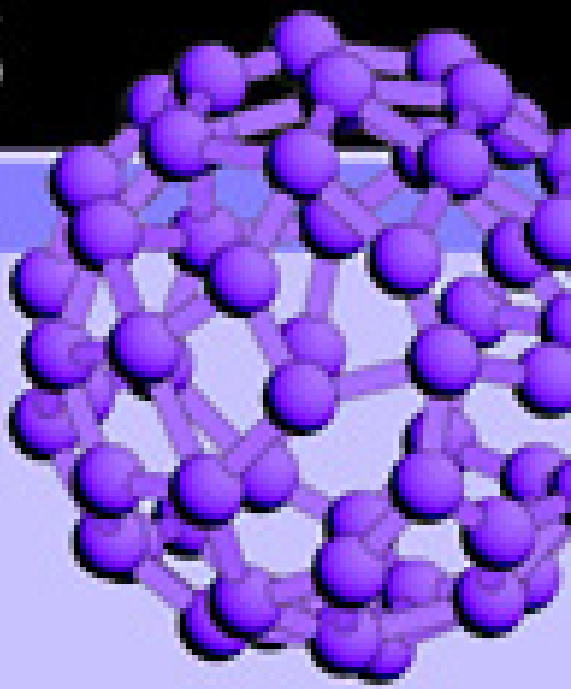
An Introduction to Computational Physics

SECOND EDITION



TAO PANG

CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521825696

This page intentionally left blank

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™, 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-13 978-0-511-13794-5 eBook (Adobe Reader)
ISBN-10 0-511-13794-X eBook (Adobe Reader)

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

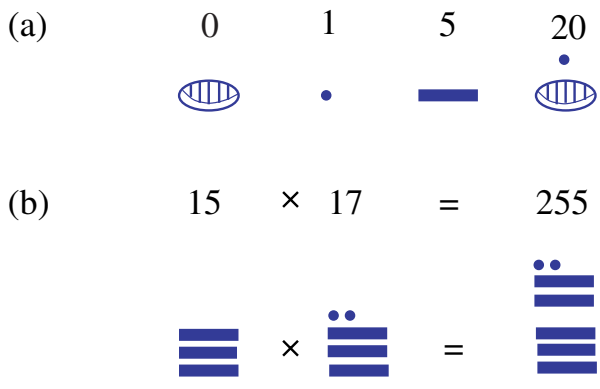


Fig. 1.1 The Mayan number system: (a) examples of using dots and bars to represent numbers; (b) an example of recording multiplication.

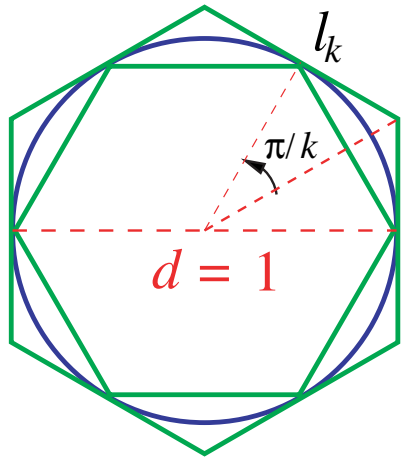


Fig. 1.2 A circle inscribed and circumscribed by two hexagons. The inside polygon sets the lower bound while the outside polygon sets the upper bound of the circumference.

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

$$\pi_k = k l_k . \tag{1.1}$$

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

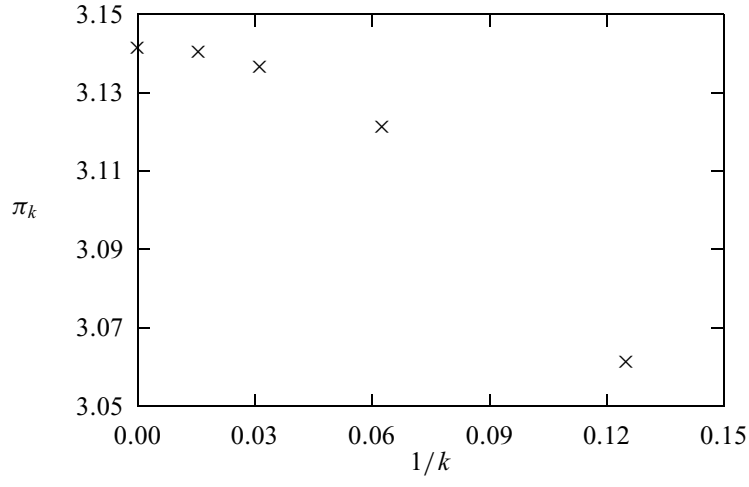


Fig. 1.3 The values of π_k, with $k = 8, 16, 32$, and 64, plotted together with the extrapolated π_∞.

