mackay icosahedron and decahedron the stable structures of the given systems?

11.5 Using the four basic operations, addition, subtraction, multiplication, and division, and the data points from the previous time steps, find the correct expression for the right-hand side of the Duffing model

$$\frac{d^2x}{dt^2} = b\cos t - g\frac{dx}{dt} - x^3$$

and forecast chaos.

11.6 Create a long set of data points with an equal-time step for a simple harmonic oscillator, $(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n)$, where $\mathbf{y}_k$ for $k = 1, 2, \ldots, n$ are two-component vectors, with one component for the angle of the pendulum away from the vertical and the other for its time derivative. Determine the optimal function form of **f** from

$$\mathbf{y}_i = \mathbf{f}(\mathbf{y}_{i-1}, \mathbf{y}_{i-2}, \ldots, \mathbf{y}_{i-k}),$$

for $k = 1, 2, 3, 4$. What happens if the pendulum is a driven pendulum with damping?

11.7 Generate a long, random set of data and determine that it cannot come from any dynamical system. A specific model can be used as a test.

11.8 Develop a genetic algorithm program to play the optimal minority game. When each of the three operations, selection, crossover, and mutation are analyzed, which one is most critical?

11.9 Use genetic programming to create a solution of a two-dimensional maze built from $n \times n$ squares.

11.10 The Chinese board game *Weiqi* has 361 vertices through 19×19 perpendicular lines. Two players take turns to occupy the vertices, one at a time with two types of stones (black and white), one for each player. If one player completely surrounds one or more of his opponents stones, he removes the surrounded stones. The winner is the one who occupies most vertices at the end. Using the genetic algorithm or genetic programming, develop a computer program that can play Weiqi against a skillful human player.

Chapter 12
Numerical renormalization

Renormalization is a method that was first introduced in quantum field theory to deal with far-infrared divergence in Feynman diagrams. The idea was further explored in statistical physics to tackle the problems associated with adiabatic continuity problems in phase transitions and critical phenomena. So far, the renormalization method has been applied to many problems in physics, including chaos, percolation, critical phenomena, and quantum many-body problems.

Even though many aspects of the renormalization group have been studied, it is still unclear how to apply the method to study most highly correlated systems that we are interested in now, for example, the systems described by the two-dimensional Hubbard model. However, it is believed by some physicists that the idea of renormalization will eventually be applied to solve the typical quantum many-body problems we are encountering, especially with the availability of modern computers, new numerical algorithms, and fresh ideas for setting up proper renormalization schemes.

12.1 The scaling concept

The renormalization technique in statistical physics is based on the scaling hypothesis that the competition between the long-range correlation and the fluctuation of the order parameters is the cause of all the singular behavior of the relevant physical quantities at the critical point. The longest length of the order parameter that stays correlated at a given temperature T is defined as the correlation length of the system, which diverges when the system approaches the critical temperature T_c:

$$\xi \propto t^{-\nu}, \tag{12.1}$$

where ξ is the correlation length, $t = |T - T_c|/T_c$ is the reduced temperature, and ν is the critical exponent of the correlation length. The scaling hypothesis states that the divergence of all the other quantities at the critical temperature is the result of the divergence of the correlation length. Based on this hypothesis, we can obtain several relations between the exponents of different physical quantities. Detailed discussions on the scaling hypothesis and exponent relations can be found in

Ma (1976) and Kadanoff (2000). Here we just want to give a brief discussion of the scaling concept introduced by Widom (1965a; 1965b) and Kadanoff (1966). Widom's idea is based on the fact that the thermodynamic potentials, such as the Helmholtz free energy, are homogeneous functions, which scale with the extensive variables of the system. For example, the internal energy doubles if we double the volume and entropy of the system.

If we take a spin system as an example, the Helmholtz free energy satisfies

$$F(t, B) = \Lambda^{-1} F(\Lambda^x t, \Lambda^y B), \tag{12.2}$$

where Λ is a parameter that changes the length scale of the system and x and y are two exponents determining how the reduced temperature t and the magnetic field strength B scale with Λ. We can view the above equation as dividing the system by a length scale Λ along one specific direction with x and y characterizing the corresponding changes in the intensive variables t and B. The critical behavior of the magnetization m is characterized by two exponents β and δ as

$$m(t, B = 0) \propto t^{\beta}, \tag{12.3}$$

$$m(t = 0, B) \propto B^{1/\delta}. \tag{12.4}$$

Now let us show how we can derive the relations of the critical exponents from the assumption that the free energy is a homogeneous function. The magnetization is defined as

$$m(t, B) = -\frac{\partial F(t, B)}{\partial B}, \tag{12.5}$$

which leads to a rescaling relation

$$m(t, B) = \Lambda^{y-1} m(\Lambda^x t, \Lambda^y B), \tag{12.6}$$

which in turn gives

$$\beta = \frac{1-y}{x}, \tag{12.7}$$

$$\delta = \frac{y}{1-y}, \tag{12.8}$$

if we take $\Lambda^x t = 1$ and $B = 0$, and $t = 0$ and $\Lambda^y B = 1$. Similarly, we can obtain the exponent γ of the susceptibility from the definition

$$\chi(t, B = 0) = \left.\frac{\partial m(t, B)}{\partial B}\right|_{B=0} = \Lambda^{2y-1} \chi(\Lambda^x t, 0) \propto t^{-\gamma}, \tag{12.9}$$

which gives

$$\gamma = \frac{2y-1}{x} = \beta(\delta - 1), \tag{12.10}$$

if we take $\Lambda^x t = 1$. Following exactly the same process, we can obtain the exponent α for the specific heat

$$C(t, B = 0) = \left.\frac{\partial^2 F(t, B)}{\partial t^2}\right|_{B=0} \propto t^{-\alpha}, \tag{12.11}$$

which leads to

$$\alpha = 2 - \frac{1}{x} = 2 - \beta(\delta + 1). \tag{12.12}$$

The above relations indicate that if we know x and y, we know β, δ, γ, and α. In other words, if we know any two exponents, we know the rest. In most cases, the behavior on each side of the critical point is symmetric, so we do not need to distinguish the two limiting processes of approaching the critical point. However, we have to realize that this may not always be true. So a more detailed analysis should consider them separately.

How can we relate the critical exponents of the thermodynamic quantities discussed above to the fundamental exponent of the correlation length? It was Kadanoff (1966) who first introduced the concept of the block Hamiltonian and then related the exponent of the correlation length to the critical exponents of the thermodynamic quantities. Kadanoff's idea is based on the observation that the correlation length diverges at the critical point. So the details of the behavior of the system at a shorter length than that of the correlation do not matter very much. Thus, we can imagine a process in which the spins in a block of dimension Λ point in more or less the same direction so we can treat the whole block like a single spin. The Helmholtz free energy then scales as

$$F(\Lambda^a t, \Lambda^b B) = \Lambda^d F(t, B), \tag{12.13}$$

where a and b are the rescaling exponents for t and B, respectively, and d is the dimensionality of the lattice. The above relation shows that

$$a = xd, \tag{12.14}$$

$$b = yd, \tag{12.15}$$

in comparison with the Widom scaling based on the homogeneous function assumption. The spin correlation function is defined as

$$G(r_{ij}, t) = \langle s_i s_j \rangle - \langle s_i \rangle \langle s_j \rangle, \tag{12.16}$$

which satisfies a scaling relation

$$G(r_{ij}, t) = \Lambda^{2(b-d)} G(\Lambda^{-1} r_{ij}, \Lambda^a t), \tag{12.17}$$

which leads to

$$\nu = \frac{1}{a} = \frac{1}{xd} = \frac{2-\alpha}{d}. \tag{12.18}$$

The exponent of the correlation function with the distance is given by

$$G(r_{ij}, t = 0) \propto \left(\frac{1}{r_{ij}}\right)^{d-2-\eta}, \tag{12.19}$$

which is related to the other exponents by

$$\eta = 2 - \frac{d(\delta - 1)}{\delta + 1}. \tag{12.20}$$

In the next section we will show how we can construct a renormalization scheme, the mathematical form of the Kadanoff block spin transformation, and then show how the exponents are related to the eigenvalues of the linear representation of the transformation around the critical point.

12.2 Renormalization transform

As argued by Wilson (1975), the physical quantities of statistical physics are controlled by fluctuations at all length scales up to the correlation length ξ. These fluctuations occur in every physical observable at every energy scale, and the renormalization scheme is aimed at finding the structure of this hierarchy, or its approximation, if it exists. One way to obtain the structure of the hierarchy in the ferromagnetic spin system is through the Kadanoff transform (Kadanoff, 1966), which partitions the lattice to form a new lattice with the definition of the block spins. This procedure is repeated until a stable fixed point is reached. Intuitively, we can view this transform as a process for recovering the long-range behavior of the system. For example, when the system changes from a paramagnetic state to a ferromagnetic state, at any length scale, the magnetization is more or less the same. In other words, the spins tend to line up to form ferromagnetic domains, and the domains will continue to line up to form larger domains until the whole system becomes one big ferromagnetic domain. In general, this transform can mathematically be formulated into a semigroup, that is, a series of operations without an inverse. For some simple cases, the inverse transform can also be found; a true group then results. To illustrate how this is done, assume that the system is the d-dimensional Ising spin system with a length L along each direction. We can first divide the system into many blocks, and a block spin s_i can then be formulated from all the lattice spins in that block:

$$s_i = g\{s_{i\sigma}\}, \tag{12.21}$$

where we have used $i = 1, 2, \ldots, (L/\Lambda)^d$, where Λ is the length of the block. We have also used $\{s_i\}$ to denote a configuration of s_i with $i = 1, 2, 3, \ldots$ being a generic index for the sites involved, and σ is used to label spins within a block, that is, $\sigma = 1, 2, \ldots, (\Lambda/a)^d$, with a being the lattice constant. The specific function form of $g\{s_{i\sigma}\}$ is not always simple; the choice has to reflect the physical process of the system. For the Ising model, a common practice is to use the majority rule: The block spin is assigned $+1$ (-1) if the majority of spins in the block are up (down). When the block has the same number of up and down sites, we can assign either $+1$ or -1 to the block spin with an equal probability. The transform is continued by taking the block spins as the new lattice spins

$$s_i^{(n)} = g\left\{s_{i\sigma}^{(n)}\right\}, \tag{12.22}$$

where $s_i^{(0)}$ are the original lattice spins. To simplify our notation, we will use $\mathbf{S} = (s_1, s_2, s_3, \dots)$ for a total spin configuration. The so-called renormalization transform is defined with the new distribution function

$$\mathcal{W}^{(n+1)}\left(\mathbf{S}^{(n+1)}\right) = \frac{1}{\mathcal{Z}^{(n+1)}} \sum_{\sigma} \delta\left[\mathbf{S}^{(n+1)} - \mathbf{g}\left(\mathbf{S}_\sigma^{(n)}\right)\right] e^{-\mathcal{H}^{(n)}(\mathbf{S}^{(n)})}, \tag{12.23}$$

where the effective Hamiltonian $\mathcal{H}$ is also a function of the coupling constant and the external field. Temperature is absorbed into the coupling constant and the external field. The partition function at each iteration of the transform is given by

$$\mathcal{Z}^{(n)} = \sum_{\mathbf{S}^{(n)}} e^{-\mathcal{H}^{(n)}(\mathbf{S}^{(n)})}. \tag{12.24}$$

We can easily show that the average of an observable is related through

$$\langle A \rangle = \sum_{\mathbf{S}^{(n+1)}} A\left(\mathbf{S}^{(n+1)}\right)\mathcal{W}(\mathbf{S}^{(n+1)}) = \sum_{\mathbf{S}^{(n)}} A\left(\mathbf{S}^{(n+1)}\right)\mathcal{W}\left(\mathbf{S}^{(n)}\right), \tag{12.25}$$

during the transformation. The block Hamiltonian $\mathcal{H}^{(n+1)}$ is completely given by $\mathcal{H}^{(n)}$ through the transform up to a constant. Symbolically, we can write all the parameters in the Hamiltonian as the components of a vector $\mathbf{H}$, and the transform can be expressed as a matrix multiplication

$$\mathbf{H}^{(n+1)} = \mathbf{T}\mathbf{H}^{(n)}, \tag{12.26}$$

where $\mathbf{T}$ is the transform matrix defined through the distribution function with block variables. $\mathbf{T}$ is quite complex in most problems. A *fixed point* is found if the mapping goes to itself with

$$\mathbf{H}_0 = \mathbf{T}\mathbf{H}_0, \tag{12.27}$$

which can be one of several types. If the flow lines move toward the fixed point along all the directions, we call it a *stable fixed point*, which usually corresponds to $T = 0$ or $T = \infty$. If the flow lines move away from the fixed point, we call it an *unstable fixed point*. The critical point is a mixture of unstable and stable behavior. Along some directions the flow lines move toward the fixed point, but along other directions, the flow lines move away from the fixed point.

