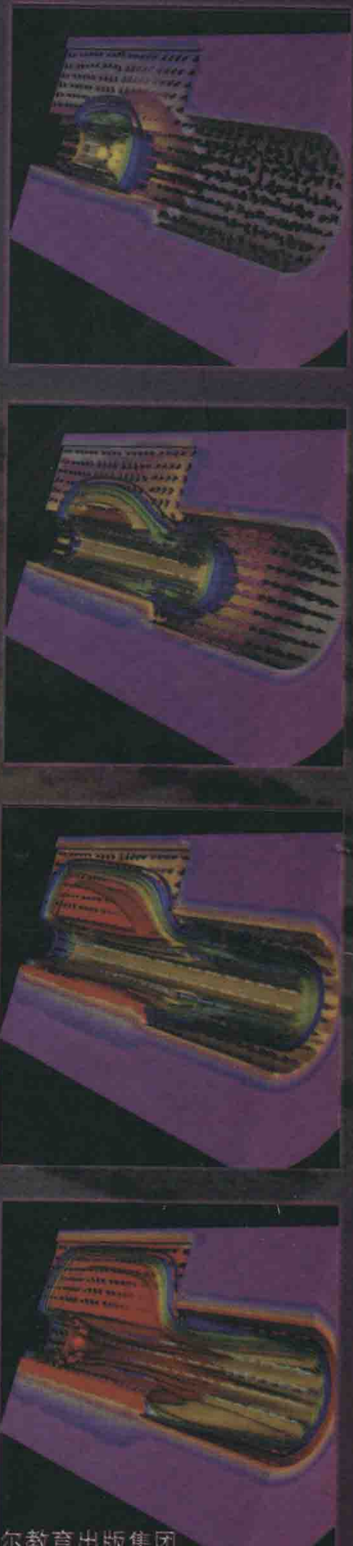

科学计算导论

（第2版）

SCIENTIFIC COMPUTING
AN INTRODUCTORY SURVEY

SECOND EDITION

MICHAEL T. HEATH

清华大学出版社
http://www.tup.tsinghua.edu.cn

McGraw Hill 麦格劳-希尔教育出版集团
http://www.mheducation.com

SCIENTIFIC COMPUTING
An Introductory Survey

Second Edition

科学计算导论（第2版）

Michael T. Heath
University of Illinois
at Urbana-Champaign

清华大学出版社
http://www.tup.tsinghua.edu.cn

Mc Graw Hill 麦格劳-希尔教育出版集团

(京) 新登字 158 号

Michael T. Heath
Scientific Computing: An Introductory Survey, Second Edition
ISBN: 0-07-239910-4 Reprint ISBN: 0-07-112229-X
Copyright © 2002, 1997 by the McGraw-Hill Companies, Inc.

北京市版权局著作权合同登记号: 01-2001-3467

书　名: Scientific Computing: An Introductory Survey, Second Edition
　　　　科学计算导论(第 2 版)
作　者: M. T. Heath
出版者: 清华大学出版社(北京清华大学学研大厦,邮编 100084)
　　　　http://www.tup.tsinghua.edu.cn
印刷者: 北京市密云胶印厂
发行者: 新华书店总店北京发行所
开　本: 787×960 1/16　印张: 36.25
版　次: 2001 年 10 月第 1 版　2001 年 10 月第 1 次印刷
书　号: ISBN 7-302-04900-9/O・270
印　数: 0001～3000
定　价: 55.00 元

影印版序

在刚刚过去的20世纪，数学经历了前所未有的繁荣与发展，其推动力当然来自于数学的逻辑体系内部，但更重要的则是自然科学和工程技术的推动。一方面今天几乎没有一个数学分支是“没用”的，数学在其他学科的应用空前广泛；另一方面其他学科不断提出全新的问题，它们成为数学发展的基本源泉。

