国际著名数学图书——影印版

Applied Numerical Linear Algebra

应用数值线性代数

James W. Demmel 著

TSINGHUA UNIVERSITY PRESS

siam

Applied Numerical Linear Algebra

应用数值线性代数

James W. Demmel 著

TSINGHUA UNIVERSITY PRESS
北京

siam

James W. Demmel
Applied Numerical Linear Algebra
ISBN:0-89871-389-7

图书在版编目（CIP）数据

应用数值线性代数 = Applied Numerical Linear Algebra：英文/（美）戴梅尔（Demmel，J. W.）著.--影印本.--北京：清华大学出版社，2011.2
（国际著名数学图书）
ISBN 978-7-302-24500-1

Ⅰ. ①应… Ⅱ. ①戴… Ⅲ. ①线性代数计算法-英文 Ⅳ. ①O241.6

中国版本图书馆CIP数据核字（2011）第006627号

责任编辑：陈朝晖
责任印制：王秀菊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印　装　者：北京密云胶印厂
经　　销：全国新华书店
开　　本：175×245　**印　张**：27.25
版　　次：2011 年 2 月第 1 版　　**印　　次**：2011 年 2 月第1 次印刷
印　　数：1～3000
定　　价：59.00 元

产品编号：027797-01

Preface

This textbook covers both direct and iterative methods for the solution of linear systems, least squares problems, eigenproblems, and the singular value decomposition. Earlier versions have been used by the author in graduate classes in the Mathematics Department of the University of California at Berkeley since 1990 and at the Courant Institute before then.

In writing this textbook I aspired to meet the following goals:

1. The text should be attractive to first-year graduate students from a variety of engineering and scientific disciplines.

2. It should be self-contained, assuming only a good undergraduate background in linear algebra.

3. The students should learn the mathematical basis of the field, as well as how to build or find good numerical software.

4. Students should acquire practical knowledge for solving real problems efficiently. In particular, they should know what the state-of-the-art techniques are in each area or when to look for them and where to find them, even if I analyze only simpler versions in the text.

5. It should all fit in one semester, since that is what most students have available for this subject.

Indeed, I was motivated to write this book because the available textbooks, while very good, did not meet these goals. Golub and Van Loan's text [121] is too encyclopedic in style, while still omitting some important topics such as multigrid, domain decomposition, and some recent algorithms for eigenvalue problems. Watkins's [252] and Trefethen's and Bau's [243] also omit some state-of-the-art algorithms.

While I believe that these five goals were met, the fifth goal was the hardest to manage, especially as the text grew over time to include recent research results and requests from colleagues for new sections. A reasonable one-semester curriculum based on this book would cover

- Chapter 1, excluding section 1.5.1;
- Chapter 2, excluding sections 2.2.1, 2.4.3, 2.5, 2.6.3, and 2.6.4;
- Chapter 3, excluding sections 3.5 and 3.6;

- Chapter 4, up to and including section 4.4.5;
- Chapter 5, excluding sections 5.2.1, 5.3.5, 5.4 and 5.5;
- Chapter 6, excluding sections 6.3.3, 6.5.5, 6.5.6, 6.6.6, 6.7.2, 6.7.3, 6.7.4, 6.8, 6.9.2, and 6.10; and
- Chapter 7, up to and including section 7.3.

Notable features of this book include

- a class homepage with Matlab source code for examples and homework problems in the text;
- frequent recommendations and pointers to the best software currently available (from LAPACK and elsewhere);
- a discussion of how modern cache-based computer memories impact algorithm design;
- performance comparisons of competing algorithms for least squares and symmetric eigenvalue problems;
- a discussion of a variety of iterative methods, from Jacobi's to multigrid, with detailed performance comparisons for solving Poisson's equation on a square grid;
- detailed discussion and numerical examples for the Lanczos algorithm for the symmetric eigenvalue problem;
- numerical examples drawn from fields ranging from mechanical vibrations to computational geometry;
- sections on "relative perturbation theory" and corresponding high-accuracy algorithms for symmetric eigenvalue problems and the singular value decomposition; and
- dynamical systems interpretations of eigenvalue algorithms.

The URL for the class homepage will be abbreviated to HOMEPAGE throughout the text, standing for http://www.siam.org/books/demmel/demmel_class. Two other abbreviated URLs will be used as well. PARALLEL_HOMEPAGE is an abbreviation for http://www.siam.org/books/demmel/demmel_parallelclass and points to a related on-line class by the author on parallel computing. NETLIB is an abbreviation for http://www.netlib.org.