If we expand the above renormalization equation around the fixed point, we have

$$\mathbf{H}_0 + \delta\mathbf{H}' = \mathbf{T}(\mathbf{H}_0 + \delta\mathbf{H}). \tag{12.28}$$

When $\delta\mathbf{H}$ is very close to zero, we can carry out the Taylor expansion

$$\mathbf{T}(\mathbf{H}_0 + \delta\mathbf{H}) = \mathbf{T}(\mathbf{H}_0) + \mathbf{A}\delta\mathbf{H} + O(\delta\mathbf{H}^2) \tag{12.29}$$

up to the linear terms, where $\mathbf{A}$ is given by partial derivatives

$$A_{ij} = \frac{\partial T_i(\mathbf{H}_0)}{\partial H_j}. \tag{12.30}$$

In the neighborhood of the fixed point, we then must have

$$\delta \mathbf{H}' = \mathbf{A}\delta \mathbf{H}. \tag{12.31}$$

The eigenvalues λ_α of $\mathbf{A}$ from

$$\mathbf{A}\mathbf{x}_\alpha = \lambda_\alpha \mathbf{x}_\alpha \tag{12.32}$$

determine the behavior of the fixed point along the directions of the eigenvectors. If $\lambda_\alpha > 1$, then the fixed point along the direction of $\mathbf{x}_\alpha$ is unstable; otherwise, it is stable with $\lambda_\alpha < 1$ or marginal with $\lambda_\alpha = 1$.

The critical exponent can be related to the eigenvalue of the linearized transform around the critical point. For example, the critical exponent of the correlation length is given by

$$\nu = \frac{\ln \Lambda}{\ln \lambda_\sigma}, \tag{12.33}$$

where λ_σ is one of the eigenvalues of $\mathbf{A}$ that is greater than 1. This is derived from the fact that around the critical point, the correlation function scales as

$$G(\delta \mathbf{H}) = \Lambda^{-d} G(\mathbf{A}\delta \mathbf{H}), \tag{12.34}$$

which can be used to relate Λ and λ_σ to the exponent x and then the exponent ν.

12.3 Critical phenomena: the Ising model

We will use the ferromagnetic Ising model in two dimensions to illustrate several important aspects of the renormalization scheme outlined in the preceding section. The original work was carried out by Niemeijer and van Leeuwen (1974).

The ferromagnetic Ising model is given by the Hamiltonian

$$\mathcal{H} = -J \sum_{\langle ij \rangle}^{N} s_i s_j - B \sum_{i=1}^{N} s_i, \tag{12.35}$$

where $J > 0$ is the spin coupling strength, B is the external field, and i and j are nearest neighbors. For the Ising model, s_i is either 1 or -1. Assume that the lattice is a triangular lattice. Let us take the triangle of three nearest sites as the block and use the majority rule to define the block spin, that is, $s_i = +1$ if two or three spins in the ith block are up and $s_i = -1$ if two or three spins in the block are down. We will absorb the temperature into J and B. For notational simplicity, we will suppress the interaction indices when there is no confusion. The renormalization transform is given by

$$e^{-\mathcal{H}\{s_i\}} = \sum_\sigma e^{-\mathcal{H}\{s_{i\sigma}\}}, \tag{12.36}$$

where s_i are the block spins given by

$$s_i = g\{s_{i\sigma}\} = \mathrm{sgn}(s_{i1} + s_{i2} + s_{i3}) \tag{12.37}$$

if the majority rule is adopted. Here s_i is either 1 or -1 depending on whether the majority of spins are up or down. For each value of s_i there are four spin configurations of the three spins in the block: three from two spins pointing in the same direction and one from all three spins pointing in the same direction.

We can separate the lattice Hamiltonian into two parts, one for the intrablock spin interactions given as

$$\mathcal{H}_0 = -J \sum_{i,\sigma,\mu} s_{i\sigma} s_{i\mu}, \tag{12.38}$$

and another for the interblock interactions and the external field,

$$V = -J \sum_{\langle ij\rangle,\sigma,\mu} s_{i\sigma} s_{j\mu} - B \sum_{i,\sigma} s_{i\sigma}, \tag{12.39}$$

where $\langle ij\rangle$ indicates the summation over the nearest blocks. An expectation value under the intrablock part of the Hamiltonian is given by

$$\langle A\rangle_0 = \frac{1}{\mathcal{Z}_0} \sum_{\{s_i\}} A\{s_{i\sigma}\} \delta(s_i - g\{s_{i\sigma}\}) e^{-\mathcal{H}_0\{s_{i\sigma}\}}. \tag{12.40}$$

Here the partition function

$$\mathcal{Z}_0 = \sum_{\{s_i\}} \delta(s_i - g\{s_{i\sigma}\}) e^{-\mathcal{H}_0\{s_{i\sigma}\}} = z_0^M \tag{12.41}$$

is the total contribution of the partition functions from all the M blocks, where

$$z_0 = \sum_{\sigma} e^{-\mathcal{H}_0\{s_{i\sigma}\}} = 3e^{-J} + e^{3J}, \tag{12.42}$$

is the partition function of a single block. Note that we did not counter the degeneracy with all the spins flipped in the above partition function because it would add only an overall factor.

From the definition in Eq. (12.40) of the average over the spin configurations from all the blocks, the renormalization transform can be written as

$$e^{-\mathcal{H}\{s_i\}} = \left[\sum_{\{s_i\}} \delta(s_i - g\{s_{i\sigma}\}) e^{-\mathcal{H}_0\{s_{i\sigma}\}} \right] \langle e^{-V}\rangle_0, \tag{12.43}$$

where the first part is simply given by the partition function $\mathcal{Z}_0$ and the second part needs to be evaluated term by term in an expansion if we are interested in the analytic results. In the next section, we will explain how to use Monte Carlo simulations to obtain the averages needed for the renormalization transform. Equation (12.43) is the key for performing the renormalization transform, and we can easily show that it is correct by carrying out a Taylor expansion for the exponent.

We can formally write the average left as

$$\begin{aligned} \langle e^{-V}\rangle_0 &= -\langle V\rangle_0 + \frac{1}{2}\langle V^2\rangle_0 - \cdots \\ &= \exp\{-\langle V\rangle_0 - [\langle V^2\rangle_0 - \langle V\rangle_0^2]/2 + \cdots\}, \end{aligned} \tag{12.44}$$

after Taylor expansion of the exponential function and resummation in terms of the moments of the statistical averages. The above equation then leads to the renormalization transform between the lattice Hamiltonian and the block Hamiltonian as

$$\mathcal{H}\{s_i\} = \ln \mathcal{Z}_0 + \langle V\rangle_0 - \frac{1}{2}\left[\langle V^2\rangle_0 - \langle V\rangle_0^2\right] + \cdots . \tag{12.45}$$

The right-hand side has an infinite number of terms, but we can truncate the series if the fixed points have small values of J and B.

Now we can work out the transform analytically, as well as the exponents associated with the fixed point. We already know $\mathcal{Z}_0$ from Eqs. (12.41) and (12.42). $\langle V\rangle_0$ can then be evaluated from the definition. If we concentrate on the two nearest neighboring blocks, i and j, we have

$$\langle V_{ij}\rangle_0 = -J\langle s_{i1}s_{j2} + s_{i1}s_{j3}\rangle_0 = -2J\langle s_{i1}\rangle_0\langle s_{j2}\rangle_0, \tag{12.46}$$

due to the symmetry of each block, and the average is performed over the interactions within each individual block. The average over a specific spin can easily be computed with a fixed block spin and we have

$$\begin{aligned}\langle s_{i\sigma}\rangle_0 &= \frac{1}{z_0}\sum_{s_i}\delta[s_i - \operatorname{sgn}(s_{i1}+s_{i2}+s_{i3})]e^{-J(s_{i1}s_{i2}+s_{i1}s_{i3}+s_{i2}s_{i3})}\\ &= \frac{s_i}{z_0}\left(e^{3J} + e^{-J}\right).\end{aligned} \tag{12.47}$$

If we gather all the terms above into the renormalization transform of Eq. (12.45), we obtain

$$\mathcal{H}^{(n+1)}\left\{s_i^{(n+1)}\right\} = \ln \mathcal{Z}_0^{(n)} - J^{(n+1)}\sum_{\langle ij\rangle} s_i^{(n+1)}s_j^{(n+1)} - B^{(n+1)}\sum_i s_i^{(n+1)}, \tag{12.48}$$

with

$$J^{(n+1)} \simeq 2\left(\frac{e^{3J^{(n)}} + e^{-J^{(n)}}}{e^{3J^{(n)}} + 3e^{-J^{(n)}}}\right)^2 J^{(n)}, \tag{12.49}$$

and

$$B^{(n+1)} \simeq 3\left(\frac{e^{3J^{(n)}} + e^{-J^{(n)}}}{e^{3J^{(n)}} + 3e^{-J^{(n)}}}\right) B^{(n)}, \tag{12.50}$$

which can be used to construct the matrix for the fixed points. If we use the vector representation of the Hamiltonian parameters, we have $\mathbf{H} = (H_1, H_2) = (B, J)$. The fixed points are given by the invariants of the transform. Using Eqs. (12.49) and (12.50), we obtain $B_0 = 0$ and $J_0 = 0$, and $B_0 = 0$ and $J_0 = \ln(2\sqrt{2}+1)/4 \simeq 0.3356$. In order to study the behavior around these fixed points, we can linearize the transform as discussed in the preceding section with the matrix elements given by

$$\mathbf{A} = \frac{\partial \mathbf{T(H)}}{\partial \mathbf{H}}. \tag{12.51}$$

For $(B_0, J_0) = (0, 0)$, we have

$$\mathbf{A} = \begin{pmatrix} 1.5 & 0 \\ 0 & 0.5 \end{pmatrix}, \tag{12.52}$$

which has two eigenvalues $\lambda_B = 1.5 > 1$ and $\lambda_J = 0.5 < 1$, corresponding to the high-temperature limit, where the interaction is totally unimportant. For the other fixed point, with $(B_0, J_0) = (0, 0.3356)$, we have

$$\mathbf{A} = \begin{pmatrix} 2.1213 & 0 \\ 0 & 1.6235 \end{pmatrix}, \tag{12.53}$$

which has the two eigenvalues $\lambda_B = 2.1213 > 1$ and $\lambda_J = 1.6235 > 1$, corresponding to the unstable fixed point, the critical point. Because we have truncated the series at its first-order terms, these results are only approximately correct.

From the exact Onsager solution (McCoy and Wu, 1973), we know that at the critical point, $J_c = J/k_B T_c = \ln \sqrt{3}/2 \simeq 0.2747$, which is smaller than the result for J_0 obtained above. We can also calculate the critical exponents, for example,

$$\nu = \frac{\ln \Lambda}{\ln \lambda_J} = \frac{\ln \sqrt{3}}{\ln 1.6235} = 1.1336 \tag{12.54}$$

and

$$yd = \frac{\ln \lambda_B}{\ln \Lambda} = \frac{\ln 2.1213}{\ln \sqrt{3}} = 1.3690, \tag{12.55}$$

which can be used further to obtain the exponents $\alpha = 2 - \nu d = -0.2672$ and $\delta = y/(1-y) = 2.1696$. These values are still quite different from the exact results of $\alpha = 0$ and $\delta = 15$. The discrepancies are due to the approximation in the evaluation of $\langle \exp(-V) \rangle_0$. There are two ways to improve the accuracy: we can either include the higher-order terms in the expansion for $\langle \exp(-V) \rangle_0$, which seems to improve the results quite slowly, or perform the evaluation of $\langle \exp(-V) \rangle_0$ numerically, as we will discuss in the next section. Numerical evaluation of the averages in the renormalization transform has been a very successful approach.