科学计算作为当今科学研究的三种基本手段之一，是数学将触角伸向其他学科的桥梁，因此它的发展受到广泛关注。有些发达国家甚至将科学计算作为衡量国家综合实力的一个重要方面，大力推动其发展。也正因为如此，“科学计算”(或传统上所称的“数值分析”、“计算方法”)成为国内外理工科大学中开设最普遍的数学课程之一，其对象主要为研究生或高年级本科生。

国内外现有的“数值分析”教材有很多。笔者认为 M. T. HEATH 所著的这本“Scientific Computing: An Introductory Survey”很有特色，是近几年该领域少见的优秀之作。特别是第2版修订之后，更有可圈可点之处。

在学习这门课程的学生中，应区分为两个群体：其一是所谓“计算数学”专业的学生，他们学习的目标是要“创造”和“研究”算法的；而另一个群体显然要大得多，他们学习这门课程的目的是要“使用”算法的。后者中的优秀分子无疑会对“创造”算法有重要贡献，但这仍不是他们学习的目标所在。由于这两个群体学习的目标不同，所以教材的内容、体系及侧重点都应有相当的差别。但这个问题实际上是大部分现有教材没有很好解决的问题。

这本书突出的特点是：

(1) 它以“使用”算法的群体为对象，重点讲授算法背后的思想和原理，而不是算法及分析的细节，是一个很好的尝试。

(2) 比较多地讨论了算法间的联系，同一问题不同算法的比较和评价，提高学生对算法的“鉴赏”能力，从而达到正确使用算法的目标。

(3) 强调和突出了科学计算中的基本概念。如对一般“数值分析”教材中强调不充分的“病态性”和“敏感性”概念有较深入的讨论。

(4) 全书的每一章都有一节专门介绍和讨论有关的数学软件(包括 Internet 上的可以获得的免费软件和带有版权保护的商业软件平台)，对实际计算非常有价值。

(5) 配备了丰富的例题和习题，书中包括160多道例题，500多道思考题，240多道练习题和200多道数值计算题。

本书结构合理，可读性强，对以科学计算为工具的科技人员有参考价值，更可以作为

研究生“数值分析”课程的教材或参考书。

清华大学出版社从麦格劳—希尔(McGraw-Hill)公司引进该书在国内发行是一件十分有意义的事情。该书当然并不是十全十美,但相信对推动国内科学计算类课程的建设有所助益。

白峰杉教授
清华大学数学科学系
2001 年 8 月

About the Author

Michael T. Heath is a Professor of Computer Science at the University of Illinois at Urbana-Champaign, where he is also Director of the Computational Science and Engineering Program, Director of the Center for Simulation of Advanced Rockets, and Senior Research Scientist at the National Center for Supercomputing Applications (NCSA). He received a B.A. in Mathematics from the University of Kentucky, an M.S. in Mathematics from the University of Tennessee, and a Ph.D. in Computer Science from Stanford University. Before joining the University of Illinois in 1991, he spent a number of years at Oak Ridge National Laboratory, first as Eugene P. Wigner Postdoctoral Fellow and later as Computer Science Group Leader in the Mathematical Sciences Research Section. His research interests are in numerical analysis—particularly numerical linear algebra and optimization—and in parallel computing. He has been an editor of the *SIAM Journal on Scientific Computing*, *SIAM Review*, and the *International Journal of High Performance Computing Applications*, as well as several conference proceedings. In 2000, he was named an ACM Fellow by the Association for Computing Machinery.

Preface

This book presents a broad overview of numerical methods for students and professionals in computationally oriented disciplines who need to solve mathematical problems. It differs from traditional numerical analysis texts in that it focuses on the motivation and ideas behind the algorithms presented rather than on detailed analyses of them. I try to convey a general understanding of the techniques available for solving problems in each major category, including proper problem formulation and interpretation of results, but I advocate the use of professionally written mathematical software for obtaining solutions whenever possible. The book is aimed much more at potential users of mathematical software than at potential creators of such software. I hope to make the reader aware of the relevant issues in selecting appropriate methods and software and using them wisely.