Homework problems are marked Easy, Medium, or Hard, according to their difficulty. Problems involving significant amounts of programming are marked "programming."

Many people have contributed to this text. Most notably, Zhaojun Bai used this text at Texas A&M and the University of Kentucky, contributed numerous questions, and made many useful suggestions. Alan Edelman (who used this book at MIT), Martin Gutknecht (who used this book at ETH Zurich), Velvel Kahan (who used this book at Berkeley), Richard Lehoucq, Beresford Parlett, and many anonymous referees made detailed comments on various parts of the text. In addition, Alan Edelman and Martin Gutknecht provided hospitable surroundings while this final edition was being prepared. Table 2.2 is taken from the Ph.D. thesis of my former student Xiaoye Li. Mark Adams, Tzu-Yi Chen, Inderjit Dhillon, Jian Xun He, Melody Ivory, Xiaoye Li, Bernd Pfrommer, Huan Ren, and Ken Stanley, along with many other students at Courant, Berkeley, Kentucky, and MIT over the years, helped debug the text. Bob Untiedt and Selene Victor were of great help in typesetting and producing figures. Megan supplied the cover photo. Finally, Kathy Yelick has contributed support over more years than either of us expected this project to take.

James Demmel
Berkeley, California
June 1997

Contents

1

Introduction

1.1. Basic Notation

In this course we will refer frequently to *matrices*, *vectors*, and *scalars*. A matrix will be denoted by an upper case letter such as A, and its (i,j)th element will be denoted by a_{ij}. If the matrix is given by an expression such as $A+B$, we will write $(A+B)_{ij}$. In detailed algorithmic descriptions we will sometimes write $A(i,j)$ or use the Matlab^{TM} [1] [184] notation $A(i:j,k:l)$ to denote the submatrix of A lying in rows i through j and columns k through l. A lower-case letter like x will denote a vector, and its ith element will be written x_i. Vectors will almost always be column vectors, which are the same as matrices with one column. Lower-case Greek letters (and occasionally lower-case letters) will denote scalars. $\mathbb{R}$ will denote the set of real numbers; $\mathbb{R}^n$, the set of n-dimensional real vectors; and $\mathbb{R}^{m\times n}$, the set of m-by-n real matrices. $\mathbb{C}$, $\mathbb{C}^n$, and $\mathbb{C}^{m\times n}$ denote complex numbers, vectors, and matrices, respectively. Occasionally we will use the shorthand $A^{m\times n}$ to indicate that A is an m-by-n matrix. A^T will denote the *transpose* of the matrix A: $(A^T)_{ij}=a_{ji}$. For complex matrices we will also use the *conjugate transpose* A^*: $(A^*)_{ij}=\bar{a}_{ji}$. $\Re z$ and $\Im z$ will denote the real and imaginary parts of the complex number z, respectively. If A is m-by-n, then $|A|$ is the m-by-n matrix of absolute values of entries of A: $(|A|)_{ij}=|a_{ij}|$. Inequalities like $|A|\le|B|$ are meant componentwise: $|a_{ij}|\le|b_{ij}|$ for all i and j. We will also use this absolute value notation for vectors: $(|x|)_i=|x_i|$. Ends of proofs will be marked by $\square$, and ends of examples by $\diamond$. Other notation will be introduced as needed.

1.2. Standard Problems of Numerical Linear Algebra

We will consider the following standard problems:

[1] Matlab is a registered trademark of The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760, USA, tel. 508-647-7000, fax 508-647-7001, info@mathworks.com, http://www.mathworks.com.

- *Linear systems of equations*: Solve $Ax = b$. Here A is a given n-by-n nonsingular real or complex matrix, b is a given column vector with n entries, and x is a column vector with n entries that we wish to compute.

- *Least squares problems*: Compute the x that minimizes $\|Ax - b\|_2$. Here A is m-by-n, b is m-by-1, x is n-by-1, and $\|y\|_2 \equiv \sqrt{\sum_i |y_i|^2}$ is called the *two-norm* of the vector y. If $m > n$ so that we have more equations than unknowns, the system is called *overdetermined*. In this case we cannot generally solve $Ax = b$ exactly. If $m < n$, the system is called *underdetermined*, and we will have infinitely many solutions.

- *Eigenvalue problems*: Given an n-by-n matrix A, find an n-by-1 nonzero vector x and a scalar λ so that $Ax = \lambda x$.