12.4 Renormalization with Monte Carlo simulation

As we discussed in the preceding section, we have difficulty in obtaining an accurate renormalization matrix around the fixed point because the evaluation of nontrivial averages such as $\langle \exp(-V) \rangle_0$ is needed. However, if we recall the Monte Carlo scheme in statistical physics discussed Chapter 10, these averages should not be very difficult to evaluate through the Metropolis algorithm. A scheme introduced by Swendsen (1979) was devised for such a purpose. Let us still take the Ising model as the illustrative example. The renormalization transform is given by

$$\mathcal{H}^{(n+1)} = -J^{(n+1)} \sum_{\langle ij \rangle} s_i^{(n+1)} s_j^{(n+1)} - B^{(n+1)} \sum_i s_i^{(n+1)}, \tag{12.56}$$

up to a constant term. We will label the Hamiltonian by a vector $\mathbf{H} = (H_1, H_2) = (B, J)$, as we did in the preceding section. Then we have the renormalization matrix

$$A_{ij} = \frac{\partial H_i^{(n+1)}}{\partial H_j^{(n)}}. \tag{12.57}$$

The Hamiltonian can also be written in terms of the moments of the spin variables. For example, we can write

$$\mathcal{H} = \sum_{i=1}^{m} H_i M_i, \tag{12.58}$$

with, for example, $M_1 = \sum_j s_j$, $M_2 = \sum_{\langle jk \rangle} s_j s_k$, and so on. Now we can relate the renormalization transform matrix to the thermal averages of the moments through

$$\frac{\partial \left\langle M_i^{(n+1)} \right\rangle}{\partial H_j^{(n)}} = \sum_k \frac{\partial \left\langle M_i^{(n+1)} \right\rangle}{\partial H_k^{(n+1)}} \frac{\partial H_k^{(n+1)}}{\partial H_j^{(n)}}, \tag{12.59}$$

which is equivalent to

$$\mathbf{F} = \mathbf{GA}, \tag{12.60}$$

with

$$F_{ij} = \frac{\partial \left\langle M_i^{(n+1)} \right\rangle}{\partial H_j^{(n)}} \tag{12.61}$$

and

$$G_{ij} = \frac{\partial \left\langle M_i^{(n+1)} \right\rangle}{\partial H_j^{(n+1)}}. \tag{12.62}$$

We can express F_{ij} and G_{ij} in terms of the averages of M_i and $M_i M_j$ from the relation between H_i and M_i in the Hamiltonian and the definition of the thermal average, for example,

$$\langle M_i \rangle = \frac{\sum_{\{s_i\}} M_i e^{-\sum_j H_j M_j}}{\sum_{\{s_i\}} e^{-\sum_j H_j M_j}}. \tag{12.63}$$

If we take the derivative with respect to H_j in the above expression, we have,

$$F_{ij} = \left\langle M_i^{(n+1)} M_j^{(n)} \right\rangle - \left\langle M_i^{(n+1)} \right\rangle \left\langle M_j^{(n)} \right\rangle, \tag{12.64}$$

$$G_{ij} = \left\langle M_i^{(n+1)} M_j^{(n+1)} \right\rangle - \left\langle M_i^{(n+1)} \rangle \langle M_j^{(n+1)} \right\rangle. \tag{12.65}$$

The Swendsen scheme evaluates the matrices **F** and **G** in each renormalization transform step, and then the matrix **A** is obtained through Eq. (12.60), with **F** multiplied by the inverse of **G**. As we discussed in the preceding section, we can now calculate the relevant critical exponents from the eigenvalues of the matrix **A**. For more details, see Swendsen (1979).

12.5 Crossover: the Kondo problem

The Kondo problem has been one of the most interesting problems in the history of theoretical physics. The full solution of the problem, first accomplished by Wilson numerically through the renormalization group, contains both mathematical elegance and physical insight. The problem was first noticed in the experimental observation of the resistivity of simple metals such as copper, embedded with very dilute magnetic impurities, for example, chromium. The resistivity first decreases as the temperature is lowered, which is common in all nonmagnetic impurity scattering cases and well understood. However, at very low temperatures, typically on the order of 1 K, the resistivity starts to increase and eventually saturates at zero temperature. Kondo argued that this was due to the spin-flip scattering of the electron and magnetic moment of the impurity and came up with a very simple Hamiltonian for single-impurity scattering:

$$\mathcal{H} = \sum_{\mathbf{k}\sigma} \varepsilon_{\mathbf{k}} c^{\dagger}_{\mathbf{k}\sigma} c_{\mathbf{k}\sigma} + J\mathbf{S} \cdot \sum_{\mathbf{k}\sigma,\, \mathbf{q}\mu} c^{\dagger}_{\mathbf{k}\sigma} \mathbf{s}_{\sigma\mu} c_{\mathbf{q}\mu}, \tag{12.66}$$

where $c^{\dagger}_{\mathbf{k}\sigma}$ and $c_{\mathbf{k}\sigma}$ are creation and annihilation operators for the electrons, $\varepsilon_{\mathbf{k}}$ is the band energy dispersion, $\mathbf{s}_{\sigma\mu}$ are Pauli matrices for the electron spins, $\mathbf{S}$ is the spin operator for the impurity, and J is the spin coupling constant. For ferromagnetic coupling, that is, $J < 0$, perturbation theory provides a convergent solution of this model. The typical electron and magnetic impurity system, however, has antiferromagnetic coupling. Kondo used the Hamiltonian in Eq. (12.66) to calculate the resistivity of the electrons due to impurity scattering with a perturbation method. After summing up a specific class of infinite terms, Kondo found that the series diverges logarithmically when the temperature approaches a characteristic temperature of the system, now known as the Kondo temperature T_{K}, which is typically on the order of 1 K. A significant amount of theoretical effort followed the discovery of the problem by Kondo, and this effort was summarized in Kondo (1969). And it was clear then that a full solution of the problem was not going to be straightforward.

It was Wilson who devised the numerical renormalization method and solved the Kondo problem completely. Even though the problem was later solved exactly using an analytic method, it is still fair to say that Wilson's solution captured all the relevant physics of the problem (the process now known as the crossover phenomenon); that is, the system changes its behavior gradually from the high-temperature or the weak-coupling state to the low-temperature or the strong-coupling state without a phase transition.

Wilson made several modifications to the original Hamiltonian in order to solve it numerically with his renormalization method. First, the free electron energy band is assumed to be linear, that is, that $\varepsilon_k \propto k$. This modification of the Hamiltonian does not significantly change the physics of the model if the relevant energy scale is much smaller that the energy bandwidth. After a proper choice of the units and the Fermi level, the modified Kondo Hamiltonian is given by

$$\mathcal{H} = \int_{-1}^{1} c_k^\dagger c_k \, dk + J A^\dagger \mathbf{s} A \cdot \mathbf{S}, \tag{12.67}$$

where **s** and **S** are the Pauli matrices for an electron and for the impurity, respectively. The operators A and $A^\dagger$ are the collective operators defined by

$$A = \int_{-1}^{1} c_k \, dk. \tag{12.68}$$

Note that the spin indices have been suppressed in the above expressions for convenience of notation. Both c_k and A have spin indices and when the operators appear in pairs, the spin indices are all summed. For example,

$$A^\dagger \mathbf{s} A = \sum_{\sigma\mu} A_\sigma^\dagger \mathbf{s}_{\sigma\mu} A_\mu. \tag{12.69}$$

Then the momentum space is discretized with a logarithmic decrease of the space intervals toward the center of the band, that is, the Fermi level, with the lattice points at

$$k_m = \frac{\pm 1}{\Lambda^m}, \tag{12.70}$$

for $m = 0, 1, \ldots, \infty$, with $\Lambda > 1$. When $\Lambda \to 1$, the space approaches the continuous limit. We can then construct a set of orthogonal basis states defined in each interval $\Lambda^{-(m+1)} < k < \Lambda^{-m}$ as

$$\phi_{ml}(k) = \frac{\Lambda^{(m+1)/2}}{\sqrt{\Lambda - 1}} e^{i\omega_m k l}, \tag{12.71}$$

for $l = 0, 1, \ldots, \infty$, with

$$\omega_m = \frac{2\pi \Lambda^{m+1}}{\Lambda - 1}. \tag{12.72}$$

The state in Eq. (12.71) is nonzero only in the interval defined. Because the Fermi level is at the center of the band, $\phi_{ml}(k)$ and $\phi_{ml}(-k)$ for $k \geq 0$ form a complete basis for the whole momentum space. The creation and annihilation operators of electrons can then be expressed in terms of these discrete states as

$$a_k = \sum_{ml} [c_{ml} \phi_{ml}(k) + d_{ml} \phi_{ml}(-k)], \tag{12.73}$$

where c_{ml} and d_{ml} satisfy the fermion anticommutative relation, for example,

$$[c_{kl}, c_{mn}^\dagger]_+ = \delta_{km} \delta_{ln}, \tag{12.74}$$

and they can be interpreted as different fermion operators. If we only keep the $l = 0$ states in the above expansion, the Kondo Hamiltonian is further simplified to

$$\mathcal{H} = \varepsilon_0 \sum_m \frac{1}{\Lambda^m}(c_m^\dagger c_m - d_m^\dagger d_m) + JA^\dagger \mathbf{s} A \cdot \mathbf{S}, \tag{12.75}$$

where $c_m = c_{m0}$, $d_m = d_{m0}$, and $\varepsilon_0 = (1 + \Lambda^{-1})/2$. A transform can be made so that the Hamiltonian can be described by just one set of parameters, given as

$$\mathcal{H} = \sum_{k=0}^{\infty} \varepsilon_k (f_k^\dagger f_{k+1} + f_{k+1}^\dagger f_k) + \tilde{J} f_0^+ \mathbf{s} f_0 \cdot \mathbf{S}, \tag{12.76}$$

where f_0 is defined directly from A as

$$f_0 = \frac{1}{\sqrt{2}} A, \tag{12.77}$$

and f_k for $k > 0$ are given from an orthogonal transform of c_m and d_m with

$$f_k = \sum_m (u_{km} c_m + v_{km} d_m). \tag{12.78}$$

After rescaling, we have $\varepsilon_k = 1/\Lambda^{k/2}$ and $\tilde{J} = 4J\Lambda/(\Lambda + 1)$. This rescaling has some effects on terms with small values of k but not on terms with large values of k. We can obtain all the transform coefficients u_{km} and v_{km} and show that f_k satisfies the fermion anticommutative relation

$$[f_k, f_l^\dagger]_+ = \delta_{kl} \tag{12.79}$$

from the properties of c_{kl} and d_{kl} as well as the orthogonal transform. For more details of the approximation and transform, see Wilson (1975).

Before we describe Wilson's solution of the above Hamiltonian, a technical detail is worth mentioning. Assume that we want to obtain the spectrum of the Hamiltonian

$$\mathcal{H} = \mathcal{H}_0 + \mathcal{H}_1 + \mathcal{H}_2 + \cdots, \tag{12.80}$$

where $\mathcal{H}_0$ has a degenerate ground state and the elements in $\mathcal{H}_n$ are several orders smaller than the elements in $\mathcal{H}_{n-1}$. In order to have an accurate structure of the eigenvalue spectrum, one cannot simply diagonalize $\mathcal{H}$ with all the elements in $\mathcal{H}_n$ with $n > 0$ substituted. The reason is that the numerical rounding error will kill the detailed structure of the spectrum due to the smallness of $\mathcal{H}_n$ for $n > 0$. A better way, perhaps the only way, to maintain the accuracy in the spectrum is to diagonalize $\mathcal{H}_0$ first and then treat $\mathcal{H}_n$ as perturbations term by term. Note that the accuracy we are talking about here is not the absolute values of the eigenvalues but rather the structure, that is, the correct splitting of the energy levels or the hierarchical structure of the spectrum. The structure of the energy spectrum determines the properties of the system.