At the University of Illinois, this book is used as the text for a comprehensive, one-semester course on numerical methods that serves three main purposes:

- As a terminal course for senior undergraduates, mainly computer science, mathematics, and engineering majors
- As a breadth course for graduate students in computer science who do *not* intend to specialize in numerical analysis
- As a training course for graduate students in science and engineering who need to use numerical methods and software in their research. It is a core course for the interdisciplinary graduate program in Computational Science and Engineering sponsored by the College of Engineering.

To accommodate this diverse student clientele, the prerequisites for the course and the book have been kept to a minimum: basic familiarity with linear algebra, multivariate calculus, and a smattering of differential equations. No prior familiarity with numerical methods is assumed. The book adopts a fairly sophisticated perspective, however, so a reasonable level of maturity on the part of the student (or reader) is advisable. Beyond the academic setting, I hope that the book will also be useful as a reference for practicing engineers and scientists who may need a quick overview of a given computational problem and the methods and software available for solving it.

Although the book emphasizes the use of mathematical software, unlike some other software-oriented texts it does not provide any software, nor does it concentrate on any specific software packages, libraries, or environments. Instead, for each problem category pointers are provided to specific routines available from publicly accessible repositories and the major commercial libraries and packages. In many academic and industrial computing environments such software is already installed, and in any case pointers are also provided to public domain software that is freely accessible via the Internet. The computer exercises in the book are not dependent on any specific choice of software or programming language.

The main elements in the organization of the book are as follows:

Chapters: Each chapter of the book covers a major computational problem area. The first half of the book deals primarily with algebraic problems, whereas the second half treats analytic problems involving derivatives and integrals. The first two chapters are fundamental to the remainder of the book, but the subsequent chapters can be taken in various orders according to the instructor's preference. More specifically, the major dependences among chapters are roughly as follows:

Chapter	Depends on	Chapter	Depends on	Chapter	Depends on
2	1	6	1, 2, 3, 5	10	1, 2, 5, 7, 9
3	1, 2	7	1, 2	11	1, 2, 7, 9, 10
4	1, 2, 3	8	1, 2, 7	12	1, 2, 7
5	1, 2	9	1, 2, 5, 7	13	1

Thus, Chapters 7, 8, 12, and 13 could be covered much earlier, and Chapters 3, 4 and 6 much later, than their appearance in the book. For example, Chapters 3, 7, and 12 all involve some type of data fitting, so it might be desirable to cover them as a unit. As another example, iterative methods for linear systems are contained in Chapter 11 on partial differential equations because that is where the most important motivating examples come from, but much of this material could be covered immediately following direct methods for linear systems in Chapter 2. Note that eigenvalues are used freely throughout the remainder of the book, so there is some incentive for covering Chapter 4 fairly early unless the students are already familiar with the basics of this topic.

There is more than enough material in the book for a full semester course, so some judicious omissions will likely be required in a one-term course. For example, Chapter 13 on random numbers and stochastic simulation is only peripherally related to the remainder of the book and is an obvious candidate for omission (random number generators are used in a number of exercises throughout the book, however). The entire book can be covered in a two-quarter or two-semester course.

Examples: Almost every concept and method introduced is illustrated by one or more examples. These examples are meant to supplement the relatively terse general discussion and should be read as an essential part of the text. The examples have been kept as simple as possible (sometimes at the risk of oversimplification) so that the reader can easily follow them. In my experience, a simple example that is thoroughly understood is usually more helpful than a more realistic example that is more difficult to follow.

Software: The lists of available software for each problem category are meant to be reasonably comprehensive. I have not attempted to single out the "best" software available for a given problem, partly because usually no single package is superior in all respects and partly to allow for the varied software availability and choice of programming language that may apply for different readers. All of the software cited is at least competently written, and some of it is superb.

Exercises: The book contains many exercises, which are divided into three categories:

- *Review questions*, which are short-answer questions designed to test basic conceptual understanding
- *Exercises*, which require somewhat more thought, longer answers, and possibly some hand computation
- *Computer problems*, which require some programming and often involve the use of existing software.