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

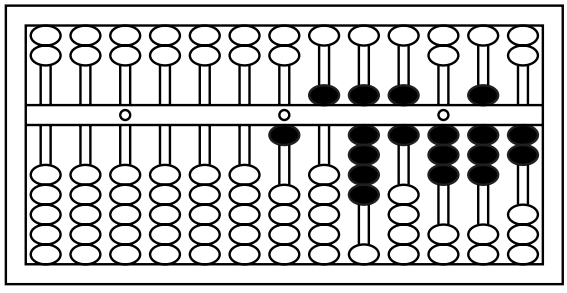


Fig. 1.4 A sketch of a Chinese abacus with the number 15 963.82 shown.

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.

```
// 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

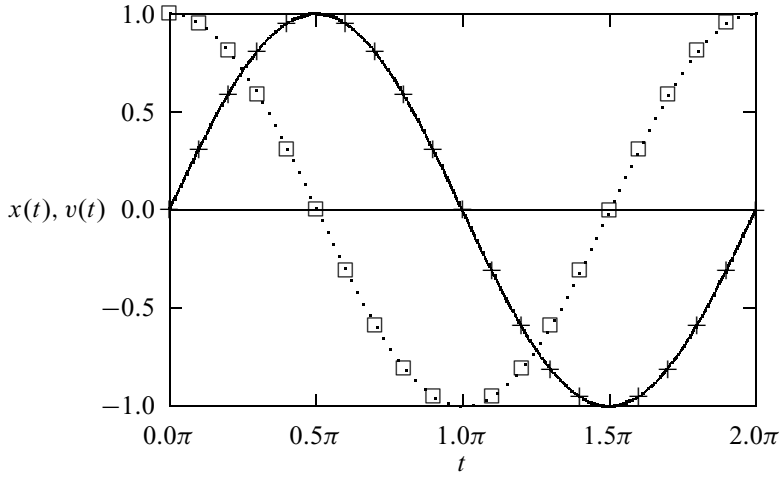


Fig. 1.5 The time-dependent position (+) and velocity (□) of the particle generated from the program `Motion.java` and the corresponding analytical results (solid and dotted lines, respectively).

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 languge 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 $G = 6.67 \times 10^{-11}\ \mathrm{N\,m^2/kg^2}$ 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 $A = 0.93\ \mathrm{m^2}$, the maximum taking-off speed of the motorcycle is 67 m/s, the air density is $\rho = 1.2\ \mathrm{kg/m^3}$, 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

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

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

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

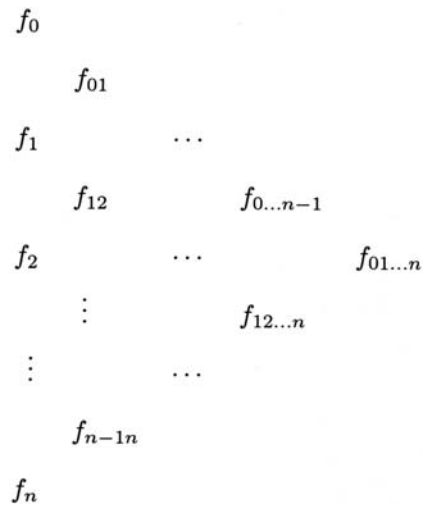


Fig. 2.1 The hierarchy in the Aitken scheme for $n+1$ data points.

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 $(x - x_0)(x - x_n)$, $(x - x_1)(x - x_{n-1})$, ..., 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

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

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

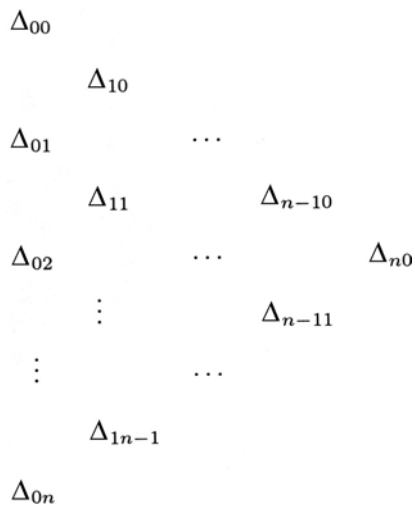


Fig. 2.2 The hierarchy for both Δ_{ij}^{+} and Δ_{ij}^{-} in the upward and downward correction method for $n+1$ data points.

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 $\Delta_{0j}^{\pm} = f_j$. 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.