The Hamiltonian of Eq. (12.76) is ready to be formulated in a recursive form that defines the renormalization transform. Let us define a Hamiltonian with k

truncated at $k = n - 1$:

$$\mathcal{H}^{(n)} = \Lambda^{(n-1)/2} \sum_{k=0}^{n-1} \varepsilon_k (f_k^\dagger f_{k+1} + f_{k+1}^\dagger f_k) + \tilde{J} f_0^\dagger \mathbf{s} f_0 \cdot \mathbf{S}, \tag{12.81}$$

where the rescaling factor $\Lambda^{(n-1)/2}$ is used to make the smallest term in the Hamiltonian on the order of 1. The results of the original Hamiltonian can be recovered by dividing the eigenvalues by $\Lambda^{(n-1)/2}$ after they are obtained. The recursive relation or the renormalization transform is then formulated as

$$\mathcal{H}^{(n+1)} = T[\mathcal{H}^{(n)}] = \frac{1}{\sqrt{\Lambda}} \mathcal{H}^{(n)} + f_n^\dagger f_{n+1} + f_{n+1}^\dagger f_n - E_0^{(n+1)}, \tag{12.82}$$

where $E_0^{(n+1)}$ is the ground-state energy of the Hamiltonian $\mathcal{H}^{(n+1)}$. The Hamiltonian can now be diagonalized with a combination of numerical diagonalization of the first part of the Hamiltonian and then degenerate perturbations of other terms. The number of states increases exponentially with n. We have to truncate the number of states used in the recursions at some reasonable number so that the problem can be dealt with on a computer.

Wilson discovered that no matter how small the original coupling constant $J > 0$ is, the behavior of the system always crosses over to that of $J = \infty$ after many steps of recursion. In physical terms, this means that no matter how small the coupling between an electron and the impurity, if it is antiferromagnetic, when the temperature approaches zero, the system eventually moves to the strong-coupling limit: that is, the electron spins screen the impurity spin completely at a temperature of zero. The temperature at which the crossover happens is the Kondo temperature, which is a function of the original coupling constant J.

We are not going to have more discussion here of the methods of evaluation of physical quantities such as the magnetic susceptibility or specific heat of the impurity. Interested readers can find a full discussion of the Kondo problem in Hewson (1993).

12.6 Quantum lattice renormalization

After Wilson's celebrated work on the numerical renormalization study of the Kondo problem in the early 1970s, a series of attempts were made to generalize the idea in the study of quantum lattice models. These attempts turned out not as successful as the study of the Kondo problem. However, these studies did provide some qualitative understanding of these systems and have formed the basis for further development, especially the density matrix renormalization of White (1992; 1993).

In this section, we discuss how to generalize the Wilson method to quantum lattice models. What we need to find is the transform

$$\mathcal{H}^{(n+1)} = T[\mathcal{H}^{(n)}], \tag{12.83}$$

which can accurately describe the energy structure of the system, at least for low-lying excited states. We will consider an infinite chain system and divide it first into small blocks. The renormalization transform then combines two nearest blocks into one new block. We focus on two blocks and assume that the left-hand block has N_{L} independent states and the right-hand block has N_{R} independent states, so the total number of states of the two-block system is $N = N_{\mathrm{L}} \times N_{\mathrm{R}}$. The Hamiltonian can be written symbolically as

$$\mathcal{H}^{(n)} = \mathcal{H}_{\mathrm{L}}^{(n)} + \mathcal{H}_{\mathrm{R}}^{(n)} + V_{\mathrm{LR}}^{(n)}, \tag{12.84}$$

where $\mathcal{H}_{\mathrm{L,\,R}}^{(n)}$ is the intrablock Hamiltonian and $V_{\mathrm{LR}}^{(n)}$ is the coupling between the blocks. Let us take $|l\rangle$ with $l = 1, 2, \ldots, N_{\mathrm{L}}$ and $|r\rangle$ with $r = 1, 2, \ldots, N_{\mathrm{R}}$ as the eigenstates of the left-hand block and the right-hand block, respectively, and the direct product $|i\rangle = |l\rangle \otimes |r\rangle$, where $i = 1, 2, \ldots, N$ is the basis functions for constructing the eigenstates of $\mathcal{H}^{(n)}$, given as

$$|k\rangle = \sum_{i=1}^{N} a_{ki}|i\rangle, \tag{12.85}$$

where $k = 1, 2, \ldots, N$, and then we have

$$\mathcal{H}^{(n)}|k\rangle = \mathcal{E}_k|k\rangle. \tag{12.86}$$

The direct product of two vectors $|i\rangle = |l\rangle \otimes |r\rangle$ will be illustrated later in this section in an example. The renormalization transform is then performed with $K \leq N$ states selected from $|k\rangle$ with

$$\mathbf{H}^{(n+1)} = \mathbf{A}\mathbf{H}^{(n)}\mathbf{A}^{\dagger}, \tag{12.87}$$

where $\mathbf{A}$ is a $K \times N$ matrix constructed from

$$A_{ki} = a_{ki} \tag{12.88}$$

and the matrix representation of $\mathcal{H}^{(n)}$ on the right-hand side of Eq. (12.87) is in the original direct product state $|i\rangle$, given as

$$\langle i|\mathcal{H}^{(n)}|i'\rangle = \langle l|\mathcal{H}_{\mathrm{L}}|l'\rangle \otimes \mathbf{I}_{\mathrm{R}} + \mathbf{I}_{\mathrm{L}} \otimes \langle r|\mathcal{H}_{\mathrm{R}}|r'\rangle + \langle i|V_{\mathrm{LR}}|i'\rangle, \tag{12.89}$$

where $\mathbf{I}_{\mathrm{R}}$ and $\mathbf{I}_{\mathrm{L}}$ are unit matrices with dimensions $N_{\mathrm{R}} \times N_{\mathrm{R}}$ and $N_{\mathrm{L}} \times N_{\mathrm{L}}$, respectively, and $\otimes$ has the same meaning as the direct product of two irreducible matrix representations of a group (Tinkham, 2003).

One aspect requiring special care is the matrix representation of the interaction term between two blocks. For example, if the interaction is of the Ising type with

$$V_{\mathrm{LR}} = J s_{\mathrm{r}}^{x} s_{\mathrm{l}}^{x}, \tag{12.90}$$

we have

$$\langle i|V_{\mathrm{LR}}|i'\rangle = J\langle l|s_{\mathrm{r}}^{x(n)}|l'\rangle \otimes \left\langle r|s_{\mathrm{l}}^{x(n)}|r'\right\rangle, \tag{12.91}$$

where s_{r}^x and s_{l}^x are the spin operators at the right-hand and left-hand ends of the blocks, respectively.

In fact, all other variables are transformed in a way similar to the transformation of the Hamiltonian, particularly the operators at the boundaries, which are needed to construct the next iteration of the Hamiltonian. For example, the new spin operator for the right-hand boundary is given by

$$s_{\mathrm{r}}^{x(n+1)} = \mathbf{A} s_{\mathrm{r}}^{x(n)} \mathbf{A}^{\dagger}. \tag{12.92}$$

Note that we also need to expand the dimensionality of $s_{\mathrm{r,l}}^{x(n)}$ for the transform by a direct product with a unit matrix, for example,

$$\langle i \left| s_{\mathrm{r}}^{x(n)} \right| i' \rangle = \mathbf{I}_{\mathrm{L}} \otimes \langle r \left| s_{\mathrm{r}}^{x(n)} \right| r' \rangle; \tag{12.93}$$

$$\left\langle i |s_{\mathrm{l}}^{x(n)}| i' \right\rangle = \left\langle l |s_{\mathrm{l}}^{x(n)}| l' \right\rangle \otimes \mathbf{I}_{\mathrm{R}}. \tag{12.94}$$

There is one important aspect we have not specified, that is, the criteria for selecting the K states out of a total number of N states to construct the renormalization transform. Traditionally, the states with lowest energies were taken for the renormalization transform. However, there is a difference between the lattice Hamiltonian and the Kondo Hamiltonian worked out by Wilson. In the Kondo problem, every term added during the transformation is much smaller than the original elements, so we would not expect, for example, any level crossing. The lower states contribute more to the true ground state. However, in the lattice case, there are no such small elements. The terms added in are as large as other elements. A problem arises when we perform the transformation by keeping the few (for example, 100) lowest states because the higher levels still contribute to the true ground state or lower excited states at the infinite limit. In the next section, we will discuss the reduced density matrix formulation of the renormalization transform introduced by White (1992; 1993), which resolves the problem of level crossing during the transformation, at least for one-dimensional systems.

In order to have a better understanding of the scheme outlined above, let us apply it to an extremely simple model, the spin-$\frac{1}{2}$ quantum Ising chain in a transverse magnetic field, given as

$$\mathcal{H} = J \sum_{i=-\infty}^{\infty} s_i^x s_{i+1}^x - B \sum_{i=-\infty}^{\infty} s_i^z, \tag{12.95}$$

where s^x and s^z can be expressed in the Pauli matrices:

$$s^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \quad s^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{12.96}$$

if we take $\hbar/2 = 1$. This model was studied by Jullien *et al.* (1978) in great detail. What we would like to obtain is a transform

$$\mathcal{H}^{(n+1)} = T[\mathcal{H}^{(n)}]. \tag{12.97}$$

In order to simplify our discussion, we will only take one site in each block and then combine two blocks into a new block during the transformation. We will also take the two states with lowest eigenvalues for the transform, which means that $N_{\mathrm{L}} = N_{\mathrm{R}} = K = 2$ and $N = N_{\mathrm{L}} + N_{\mathrm{R}} = 4$. So the states in each block $|r\rangle$ or $|l\rangle$ are given by

$$|1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad |2\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{12.98}$$

Then the direct product $|i\rangle = |l\rangle \otimes |r\rangle$ generates four basis functions

$$|1\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \quad |2\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}; \quad |3\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}; \quad |4\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \tag{12.99}$$

which are obtained by taking the product of first element of the first vector and the second vector to form the first two elements and then the second element of the first vector and the second vector to form the the next two elements. The matrix direct products are performed in exactly the same manner. In general, we want to obtain the direct product $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$, where $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{B}$ is an $m \times m$ matrix. Then $\mathbf{C}$ is an $nm \times nm$ matrix, which has $n \times n$ blocks with $m \times m$ elements in each block constructed from the corresponding element of $\mathbf{A}$ multiplied by the the second matrix $\mathbf{B}$. Based on this rule of the direct product, we can obtain all the terms in a Hamiltonian of two blocks. And if we put all these terms together, we have

$$\mathbf{H}^{(n)} = \begin{pmatrix} -2B & 0 & 0 & J \\ 0 & 0 & J & 0 \\ 0 & J & 0 & 0 \\ J & 0 & 0 & 2B \end{pmatrix}, \tag{12.100}$$

which can be diagonalized by the following four states:

$$\begin{aligned} |1\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ -1 \\ 1 \\ 0 \end{pmatrix}; \quad |2\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}; \\ |3\rangle &= \frac{1}{\sqrt{1+\alpha^2}} \begin{pmatrix} -\alpha \\ 0 \\ 0 \\ 1 \end{pmatrix}; |4\rangle = \frac{1}{\sqrt{1+\beta^2}} \begin{pmatrix} -\beta \\ 1 \\ 1 \\ 0 \end{pmatrix}. \end{aligned} \tag{12.101}$$

where α and β are given by

$$\alpha = \frac{2B + \sqrt{4B^2 + J^2}}{J}; \quad \beta = \frac{2B - \sqrt{4B^2 + J^2}}{J}. \tag{12.102}$$

The corresponding eigenvalues are

$$\mathcal{E}_k = -J, \ J, \ -\sqrt{4B^2 + J^2}, \ \sqrt{4B^2 + J^2}. \tag{12.103}$$

Because the first and third states have lower energy for $J > 0$, the transform matrix is then given as

$$\mathbf{A} = \begin{pmatrix} 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ -\alpha/\sqrt{1+\alpha^2} & 0 & 0 & 1/\sqrt{1+\alpha^2} \end{pmatrix}, \tag{12.104}$$

which can be used to construct the new block Hamiltonian

$$\mathbf{H}^{(n+1)} = \mathbf{A}\mathbf{H}^{(n)}\mathbf{A}^{\dagger} = \begin{pmatrix} -J & 0 \\ 0 & -\gamma \end{pmatrix}, \tag{12.105}$$

with $\gamma = \sqrt{4B^2 + J^2}$. Similarly, we can construct new spin operators at the boundaries, for example,

$$s_{\mathrm{r}}^{x(n+1)} = \mathbf{A} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \mathbf{A}^{\dagger} = \frac{\alpha + 1}{\sqrt{2(1+\alpha^2)}} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \tag{12.106}$$

and $s_{\mathrm{l}}^{x(n+1)} = -s_{\mathrm{r}}^{x(n+1)}$. The new interaction term is then given by

$$V_{\mathrm{LR}}^{(n+1)} = J s_{\mathrm{r}}^{x(n+1)} \otimes s_{\mathrm{l}}^{x(n+1)}. \tag{12.107}$$

The above scheme is repeated to convergence. As we pointed out at the beginning of the section, this procedure turns out to be unsuccessful in most systems. The problem lies in the facts that the addition of two blocks is not a very small perturbation to the single-block Hamiltonian and that the overlap of the higher-energy states with the lower-energy states in the infinite system is very significant.