The *review questions* are meant for self-testing on the part of the reader. They include some deliberate repetition to drive home key points and to build confidence in the mastery of the material. The longer *exercises* are meant to be suitable for written homework assignments. Some of these require manual computations with simple examples, while others are designed to supply details of derivations and proofs omitted from the main text. The latter should be especially useful if the book is used for a more theoretical course. The *computer problems* provide an opportunity for hands-on experience in using the recommended software for solving typical problems in each category. Some of these problems are generic, but others are directly related to specific applications in various scientific and engineering disciplines.

Changes for the Second Edition. Each chapter now begins with a motivational discussion and one or more illustrative examples, which are then followed by discussions of existence, uniqueness, and conditioning of solutions for the given type of problem. The idea is to enhance the student's understanding of why the problem is important and how to recognize a "good" or "bad" formulation of the problem before considering algorithms for solving it. The major algorithms are now stated formally and numbered for easy reference. The bibliography has been brought up to date and the historical notes slightly expanded. The discussion in Chapter 1 on forward and backward error and the relationship between them has been expanded and clarified. Most of the material on the singular value decomposition has been moved from Chapter 4 to Chapter 3, where its applications fit more comfortably. The coverage of eigenvalue algorithms in Chapter 4 has been expanded to include more motivation and details, especially on QR iteration, as well as some additional methods. The treatment of constrained optimization in Chapter 6 has been substantially expanded. The chapters on differential equations have been slightly reorganized and the coverage of spectral methods expanded. Chapter 12 on the fast Fourier transform has been reorganized and streamlined by deleting some extraneous material.

I would like to acknowledge the influence of the mentors who first introduced me to the unexpected charms of numerical computation, Alston Householder and Gene Golub. I am grateful for the bountiful feedback I have received from students and instructors who have used the first edition of this book. Prepublication reviewers for the first edition were Alan George, University of Waterloo; Dianne O'Leary, University of Maryland; James Ortega, University of Virginia; John Strikwerda, University of Wisconsin; and Lloyd N. Trefethen, Oxford University. Reviewers of the first edition in preparation for the second edition were Thomas Coleman, Cornell University; Robert Funderlic, North Carolina State University; Thomas Hagstrom, University of New Mexico; Ramon Moore, Ohio State University; Mark Pernarowski, Montana State University; Linda Petzold, University of California at Santa Barbara; and Brian Suchomel, University of Minnesota. I thank all of these reviewers for their invaluable suggestions. In addition, I particularly want to acknowledge my colleagues Joerg Liesen, Paul Saylor, and Eric de Sturler, all of the University of Illinois, each of whom read some or all of the revised manuscript and provided invaluable feedback. I would like to thank Melchior Franz and Justin Winkler for helpful advice on typesetting that was crucial in enabling me to prepare camera-ready copy. Finally, I deeply appreciate the patience and understanding of my wife, Mona, during the countless hours spent in writing and revising this book. With great pleasure and gratitude I dedicate the book to her.

Michael T. Heath

Notation

The notation used in this book is fairly standard and should require little explanation. We freely use vector and matrix notation, generally using upper-case bold type for matrices, lower-case bold type for vectors, regular type for scalars. Iteration and component indices are denoted by subscripts, usually i through n. For example, a vector $\boldsymbol{x}$ and matrix $\boldsymbol{A}$ have entries x_i and a_{ij}, respectively. On the few occasions when both an iteration index and a component index are needed, the iteration is indicated by a parenthesized superscript, as in $x_i^{(k)}$ to indicate the ith component of the kth vector in a sequence. Otherwise, x_i denotes the ith component of a vector $\boldsymbol{x}$, whereas $\boldsymbol{x}_k$ denotes the kth vector in a sequence.

For simplicity, we will deal primarily with real vectors and matrices, although most of the theory and algorithms we discuss carry over with little or no change to the complex field. The set of real numbers is denoted by $\mathbb{R}$, n-dimensional real Euclidean space by $\mathbb{R}^n$, and the set of real $m \times n$ matrices by $\mathbb{R}^{m \times n}$. The analogous complex entities are denoted by $\mathbb{C}$, $\mathbb{C}^n$, and $\mathbb{C}^{m \times n}$, respectively.