12.7 Density matrix renormalization

Significant progress in the numerical renormalization study of highly correlated systems has been made by White (1992; 1993). The basic idea comes from the observation that the reduced density matrix constructed from a set of states of a specific segment of an infinite system should have the largest eigenvalues if we want this set of states to have the largest overlap with the lower-lying states of the system. Here we will just give a brief introduction to the scheme: interested readers should consult reviews on the subject, for example, White (1998), Shibata (2003), and Schollwöck (2005).

The density matrix is defined for a system that is in contact with the environment. Assume that the system is described by a set of orthonormal and complete states $|l\rangle$ for $l = 1, 2, \ldots, m$ and the environment by another set of orthonormal and complete states $|r\rangle$ for $r = 1, 2, \ldots, n$. Then an arbitrary state including the system and its environment can be expressed as

$$|\psi\rangle = \sum_{l,r=1}^{m,n} c_{lr} |l\rangle|r\rangle, \tag{12.108}$$

where $|l\rangle|r\rangle$ is used as a two-index notation, instead of the direct product $|l\rangle \otimes |r\rangle$ used earlier. This is an easier way of keeping track of the indices. The expectation value of an operator A of the system under the state $|\psi\rangle$ can then be written as

$$\langle A\rangle = \langle\psi|A|\psi\rangle = \sum_{l,l',r} c^*_{l'r} c_{lr} \langle l'|A|l\rangle = \operatorname{Tr} \rho A, \tag{12.109}$$

where the density matrix ρ of the system is defined from

$$\rho_{ij} = \sum_{r=1}^{n} c^*_{jr} c_{ir} \tag{12.110}$$

for the above expectation value. Note that we have assumed that A is not coupled to the environment and the index r for the environment is therefore in the diagonal form and summed up completely.

We can easily show that ρ is Hermitian and can therefore be diagonalized and written in the diagonal form of its eigenstates $|\alpha\rangle$ as

$$\rho = \sum_{\alpha=1}^{m} w_\alpha |\alpha\rangle\langle\alpha|, \tag{12.111}$$

where w_α can be interpreted as the probability that the system is in the state $|\alpha\rangle$ under the given system and environment. Because $|\alpha\rangle$ is a state of the system, it can be written as

$$|\alpha\rangle = \sum_{l=1}^{m} u_{\alpha l}|l\rangle. \tag{12.112}$$

Here $u_{\alpha l}$ can be viewed as the elements of a unitary matrix because $|\alpha\rangle$ also form a complete, orthonormal basis set, and the relation between $|\alpha\rangle$ and $|l\rangle$ is a canonical transform, or a coordinate rotation in Hilbert space. If we normalize the wavefunction, we have

$$\langle\psi|\psi\rangle = \operatorname{Tr}\rho = \sum_{\alpha=1}^{m} w_\alpha = 1, \tag{12.113}$$

which means that the states with higher w_α then contribute more than other states when an average of a physical quantity $\operatorname{Tr}\rho A$ is carried out. So a good approximation can be found if we use the reduced matrix

$$\rho \simeq \sum_{\alpha=1}^{k} w_\alpha |\alpha\rangle\langle\alpha| \tag{12.114}$$

for $k < m$ under the choice of order $w_1 \geq w_2 \cdots \geq w_k \cdots \geq w_m \geq 0$. This concept of using the reduced density matrix in the evaluation of a physical quantity during a block transformation forms the basic idea of density matrix renormalization (White, 1992). The states selected to construct the new blocks are the eigenstates of the density matrix with the largest eigenvalues, because they have the highest probabilities for the system to be in under the given environment. The system can then be viewed as a segment of an infinite system.

Because of the fast increase in the number of states in the system and in its environment when the size of either grows, the selection of the system and its environment becomes very important in order to have a practical and accurate scheme. As we discussed in the preceding section, a new block is usually constructed from two identical blocks. Numerical results show that it is more efficient in most cases to construct the new block by adding one more site to the old block. Because the system is growing only at one end if one site is added to the old block, the result corresponds to the condition of an open end. If a periodic boundary condition is imposed on the system, the added site(s) should also be in contact with the other end, as the size of a circle is increased by introducing a new segment into its circumference. For an infinite chain, imposing a periodic condition for the renormalization transform requires many more states to be maintained in order to have the same accuracy as for the the system with an open end.

There are two important criteria in the selection of the environment. It has to be convenient for constructing the new block Hamiltonian, and it has to be the best representation of the rest of the actual physical system within the capacity of current computers. We now go into a little more detail about the algorithms that are currently used for studying most one-dimensional systems. We first select a block with a fixed number of sites, denoted by $B_{\mathrm{L}}^{(k)}$ at the nth iteration of the renormalization transform. Then we add a single site to the right-hand side of the block to form the new block denoted by $B_{\mathrm{L}}^{(k+1)} = B_{\mathrm{L}}^{(k)}\circ$, where the circle $\circ$ means the site added to the original block $B_{\mathrm{L}}^{(k)}$. The environment is selected as the reflection of $B_{\mathrm{L}}^{(k)}\circ$, denoted as $\circ B_{\mathrm{R}}^{(k)}$. The system $B_{\mathrm{L}}^{(k)}\circ$ and the environment $\circ B_{\mathrm{R}}^{(k)}$ together form a superblock $B_{\mathrm{L}}^{(k)} \circ \circ B_{\mathrm{R}}^{(k)}$. We can solve the Hamiltonian of the superblock and construct the reduced density matrix for the new system $B_{\mathrm{L}}^{(k+1)} = B_{\mathrm{L}}^{(k)}\circ$ from the eigenstates of the density matrix with largest eigenvalues. Then we can add one more site to the new system to repeat the process. We can start the renormalization scheme with four sites as the first superblock $B_{\mathrm{L}}^{(1)} \circ \circ B_{\mathrm{R}}^{(1)}$. So each block, $B_{\mathrm{L}}^{(1)}$, or $B_{\mathrm{R}}^{(1)}$, has only one site. In general the matrix of the Hamiltonian of a superblock is usually very sparse and can be solved with the Lanczos method discussed in Chapter 5.

The above renormalization transform scheme is termed the algorithm for an infinite system because the size of the system grows with the iteration. The total number of basis functions increases exponentially with the number of sites involved. The restriction imposed by the reduced density matrix selection of the basis states therefore introduces a certain error into the wavefunction and physical quantities evaluated. To minimize this artificial uncertainty, we can also perform a finite size renormalization in each stage of the the renormalization for an infinite system, or for a finite system with a fixed size. We start first with a symmetric superblock $B_{\mathrm{L}}^{(k)} \circ \circ B_{\mathrm{R}}^{(k)}$. Then we formulate the new system as $B_{\mathrm{L}}^{(k+1)} = B_{\mathrm{L}}^{(k)}\circ$ but the environment $B_{\mathrm{R}}^{(k+1)} = \circ B_{\mathrm{R}}^{(k-1)}$, where $B_{\mathrm{R}}^{(k-1)}$ is the block with one site less than $B_{\mathrm{R}}^{(k)}$, or the previous environment in the infinite algorithm. By sweeping through the system this way, we improve the wavefunction and physical quantities evaluated for the system of a given size. In fact, if we are studying the ground state, the whole process is equivalent to a variational procedure with a controllable uncertainty.

The algorithms for both of the infinite and finite systems have been applied to many different physical systems in the last decade. We will not go into those aspects here. Interested readers can found detailed discussions in the the reviews by White (1998), Shibata (2003), and Schollwöck (2005).

Exercises

12.1 From the scaling relations discussed in the text, if two of the exponents are given, we can find the rest. For the two-dimensional Ising model, the exact results are available. If we start with $\alpha = 0$ and $\beta = 1/8$, show that $\gamma = 7/4$, $\delta = 15$, $\eta = 1/4$, and $\nu = 1$.

12.2 As discussed in the text, the approximation, up to the first order, of the coupling between two nearest neighboring spin blocks can be improved by including more terms in the expansion. If we rewrite the Hamiltonian in the form of the nearest neighbor interaction and another term of next nearest neighbor interaction (zero in the zeroth-order iteration), we can incorporate the second-order term in the renormalization transform. Derive the renormalization transform matrix for the case in which first-order and second-order terms are included in the expansion and calculate the critical exponents for the triangular Ising model. Are there any improvements over the first-order approximation?

12.3 One can also improve the calculation outlined in the text for the triangular Ising model by choosing larger blocks. Use hexagonal blocks with seven sites as the renormalization transform units and evaluate the exponents with expansion up to first order. Are there any improvements over the calculations of the triangular blocks?

12.4 Develop a program with the Swendsen Monte Carlo renormalization algorithm for the two-dimensional Ising model on a square lattice. Take the 2×2 block as the renormalization transform unit for a system of 40×40 sites and apply the majority rule for the block spin.

12.5 One of Wilson's very interesting observations is that when one has a Hamiltonian matrix with multiple energy scales, the only way to obtain the accurate structure of the energy levels is to perform degenerate perturbations level by level. Assume that we have the following Hamiltonian:

$$\mathcal{H} = H^{(0)} + H^{(1)} + H^{(2)},$$

with the zeroth order given by

$$H_0 = \begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix}$$

and the first-order and second-order terms given by

$$H_{ij}^{(1)} = \begin{cases} 0.01 & \text{for } i \neq j, \\ 0 & \text{for } i = j, \end{cases}$$

and

$$H_{ij}^{(2)} = \begin{cases} 0.0001 & \text{for } i \neq j, \\ 0 & \text{for } i = j. \end{cases}$$

Find the energy level structure of the Hamiltonian through degenerate perturbation. Keep only the ground state at each stage of the calculations.

12.6 Use the renormalization group to study the one-dimensional ferromagnetic quantum Ising model. Discuss the fixed point and the phase diagram of the system.

12.7 Develop a program using the density matrix renormalization scheme to study the one-dimensional ferromagnetic quantum Ising model. Start the scheme with a four-site superblock.

12.8 Construct the density matrix renormalization group to study of the one-dimensional anisotropic spin-$\frac{1}{2}$ Heisenberg model

$$\mathcal{H} = -J \sum_{\langle ij \rangle}^{L} \left(\lambda s_i^z s_j^z + s_i^x s_j^x + s_i^y s_j^y \right),$$

where $J > 0$. Explore the phase diagram for different λ. What happens if $J < 0$?

References

Abbas, A. (2004). *Grid Computing: A Practical Guide to Technology and Applications* (Hingham, Massachusetts: Charles River Media).

Abrahams, E., Anderson, P.W., Licciardello, D.C., and Ramakrishnan, T.V. (1979). Scaling theory of localization: absence of quantum diffusion in two dimensions, *Physical Review Letters* **42**, 673–6.

Aitken, A.C. (1932). An interpolation by iteration of proportional parts, without the use of differences, *Proceedings of Edinburgh Mathematical Society,* Series 2, **3**, 56–76.

Akulin, V.M., Bréchignac, C., and Sarfati, A. (1995). Quantum shell effect on dissociation energies, shapes, and thermal properties of metallic clusters from random matrix model, *Physical Review Letters* **75**, 220–3.

Allen, M.P. and Tildesley, D.J. (1987). *Computer Simulation of Liquids* (Oxford, UK: Clarendon).

Andersen, H.C. (1980). Molecular dynamics simulations at constant pressure and/or temperature, *Journal of Chemical Physics* **72**, 2384–93.

Anderson, J.B. (1975). A random walk simulation of the Schrödinger equation: H_3^+, *Journal of Chemical Physics* **63**, 1499–503.

Anderson, J.B. (1976). 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, *Journal of Chemical Physics* **65**, 4121–7.

Anderson, P.W. (1987). The resonating valence bond state in La_2CuO_4 and superconductivity, *Science* **235**, 1196–8.

Anderson, S.L. (1990). Random number generators on vector supercomputers and other advanced architectures, *SIAM Review* **32**, 221–51.

Attaccalite, C., Moroni, S., Gori-Giorgi, P., and Bachelet, G.B. (2002). Correlation energy and spin polarization in the 2D electron gas, *Physical Review Letters* **88**, 256–601.

Bäck, T., Fogel, D., and Michalewicz, Z. (eds.) (2003). *Handbook of Evolutionary Computation* (Bristol, UK: Institute of Physics Publishing).

Baker, G.L. and Gollub, J.P. (1996). *Chaotic Dynamics: An Introduction* (Cambridge, UK: Cambridge University Press).

Baletto, F., Rapallo, A., Rossi, G., and Ferrando, R. (2004). Dynamical effects in the formation of magic cluster structures, *Physical Review B* **69**, 235–421.

Barker, J.A. (1979). A quantum-statistical Monte Carlo method; path integrals with boundary conditions, *Journal of Chemical Physics* **70**, 2914–18.

Berendsen, H.J.C., Postma, J.P.M., van Gunsteren, W.F., DiNola, A., and Haak, J.R. (1984). Molecular dynamics with coupling to an external bath, *Journal of Chemical Physics* **81**, 3684–90.

Binder, K. (ed.) (1986). *Monte Carlo Methods in Statistical Physics* (Berlin: Springer-Verlag).

Binder, K. (ed.) (1987). *Monte Carlo Methods in Statistical Physics*, Volume II (Berlin: Springer-Verlag).

Binder, K. (ed.) (1992). *Monte Carlo Method in Condensed Matter Physics* (Berlin: Springer-Verlag).

Binder, K. and Heermann, D.W. (1988). *Monte Carlo Simulation in Statistical Physics, An Introduction* (Berlin: Springer-Verlag).

Bohigas, O. (1991). Random matrix theories and chaotic dynamics, in *Chaos and Quantum Physics*, eds. M.-J. Giannoni, A. Voros, and J. Zinn-Justin (Amsterdam: North-Holland), pp. 87–199.

Boisvert, R.F., Moreira, J., Philippsen, M., and Pozo, R. (2001). Java and numerical computing, *Computing in Science & Engineering* **3**(2), 18–24.

Boninsegni, M. and Ceperley, D.M. (1995). Path integral Monte Carlo simulation of isotopic liquid helium mixtures, *Physical Review Letters* **74**, 2288–91.

Bouchaud, J.P. and Lhuillier, C. (1987). A new variational description of liquid ^{3}He: the superfluid glass, *Europhysics Letters* **3**, 1273–80.

Brainerd, W.S., Goldberg, C.H., and Adams, J.C. (1996). *Programmer's Guide to Fortran 90* (New York: McGraw–Hill).

Bristeau, M.O., Glowinski, R., Mantel, B., Périaux, J., and Perrier, P. (1985). Numerical methods for incompressible and compressible Navier–Stokes problems, in *Finite Elements in Fluids*, Volume 6, eds. R.H. Gallagher, G. Carey, J.T. Oden, and O.C. Zienkiewicz (Chichester, New York: Wiley), pp. 1–40.

Brody, T.A., Flores, J., French, J.B., Mello, P.A., Pandey, A., and Wong, S.S.M. (1981). Random-matrix physics: Spectrum and strength fluctuations, *Reviews of Modern Physics* **53**, 383–479.

Broyden, C.G. (1970). The convergence of a class of double-rank minimization algorithms, Part I and Part II, *Journal of Institute of Mathematics and Its Applications* **6**, 76–90; 222–36.

Büchner, J., Dum, C.T., and Scholer, M. (eds.) (2003). *Space Plasma Simulation* (Berlin: Springer-Verlag).

Burks, A.R. and Burks, A.W. (1988). *The First Electronic Computer: The Atanasoff Story* (Ann Arbor, Michigan: University of Michigan).

Burnett, D.S. (1987). *Finite Element Analysis: From Concepts to Applications* (Reading, Massachusetts: Addison–Wesley).

Burrus, C.S. and Parks, T. W. (1985). *DFT/FFT and Convolution Algorithms* (New York: Wiley).

Car, R. and Parrinello, M. (1985). Unified approach for molecular dynamics and density-functional theory, *Physical Review Letters* **55**, 2471–4.

Ceperley, D.M. (1992). Path-integral calculations of normal liquid ^{3}He, *Physical Review Letters* **69**, 331–4.

Ceperley, D.M. (1995). Path integrals in theory of condensed helium, *Reviews of Modern Physics* **67**, 279–356.

Ceperley, D.M. and Alder, B.J. (1984). Quantum Monte Carlo for molecules: Green's function and nodal release, *Journal of Chemical Physics* **81**, 5833–44.

Ceperley, D.M. and Kalos, M.H. (1986). Quantum many-body problems, in *Monte Carlo Methods in Statistical Physics*, ed. K. Binder (Berlin: Springer-Verlag), pp. 145–94.

Ceperley, D.M. and Pollock, E.L. (1986). Path-integral computation of the low-temperature properties of liquid ^{4}He, *Physical Review Letters* **56**, 351–4.

Ceperley, D.M, Chester, G.V., and Kalos, M.H. (1977). Monte Carlo simulations of a many-fermion study, *Physical Review B* **16**, 3081–99.

Charbeneau, R.J. (2000). *Groundwater Hydraulics and Pollutant Transport* (Upper Saddle River, New Jersey: Prentice Hall).

Chen, H., Chen, S., and Matthaeus, W. (1992). Recovery of the Navier–Stokes equations using a lattice-gas Boltzmann method, *Physical Review A* **45**, R5339–42.

Chen, S. and Doolen, G.D. (1998). Lattice Boltzmann method for fluid flows, *Annual Review of Fluid Mechanics* **30**, 329–64.

Chui, C.K. (1992). *An Introduction to Wavelets* (San Diego, California: Academic).

Chui, C.K., Montefusco, L., and Puccio, L. (eds.) (1994). *Wavelets: Theory, Algorithms, and Applications* (San Diego, California: Academic).

Coello Coello, C.A., Van Veldhuizen, D.A., and Lamont, G.B. (2002). *Evolutionary Algorithms for Solving Multi-Objective Problems* (New York: Kluwer Academic).

Combes, J.M., Grossmann, A., and Tchanmitchian, P. (eds.) (1990). *Wavelets* (Berlin: Springer-Verlag).

Cook, R.D., Malkus, D.S., Plesha, M.E., and Witt, R.J. (2001). *Concepts and Applications of Finite Element Analysis* (New York: Wiley).

Cooley, J.W. and Tukey, J.W. (1965). An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation* **19**, 297–301.

Cooley, J.W., Lewis, P.A.W., and Welch, P.D. (1969). The fast Fourier transform and its applications, *IEEE Transactions on Education* **E-12**, 27–34.

Courant, R. and Hilbert, D. (1989). *Methods of Mathematical Physics,* Volume I and Volume II (New York: Wiley).

Cullum, J.K. and Willoughby, R.A. (1985). *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, Volume 1 and Volume 2 (Boston, Massachusetts: Birkhauser).

Dagotto, E. (1994). Correlated electrons in high temperature superconductors, *Reviews of Modern Physics* **66**, 763–840.

Dagotto, E. and Moreo, A. (1985). Improved Hamiltonian variational technique for lattice models, *Physical Review D* **31**, 865–70.

Daubechies, I. (1988). Orthonormal bases of compactly supported wavelets, *Communications on Pure and Applied Mathematics* **41**, 909–96.

Daubechies, I. (1992). *Ten Lectures on Wavelets* (Philadelphia: SIAM).

Davies, R. (1999). *Introductory Java for Scientists and Engineers* (Harlow, UK: Addison–Wesley).

Davis, P.J. and Polonsk, I. (1965). Numerical integration, differentiation and integration, in *Handbook of Mathematical Functions*, eds. M. Abramowitz and I.A. Stegun (New York: Dover), pp. 875–924.

Deaven, D.M. and Ho, K.M. (1995). Molecular geometry optimization with a genetic algorithm, *Physical Review Letters* **75**, 288–91.

Deb, K. (2001). *Multi-Objective Optimization using Evolutionary Algorithms* (Chichester, UK: Wiley).

Dennis, J.E. Jr. and Schnabel, R.B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, New Jersey: Prentice Hall).

Doolen, G.D. (ed.) (1991). *Lattice Gas Methods: Theory, Applications, and Hardware* (Cambridge, Massachusetts: MIT Press).

Edgar, S.L. (1992). *Fortran for the '90s* (New York: Freeman).

Ercolessi, F., Tosatti, E., and Parrinello, M. (1986). Au (100) surface reconstruction, *Physical Review Letters* **57**, 719–22.

Evans, D.J., Hoover, W.G., Failor, B.H., Moran, B., and Ladd, A.J.C. (1983). Nonequilibrium molecular dynamics via Gauss's principle of least constraint, *Physical Review A* **28**, 1016–21.

Evans, M.W. and Harlow, F.H. (1957). The particle-in-cell method for hydrodynamic calculations, *Los Alamos Scientific Laboratory Report No. LA-2139*.

Faddeev, D.K. and Faddeeva, W.N. (1963). *Computational Methods of Linear Algebra* (San Francisco, California: Freeman).

Fahy, S., Wang, X.W., and Louie, S.G. (1990). Variational quantum Monte Carlo nonlocal pseudopotential approach to solids: Formulation and application to diamond, graphite, and silicon, *Physical Review B* **42**, 3503–22.

Fletcher, R. (1970). A new approach to variable metric algorithms, *Computer Journal* **13**, 317–22.

Fogel, L.J. (1962). Autonomous automata, *Industrial Research* **4**, 14–9.

Fogel, L.J., Owens, A.J., and Walsh, M.J. (1966). *Artificial Intelligence through Simulated Evolution* (New York: Wiley).

Fortin, M. and Thomasset, F. (1983). Application to the Stokes and Navier–Stokes equations, in *Argumented Lagrangian Methods: Applications to the Numerical Solution of Boundary-Value Problems*, eds. M. Fortin and R. Glowinski (Amsterdam: North-Holland), pp. 47–95.

Foster, I. and Kesselman, C. (eds.) (2003). *The Grid 2: Blueprint for a New Computing Infrastructure* (San Francisco, California: Morgan Kaufmann).

Foufoula-Georgiou, E. and Kumar, P. (eds.) (1994). *Wavelets in Geophysics* (San Diego, California: Academic).

Frisch, U., d'Humières, D., Hasslacher, B., Lallemand, P., and Pomeau, Y. (1987). Lattice gas hydrodynamics in two and three dimensions, *Complex Systems* **1**, 649–701.

Frisch, U., Hasslacher, B., and Pomeau, Y. (1986). Lattice gas automata for the Navier–Stokes equation, *Physical Review Letters* **56**, 1505–8.

Gao, X.P.A., Mills, Jr. A.P., Ramirez, A.P., Pfeiffer, L.N., and West, K.W. (2002). Weak-localization-like temperature-dependent conductivity of a dilute two-dimensional hole gas in a parallel magnetic field, *Physical Review Letters* **89**, 016 801.

Gauss, C.F. (1866). Nachlass: Theoria interpolationis methodo nova tractata, in *Carl Friedrich Gauss, Werke, Band 3*, eds. E. Schering, F. Klein, M. Brendel, and L. Schlesinger (Göttingen: Königlichen Gesellschaft der Wissenschaften), pp. 265–303.

Gear, C.W. (1971). *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, New Jersey: Prentice–Hall).

Goldberg, D.E. (1989). *Genetic Algorithm in Search, Optimization, and Machine Learning* (New York: Addison–Wesley).

Goldberg, D.E. and Deb, K. (1991). A comparison of selection schemes used in genetic algorithms, *Foundations of Genetic Algorithms* **1**, 69–93.

Goldfarb, D. (1970). A family of variable metric methods derived by variational means, *Mathematics of Computation* **24**, 23–6.

Goldstine, H.H. (1977). *A History of Numerical Analysis from the 16th through the 19th Century* (New York: Springer-Verlag).

Gottwald, D., Likos, C.N., Kahl, G., and Löwen, H. (2004). Phase behavior of ionic microgels, *Physical Review Letters* **92**, 068 301.