The transpose of a vector or matrix is indicated by a superscript T, and the conjugate transpose by superscript H (for Hermitian transpose). Unless otherwise indicated, all vectors are regarded as column vectors; a row vector is indicated by explicitly transposing a column vector. For typesetting convenience, the components of a column vector are sometimes indicated by transposing the corresponding row vector, as in $\boldsymbol{x} = [x_1 \quad x_2]^T$. The inner product (also known as dot product or scalar product) of two n-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is a special case of matrix multiplication and thus is denoted by $\boldsymbol{x}^T\boldsymbol{y}$ (or $\boldsymbol{x}^H\boldsymbol{y}$ in the complex case). Similarly, the outer product of two n-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, which is an $n \times n$ matrix, is denoted by $\boldsymbol{x}\boldsymbol{y}^T$. The identity matrix of order n is denoted by $\boldsymbol{I}_n$ (or just $\boldsymbol{I}$ if the dimension n is clear from context), and its ith column is denoted by $\boldsymbol{e}_i$. A zero matrix is denoted by $\boldsymbol{O}$, a zero vector by $\boldsymbol{o}$, and a zero scalar by 0. A diagonal matrix with diagonal entries $d_1, \ldots, d_n$ is denoted by $\text{diag}(d_1, \ldots, d_n)$. Inequalities between vectors or matrices are taken to apply elementwise. The subspace of $\mathbb{R}^m$ spanned by the columns of an $m \times n$ matrix $\boldsymbol{A}$, i.e., $\{\boldsymbol{A}\boldsymbol{x} : \boldsymbol{x} \in \mathbb{R}^n\}$, is denoted by $\text{span}(\boldsymbol{A})$.

The ordinary derivative of a function $f(t)$ of one variable is denoted by df/dt or by $f'(t)$. Partial derivatives of a function of several variables, such as $u(x, y)$, are denoted by $\partial u/\partial x$, for example, or in some contexts by a subscript, as in u_x. Notation for gradient vectors and Jacobian and Hessian matrices will be introduced as needed. All logarithms are natural logarithms (base $e \approx 2.718$) unless another base is explicitly indicated. We use the symbol $\approx$ to indicate approximate equality in the ordinary sense and reserve the symbol $\cong$ specifically for least squares approximations.

The computational cost, or *complexity*, of numerical algorithms is usually measured by the number of arithmetic operations required. Traditionally, numerical analysts have counted only multiplications (and possibly divisions and square roots), because multiplications were usually significantly more expensive than additions or subtractions and because in most algorithms multiplications tend to be paired with a similar number of additions (for example, in computing the inner product of two vectors). More recently, the difference in cost between additions and multiplications has largely disappeared (indeed, many modern microprocessors can perform a coupled multiplication and addition with a single `multiply-add` instruction). Computer vendors and users like to advertise the highest possible performance, so it is increasingly common for every arithmetic operation to be counted. Because certain operation counts are so well known using the traditional practice, however, only multiplications are usually counted in this book. To clarify the meaning, the phrase "and a similar number of additions" will be added, or else it will be explicitly stated when both are being counted.

In quantifying operation counts and the accuracy of approximations, we will often use "big-oh" notation to indicate the order of magnitude, or dominant term, of a function. For an operation count, we are interested in the behavior as the size of the problem, say n, becomes large. We say that

$$f(n) = \mathcal{O}(g(n))$$

(read "f is big-oh of g" or "f is of order g") if there is a positive constant C such that

$$|f(n)| \le C|g(n)|$$

for all n sufficiently large. For example,

$$2n^3 + 3n^2 + n = \mathcal{O}(n^3)$$

because as n becomes large, the terms of order lower than n^3 become relatively insignificant. For an accuracy estimate, we are interested in the behavior as some quantity h, such as a step size or mesh spacing, becomes small. We say that

$$f(h) = \mathcal{O}(g(h))$$

if there is a positive constant C such that

$$|f(h)| \le C|g(h)|$$

for all h sufficiently small. For example,

$$\frac{1}{1-h} = 1 + h + h^2 + h^3 + \cdots = 1 + h + \mathcal{O}(h^2)$$

because as h becomes small, the omitted terms beyond h^2 become relatively insignificant. Note that the two definitions are equivalent if $h = 1/n$.