Grigoryev, Yu.N., Vshivkov, V.A., and Fedoruk, M.P. (2002). *Numerical Particle-in-Cell Methods: Theory and Applications* (Leiden: VSP).

Grimmett, G. (1999). *Percolation* (Berlin: Springer-Verlag).

Gross, D.H.E. (2001). *Microcanonical Thermodynamics* (Singapore: World Scientific).

Grossmann, A., Kronland-Martinet, R., and Morlet, J. (1989). Reading and understanding continuous wavelet transforms, in *Wavelets*, eds. J.M. Combes, A. Grossmann, and P. Tchanmitchian (Berlin: Springer-Verlag), pp. 2–20.

Haar, A. (1910). Zur theorie der orthogonalen funktionensysteme, *Mathematische Annalen* **69**, 331–71.

Hackbusch, W. (1994). *Iterative Solution of Large Sparse Systems of Equations* (New York: Springer-Verlag).

Hackenbroich, G. and Weidenmüller, H.A. (1995). Universality of random-matrix results for non-Gaussian ensembles, *Physical Review Letters* **75**, 4118–21.

Harel, D. (2000). *Computers Ltd: What They Really Can't Do* (Oxford, UK: Oxford University Press).

Harlow, F.H. (1964). The particle-in-cell computing method for fluid dynamics, in *Methods in Computational Physics*, Volume 3, *Fundamental Methods in Hydrodynamics*, eds. B. Alder, S. Fernbach, and M. Rotenberg (New York: Academic), pp. 319–43.

Heath, M.T. (2002). *Scientific Computing: An Introductory Survey* (New York: McGraw Hill).

Heermann, D.W. (1986). *Computer Simulation Methods in Theoretical Physics* (Berlin: Springer-Verlag).

Hewson, A.C. (1993). *The Kondo Problem to Heavy Fermions* (Cambridge, UK: Cambridge University Press).

Higuera, F. and Jiménez, J. (1989). Boltzmann approach to lattice gas simulations, *Europhysics Letters* **9**, 663–8.

Hirsch, J.E. (1985a). Attractive interaction and pairing in fermion systems with strong on-site repulsion, *Physical Review Letters* **54**, 1317–20.

Hirsch, J.E. (1985b). Two-dimensional Hubbard model: Numerical simulation study, *Physical Review B* **31**, 4403–19.

Hochstrasser, U.W. (1965). Orthogonal polynomials, in *Handbook of Mathematical Functions*, eds. M. Abramowitz and I.A. Stegun (New York: Dover), pp. 771–802.

Hockney, R.W. and Eastwood, J.W. (1988). *Computer Simulation Using Particles* (London: McGraw–Hill).

Hohenberg, P. and Kohn, W. (1964). Inhomogeneous electron gas, *Physical Review* **136**, B864–71.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems* (Ann Arbor, Michigan: University of Michigan).

Holschneider, M. (1999). *Wavelets: An Analysis Tool* (Oxford, UK: Clarendon).

Hoover, W.G. (1985). Canonical dynamics: Equilibrium phase-space distributions, *Physical Review A* **31**, 1695–7.

Hoover, W.G. (1999). *Time Reversibility, Computer Simulation, and Chaos* (Singapore: World Scientific).

Hulthén, L. (1938). Uber das austauschproblem eines kristalles, *Arkiv för Matematik, Astronomi och Fysik* **26**(11), 1–105.

Ioffe, L.B., Feigel'man, M.V., Ioselevich, A., Ivanov, D., Troyer, M., and Blatter, G. (2002). Topologically protected quantum bits using Josephson junction arrays, *Nature* **415**, 503–6.

Jackson, J.D. (1999). *Classical Electrodynamics* (New York: Wiley).

Jacob, C. (2001). *Illustrating Evolutionary Computation with Mathematica* (San Francisco, California: Morgan Kaufmann).

Jóhannesson, G.H., Bligaard, H., Ruban, A.V., Skiver, H.L., Jacobsen, K.W., and Norskov, J.K. (2002). Combined electronic structure and evolutionary search approach to materials design, *Physical Review Letters* **88**, 255 506.

Jones, R.O. and Gunnarsson, O. (1989). The density functional formalism, its applications and prospects, *Reviews of Modern Physics* **61**, 689–746.

Jullien, R., Pfeuty, P., Fields, J.N., and Doniach, S. (1978). Zero-temperature renormalization method for quantum systems. I. Ising model in a transverse field in one dimension, *Physical Review B* **18**, 3568–78.

Kadanoff, L.P. (1966). Scaling laws for Ising models near T_c, *Physics* **2**, 263–72.

Kadanoff, L.P. (2000). *Statistical Physics: Statics, Dynamics, and Renormalization* (Singapore: World Scientific).

Kalos, M.H. (1962). Monte Carlo calculations of the ground state of three- and four-body nuclei, *Physical Review* **128**, 1791–5.

Kernighan, B.W. and Pike, R. (1984). *The UNIX Programming Environment* (Englewood Cliffs, New Jersey: Prentice–Hall).

Kernighan, B.W. and Ritchie, D.M. (1988). *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice–Hall).

Kittel, C. (1995). *Introduction to Solid State Physics* (New York: Wiley).

Knuth, D.E. (1998). *The Art of Computer Programming*, Volume 2, *Seminumerical Algorithms* (Reading, Massachusetts: Addison–Wesley).

Kohn, W. and Sham, L. (1965). Self-consistent equations including exchange and correlation effects, *Physical Review* **140**, A1133–8.

Kohn, W. and Vashishta, P. (1983). General functional density theory, in *Theory of the Inhomogeneous Electron Gas*, eds. S. Lundqvist and N.H. March (New York: Plenum), pp. 79–148.

Kondo, J. (1969). Theory of dilute magnetic alloys, in *Solid State Physics*, Volume 23, eds. F. Seitz, D. Turnbull, and H. Ehrenreich (New York: Academic), pp. 183–281.

Koniges, A.E. (ed.) (2000). *Industrial Strength Parallel Computing* (San Francisco, California: Morgan Kaufmann).

Konikow, L.F. and Reilly, T.E. (1999). Groundwater modeling, in *The Handbook of Groundwater Engineering*, ed. J.W. Delleur (Boca Raton, Florida: CRC Press), pp. 20-1–40.

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Cambridge, Massachusetts: MIT Press).

Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs* (Cambridge, Massachusetts: MIT Press).

Koza, J.R., Bennett III, F.H., Andre, D., and Keane, M.A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving* (San Francisco, California: Morgan Kaufmann).

Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., and Lanza G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence* (Boston, Massachusetts: Kluwer Academic).

Kunz, R.E. and Berry, R.S. (1993). Coexistence of multiple phases in finite systems, *Physical Review Letters* **71**, 3987–90.

Kunz, R.E. and Berry, R.S. (1994). Multiple phase coexistence in finite systems, *Physical Review E* **49**, 1895–908.

Landau, L.D. and Lifshitz, E.M. (1987). *Fluid Dynamics* (Oxford, UK: Pergamon).

Langdon, W.B. and Poli, R. (2002). *Foundations of Genetic Programming* (Berlin: Springer-Verlag).

Laughlin, R.B. (1998). A critique of two metals, *Advances in Physics* **47**, 943–58.

Laughlin, R.B. (2002). Gossamer superconductivity, cond-mat/0 209 269.

Lee, M.A. and Schmidt, K.E. (1992). Green's function Monte Carlo, *Computers in Physics* **6**, 192–7.

Lewis, D.W. (1991). *Matrix Theory* (Singapore: World Scientific).

Lifshitz, E.M. and Pitaevskii, L.P. (1981). *Physical Kinetics* (Oxford, UK: Pergamon).

Liu, K., Brown, M.G., Carter, C., Saykally, R.J., Gregory, J.K., and Clary, D.C. (1996). Characterization of a cage form of the water hexamer, *Nature* **381**, 501–3.

López, C., Álvarez, A., and Hernández-García. (2000). Forecasting confined spatiotemporal chaos with genetic algorithms. *Physical Review Letters* **85**, 2300–3.

Lyubartsev, A.P. and Vorontsov-Velyaminov, P.N. (1993). Path-integral Monte Carlo method in quantum statistics for a system of *N* identical fermions, *Physical Review A* **48**, 4075–83.

Ma, H. and Pang, T. (2004). Condensate-profile asymmetry of a boson mixture in a disk-shaped harmonic trap, *Physical Review A* **70**, 063 606.

Ma, S.-K. (1976). *Modern Theory of Critical Phenomena* (Reading, Massachusetts: Benjamin).

Mackintosh, A.R. (1987). The first electronic computer, *Physics Today* **40**(3), 25–32.

Madelung, O. (1978). *Introduction to Solid-State Theory* (Berlin: Springer-Verlag).

Mahan, G.D. (2000). *Many-Particle Physics* (New York: Plenum).

Mallat, S. (1989). A theory for multiresolution signal decomposition: The wavelet representation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11**, 674–93.

Matsuoka, H., Hirokawa, T., Matsui, M., and Doyama, M. (1992). Solid–liquid transitions in argon clusters, *Physics Review Letters* **69**, 297–300.

McCammon, J.A. and Harvey, S.C. (1987). *Dynamics of Proteins and Nucleic Acids* (Cambridge, UK: Cambridge University Press).

McCoy, B.M. and Wu, T.T. (1973). *The Two-Dimensional Ising Model* (Cambridge, Massachusetts: Harvard University Press).

McNamara, G.R. and Zanetti, G. (1988). Use of the Boltzmann equation to simulate lattice-gas automata, *Physical Review Letters* **61**, 2232–5.

Mehta, M.L. (1991). *Random Matrices* (Boston, Massachusetts: Academic).

Metcalf, M., Reid, J., and Cohen, M. (2004). *Fortran 95/2003 Explained* (New York: Oxford University Press).

Metropolis, N. and Frankel, S. (1947). Calculations in the liquid drop model of fission, *Physical Review* **72**, 186.

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., and Teller, E. (1953). Equation of state calculations by fast computing machines, *Journal of Chemical Physics* **21**, 1087–92.

Meyer, Y. (1993). *Wavelets: Algorithms & Applications*, translated and revised by R.D. Ryan (Philadelphia: SIAM).

Millikan, R.A. (1910). The isolation of an ion, a precision measurement of its charge, and the correction of Stokes's law, *Science* **32**, 436–48.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms* (Cambridge, Massachusetts: MIT Press).

Mollenhoff, C.R. (1988). *Atanasoff: Forgotten Father of the Computer* (Ames, Iowa: Iowa State University Press).

Monaghan, J.J. (1985). Particle methods for hydrodynamics, *Computer Physics Reports* **3**, 71–124.

Moreau, R. (1984). *The Computer Comes of Age: The People, the Hardware, and the Software* (Cambridge, Massachusetts: MIT Press).

Morlet, J., Arens, G., Fourgeau, E., and Giard, D. (1982a). Wave propagation and sampling theory – Part I: Complex signal and scattering in multilayered media, *Geophysics* **47**, 203–21.

Morlet, J., Arens, G., Fourgeau, E., and Giard, D. (1982b). Wave propagation and sampling theory – Part II: Sampling theory and complex waves, *Geophysics* **47**, 222–36.

Morris, J.R., Deaven, D.M., and Ho, K.M. (1996). Genetic-algorithm energy minimization for point charges on a sphere, *Physical Review B* **53**, R1740–3.

Moulopoulos, K. and Ashcroft, N.W. (1993). Many-body theory of paired electron crystals, *Physical Review B* **48**, 11 646–65.

Nash, S.G. (ed.) (1990). *A History of Scientific Computing* (Reading, Massachusetts: Addison–Wesley).

Newland, D.E. (1993). *An Introduction to Random Vibrations, Spectral and Wavelet Analysis* (Harlow, UK: Longman).

Niemeijer, Th. and van Leeuwen, J.M.J. (1974). Wilson theory for 2-dimensional Ising spin systems, *Physica* **71**, 17–40.

Nosé, S. (1984a). A molecular dynamics method for simulations in the canonical ensemble, *Molecular Physics* **52**, 255–68.

Nosé, S. (1984b). A unified formulation of the constant temperature molecular dynamics, *Journal of Chemical Physics* **81**, 511–19.

Nosé, S. (1991). Constant temperature molecular dynamics methods, *Progress of Theoretical Physics Supplement* **103**, 1–46.

Oguchi, T. and Sasaki, T. (1991). Density-functional molecular-dynamics method, *Progress of Theoretical Physics Supplement* **103**, 93–117.

Onodera, Y. (1994). Numerov integration for radial wave function in cylindrical symmetry, *Computers in Physics* **8**, 352–4.

Ortiz, G., Harris, M., and Ballone, P. (1999). Zero temperature phases of the electron gas, *Physical Review Letters* **82**, 5317–20.

Osborne, M.J. (2004). *An Introduction to Game Theory* (New York: Oxford University Press).

Pang, T. (1995). A numerical method for quantum tunneling, *Computers in Physics* **9**, 602–5.

Pang, T. (2005). The metallic state of the dilute two-dimensional electron gas: A resonating pair liquid? unpublished.

Parisi, G. (1988). *Statistical Field Theory* (Redwood City, California: Addison–Wesley).

Park, S.K. and Miller, K.W. (1988). Random number generators: Good ones are hard to find, *Communications of the ACM* **31**, 1192–201.

Parrinello, M. and Rahman, A. (1980). Crystal structure and pair potentials: A molecular-dynamics study, *Physical Review Letters* **45**, 1196–9.

Parrinello, M. and Rahman, A. (1981). Polymorphic transition in single crystal: A new molecular dynamics method, *Journal of Applied Physics* **52**, 7182–90.

Peaceman, D.W. and Rachford, H.H. Jr. (1955). The numerical solution of parabolic and elliptic difference equations, *Journal of the Society for Industrial and Applied Mathematics* **3**, 28–41.

Pearson, S., Pang, T., and Chen, C. (1998). Critical temperature of trapped hard-sphere Bose gases, *Physical Review A* **56**, 4796–800.

Percival, D.B. and Walden, A.T. (2000). *Wavelet Methods for Time Series Analysis* (Cambridge, UK: Cambridge University Press).

Pérez-Garrido, A. and Moore, M.A. (1999). Symmetric patterns of dislocations in Thomson's problem, *Physical Review B* **60**, 15 628.

Pollock, E.L. and Ceperley, D.M. (1984). Simulation of quantum many-body systems by path-integral methods, *Physical Review B* **30**, 2555–68.

Potter, D. (1977). *Computational Physics* (London: Wiley).

Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.R. (2002). *Numerical Recipes in C++: The Art of Scientific Computing* (Cambridge, UK: Cambridge University Press).

Pryce, J.D. (1993). *Numerical Solutions of Sturm–Liouville Problems* (Oxford, UK: Clarendon).

Pudalov, V.M., D'Iorio, M., Kravchenko, S.V., and Campbell, J.W. (1993). Zero-magnetic-field collective insulator phase in a dilute 2D electron system, *Physical Review Letters* **70**, 1866–9.

Rasetti, M. (ed.) (1991). *The Hubbard Model* (Singapore: World Scientific).

Reynolds, P.J., Ceperley, D.M., Alder, B.J., and Lester, W.A. Jr. (1982). Fixed-node quantum Monte Carlo for molecules, *Journal of Chemical Physics* **77**, 5593–603.

Ryan, C. (2000). *Automatic Re-engineering of Software Using Genetic Programming* (Boston, Massachusetts: Kluwer Academic).

Ryckaert, J.P., Ciccotti, G., and Berendsen, H.J.C. (1977). Numerical integration of the Cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes, *Journal of Computational Physics* **23**, 327–41.

Schmidt, K.E. and Ceperley, D.M. (1992). Monte Carlo techniques for quantum fluids, solids, and droplets, in *The Monte Carlo Method in Condensed Matter Physics*, ed. K. Binder (Berlin: Springer-Verlag), pp. 205–48.

Schollwöck, U. (2005). The density-matrix renormalization group, *Reviews of Modern Physics* **77**.

Shanno, F. (1970). Conditioning of quasi-Newton methods for function minimization, *Mathematics of Computation* **24**, 647–57.

Shashkin, A.A., Rahimi, M., Anissimova, S., Kravchenko, S.V., Dolgopolov, V.T., and Klapwijk, T.M. (2003). Spin-independent origin of the strongly enhanced effective mass in a dilute 2D electron system, *Physical Review Letters* **91**, 046 403.

Shibata, N. (2003). Application of the density matrix renormalization group method to finite temperatures and two-dimensional systems, *Journal of Physics A* **36**, R381–410.

Silverman, B.W. and Vassilicos, J.C. (eds.) (1999). *Wavelets: The Key to Intermittent Information?* (Oxford, UK: Oxford University Press).

Simos, T.E. (1993). A variable-step procedure for the numerical integration of the one-dimensional Schrödinger equation, *Computers in Physics* **7**, 460–4.

Spears, W.M. (1998). The Role of Mutation and Recombination in Evolutionary Algorithms, Ph.D. thesis, George Mason University, Fairfax, Virginia.

Stauffer, D. and Aharony, A. (1992). *Introduction to Percolation Theory* (London: Taylor & Francis).

Strandburg, K.J. (ed.) (1992). *Bond-Orientational Order in Condensed Matter Systems* (Berlin: Springer-Verlag).

Strang, G. (1989). Wavelets and dilation equations: A brief introduction, *SIAM Review* **31**, 614–27.

Strang, W.G. and Fix, G.J. (1973). *An Analysis of the Finite Element Method* (Englewood Cliffs, New Jersey: Prentice–Hall).

Stroustrup, B. (2000). *The C++ Programming Language* (Reading, Massachusetts: Addison–Wesley).

Succi, S. (2001). *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond* (Oxford, UK: Clarendon).

Sugano, S. and Koizumi, H. (1998). *Microcluster Physics* (Berlin: Springer-Verlag).

Sugar, R.L. (1990). Monte Carlo studies of many-electron systems, in *Computer Simulation Studies in Condensed Matter Physics II: New Directions*, eds. D.P. Landau, K.K. Mon, and H.-B. Schüttler (Berlin: Springer-Verlag), pp. 116–36.

Swendsen, R.H. (1979). Monte Carlo renormalization-group studies of the $d = 2$ Ising model, *Physical Review B* **20**, 2080–7.

Swendsen, R.H. and Wang, J.-S. (1987). Nonuniversal critical dynamics in Monte Carlo simulations, *Physical Review Letters* **58**, 86–8.

Swendsen, R.H., Wang, J.-S., and Ferrenberg, A.M. (1992). New Monte Carlo methods for improved efficiency of computer simulation in statistical mechanics, in *The Monte Carlo Methods in Condensed Matter Physics*, ed. K. Binder (Berlin: Springer-Verlag), pp. 75–91.

Sysi-Aho, M., Chakraborti, A., and Kaski, K. (2004). Search for good strategies in adaptive minority games, *Physical Review E* **69**, 036 125.

Szipiro, G.G. (1997). Forecasting chaotic time series with genetic algorithms, *Physical Review E* **55**, 2557–68.

Tassone, F., Mauri, F., and Car, R. (1994). Accelerated schemes for ab initio molecular-dynamics simulations and electronic-structure calculations, *Physical Review B* **50**, 10561–73.

Tinkham, M. (2003). *Group Theory and Quantum Mechanics* (New York: Dover).

Trivedi, N. and Ceperley, D.M. (1990). Ground-state correlations of quantum antiferromagnets: A Green-function Monte Carlo study, *Physical Review B* **41**, 4552–69.

Turing, A. (1936–7). On computable numbers: with an application to the entscheidungsproblem, *Proceedings of the London Mathematical Society, Series 2*, **42**, 230–65 (1936–7); correction *ibid*. **43**, 544–6 (1937).

Umrigar, C.J. (1993). Accelerated Metropolis method, *Physical Review Letters* **71**, 408–11.

Umrigar, C.J., Wilson, K.G., and Wilkins, J.W. (1988). Optimized trial wave functions for quantum Monte Carlo calculations, *Physical Review Letters* **60**, 1719–22.

van den Berg, J.C. (1999). *Wavelets in Physics* (Cambridge, UK: Cambridge University Press).

van der Linden, P. (2004). *Just Java 2* (Upper Saddle River, New Jersey: Prentice Hall).

van Gunsteren, W.F. and Berendsen, H.J.C. (1977). Algorithms for micromolecular dynamics and constraint dynamics, *Molecular Physics* **34**, 1311–27.

van Hove, L. (1954). Correlations in space and time and Born approximation scattering in systems of interacting particles, *Physical Review* **95**, 249–262.

Vashishta, P., Nakano, A., Kalia, R.K., and Ebbsjö, I. (1995). Molecular dynamics simulations of covalent amorphous insulators on parallel computers, *Journal of Non Crystalline Solids* **182**, 59–67.

Vichnevetsky, R. (1981). *Computer Methods for Partial Differential Equations, Volume I, Elliptic Equations and the Finite-Element Method* (Englewood Cliffs, New Jersey: Prentice Hall).

Vollhardt, D. (1984). Normal ^{3}He: An almost localized Fermi liquid, *Reviews of Modern Physics* **56**, 99–120.

Vose, M.D. (1999). *The Simple Genetic Algorithm: Foundations and Theory* (Cambridge, Massachusetts: MIT Press).

Wales, D.J. and Scheraga, H.A. (1999). Global optimization of clusters, crystals, and biomolecules, *Science* **285**, 1368–72.

Wang, C.Z., Chan, C.T., and Ho, K.M. (1989). Empirical tight-binding force model for molecular-dynamics simulation of Si, *Physical Review B* **39**, 8586–92.

Wang, H.F. and Anderson, M.P. (1982). *Introduction to Groundwater Modeling: Finite Difference and Finite Element Methods* (New York: Freeman).

Wenzel, W. and Hamacher, K. (1999). Stochastic tunneling approach for global optimization of complex potential energy landscapes, *Physical Review Letters* **82**, 3003–7.

White, S.R. (1992). Density matrix formulation for quantum renormalization groups, *Physical Review Letters* **69**, 2863–6.

White, S.R. (1993). Density-matrix algorithms for quantum renormalization groups, *Physical Review B* **48**, 10345–56.

White, S.R. (1998). Strongly correlated electron systems and the density matrix renormalization group, *Physics Reports* **301**, 187–204.

Widom, B. (1965a). Surface tension and molecular correlations near the critical point, *Journal of Chemical Physics* **43**, 3892–7.

Widom, B. (1965b). Equation of state in the neighborhood of the critical point, *Journal of Chemical Physics* **43**, 3898–905.

Wiegel, F.W. (1986). *Introduction to Path-Integral Methods in Physics and Polymer Science* (Singapore: World Scientific).

Wilkinson, J.H. (1963). *Rounding Errors in Algebraic Processes* (Englewood Cliffs, New Jersey: Prentice Hall).

Wilkinson, J.H. (1965). *Algebraic Eigenvalue Problems* (Oxford, UK: Clarendon).

Wille, L.T. (1986). Search potential energy surfaces by stimulated annealing, *Nature* **324**, 46–8.

Wilson, K.G. (1975). The renormalization group: Critical phenomena and the Kondo problem, *Reviews of Modern Physics* **47**, 773–840.

Wolf-Gladrow, D.A. (2000). *Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction* (Berlin: Springer-Verlag).

Wolff, U. (1989). Collective Monte Carlo updating for spin systems, *Physical Review Letters* **62**, 361–4.

Wolfram, S. (1986). Cellular automaton fluids 1: Basic theory, *Journal of Statistical Physics* **45**, 471–526.

Yonezawa, F. (1991). Glass transition and relaxation of disordered structures, in *Solid State Physics*, Volume 45, eds. H. Ehrenreich and D. Turnbull (Boston, Massachusetts: Academic), pp. 179–254.

Young, D.M. and Gregory, R.T. (1988). *A Survey of Numerical Mathematics*, Volume I and Volume II (New York: Dover).

Young, R. (1993). *Wavelet Theory and Its Applications* (Boston, Massachusetts: Kluwer Academic).

Zhang, F.C. (2003). Gossamer superconductor, Mott insulator, and resonating valence bond state in correlated electron systems, *Physical Review Letters* **90**, 207 002.

Zhou, J.G. (2004). *Lattice Boltzmann Methods for Shallow Water Flows* (Berlin: Springer-Verlag).

Index

Page links created automatically - disregard ones not formed from complete page numbers