Contents

Chapter 1

Scientific Computing

1.1 Introduction

The subject of this book is traditionally called *numerical analysis*. Numerical analysis is concerned with the design and analysis of algorithms for solving mathematical problems that arise in many fields, especially science and engineering. For this reason, numerical analysis has more recently also become known as *scientific computing*. Scientific computing is distinguished from most other parts of computer science in that it deals with quantities that are *continuous*, as opposed to discrete. It is concerned with functions and equations whose underlying variables—time, distance, velocity, temperature, density, pressure, stress, and the like—are continuous in nature.

Most of the problems of continuous mathematics (for example, almost any problem involving derivatives, integrals, or nonlinearities) cannot be solved exactly, even in principle, in a finite number of steps and thus must be solved by a (theoretically infinite) iterative process that ultimately converges to a solution. In practice one does not iterate forever, of course, but only until the answer is approximately correct, "close enough" to the desired result for practical purposes. Thus, one of the most important aspects of scientific computing is finding rapidly convergent iterative algorithms and assessing the accuracy of the resulting approximation. If convergence is sufficiently rapid, even some of the problems that *can* be solved by finite algorithms, such as systems of linear algebraic equations, may in some cases be better solved by iterative methods, as we will see.

Consequently, a second factor that distinguishes scientific computing is its concern with the effects of approximations. Many solution techniques involve a whole series of approximations of various types. Even the arithmetic used is only approximate, for digital computers cannot represent all real numbers exactly. In addition to having the usual properties of good algorithms, such as efficiency, numerical algorithms should also be as reliable and accurate as possible despite the various approximations made along the way.

1.1.1 Computational Problems

As the name suggests, many problems in scientific computing come from science and engineering, in which the ultimate aim is to understand some natural phenomenon or to design some device. *Computational simulation* is the representation and emulation of a physical system or process using a computer. Computational simulation can greatly enhance scientific understanding by allowing the investigation of situations that may be difficult or impossible to investigate by theoretical, observational, or experimental means alone. In astrophysics, for example, the detailed behavior of two colliding black holes is too complicated to determine theoretically and impossible to observe directly or duplicate in the laboratory. To simulate it computationally, however, requires only an appropriate mathematical representation (in this case Einstein's equations of general relativity), an algorithm for solving those equations numerically, and a sufficiently large computer on which to implement the algorithm.

Computational simulation is useful not just for exploring exotic or otherwise inaccessible situations, however, but also for exploring a wider variety of "normal" scenarios than could otherwise be investigated with reasonable cost and time. In engineering design, computational simulation allows a large number of design options to be tried much more quickly, inexpensively, and safely than with traditional "build-and-test" methods using physical prototypes. In this context, computational simulation has become known as *virtual prototyping*. In improving automobile safety, for example, crash testing is far less expensive and dangerous on a computer than in real life, and thus the space of possible design parameters can be explored much more thoroughly to develop an optimal design.

The overall problem solving process in computational simulation usually includes the following steps:

1. Develop a mathematical model—usually expressed by equations of some type—of a physical phenomenon or system of interest
2. Develop algorithms to solve the equations numerically
3. Implement the algorithms in computer software
4. Run the software on a computer to simulate the physical process numerically
5. Represent the computed results in some comprehensible form such as graphical visualization
6. Interpret and validate the computed results, repeating any or all of the preceding steps, if necessary.

Step 1 is often called *mathematical modeling*. It requires specific knowledge of the particular scientific or engineering disciplines involved as well as knowledge of applied mathematics. Steps 2 and 3—designing, implementing, analyzing, and using numerical algorithms and software—are the main subject matter of scientific computing, and of this book in particular. Although we will focus on Steps 2 and 3, it is essential that all of these steps, from problem formulation to interpretation and