- *Singular value problems*: Given an m-by-n matrix A, find an n-by-1 nonzero vector x and scalar λ so that $A^T Ax = \lambda x$. We will see that this special kind of eigenvalue problem is important enough to merit separate consideration and algorithms.

We choose to emphasize these standard problems because they arise so often in engineering and scientific practice. We will illustrate them throughout the book with simple examples drawn from engineering, statistics, and other fields. There are also many variations of these standard problems that we will consider, such as generalized eigenvalue problems $Ax = \lambda Bx$ (section 4.5) and "rank-deficient" least squares problems $\min_x \|Ax - b\|_2$, whose solutions are nonunique because the columns of A are linearly dependent (section 3.5).

We will learn the importance of exploiting any *special structure* our problem may have. For example, solving an n-by-n linear system costs $2/3n^3$ floating point operations if we use the most general form of Gaussian elimination. If we add the information that the system is symmetric and positive definite, we can save half the work by using another algorithm called Cholesky. If we further know the matrix is *banded* with *semibandwidth* $\sqrt{n}$ (i.e., $a_{ij} = 0$ if $|i-j| > \sqrt{n}$), then we can reduce the cost further to $O(n^2)$ by using band Cholesky. If we say quite explicitly that we are trying to solve Poisson's equation on a square using a 5-point difference approximation, which determines the matrix nearly uniquely, then by using the multigrid algorithm we can reduce the cost to $O(n)$, which is nearly as fast as possible, in the sense that we use just a constant amount of work per solution component (section 6.4).

1.3. General Techniques

There are several general concepts and techniques that we will use repeatedly:

1. matrix factorizations;

2. perturbation theory and condition numbers;

3. effects of roundoff error on algorithms, including properties of floating point arithmetic;
4. analysis of the speed of an algorithm;
5. engineering numerical software.

We discuss each of these briefly below.

1.3.1. Matrix Factorizations

A *factorization* of the matrix A is a representation of A as a product of several "simpler" matrices, which make the problem at hand easier to solve. We give two examples.

EXAMPLE 1.1. Suppose that we want to solve $Ax = b$. If A is a lower triangular matrix,

$$\begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

is easy to solve using *forward substitution*:

for $i = 1$ to n
    $x_i = (b_i - \sum_{k=1}^{i-1} a_{ik}x_k)/a_{ii}$
end for

An analogous idea, *back substitution*, works if A is upper triangular. To use this to solve a general system $Ax = b$ we need the following matrix factorization, which is just a restatement of Gaussian elimination.

THEOREM 1.1. *If the n-by-n matrix A is nonsingular, there exist a permutation matrix P (the identity matrix with its rows permuted), a nonsingular lower triangular matrix L, and a nonsingular upper triangular matrix U such that $A = P \cdot L \cdot U$. To solve $Ax = b$, we solve the equivalent system $PLUx = b$ as follows:*

$$\begin{aligned} LUx &= P^{-1}b = P^Tb && \textit{(permute entries of } b\textit{)}, \\ Ux &= L^{-1}(P^Tb) && \textit{(forward substitution)}, \\ x &= U^{-1}(L^{-1}P^Tb) && \textit{(back substitution)}. \end{aligned}$$

We will prove this theorem in section 2.3. ◇

EXAMPLE 1.2. The *Jordan canonical factorization* $A = VJV^{-1}$ exhibits the eigenvalues and eigenvectors of A. Here V is a nonsingular matrix, whose columns include the eigenvectors, and J is the *Jordan canonical form* of A,

a special triangular matrix with the eigenvalues of A on its diagonal. We will learn that it is numerically superior to compute the *Schur factorization* $A = UTU^*$, where U is a unitary matrix (i.e., U's columns are orthonormal) and T is upper triangular with A's eigenvalues on its diagonal. The Schur form T can be computed faster and more accurately than the Jordan form J. We discuss the Jordan and Schur factorizations in section 4.2. ◇

1.3.2. Perturbation Theory and Condition Numbers

The answers produced by numerical algorithms are seldom exactly correct. There are two sources of error. First, there may be errors in the input data to the algorithm, caused by prior calculations or perhaps measurement errors. Second, there are errors caused by the algorithm itself, due to approximations made within the algorithm. In order to estimate the errors in the computed answers from both these sources, we need to understand how much the solution of a problem is changed (or *perturbed*) if the input data are slightly perturbed.

EXAMPLE 1.3. Let $f(x)$ be a real-valued differentiable function of a real variable x. We want to compute $f(x)$, but we do not know x exactly. Suppose instead that we are given $x + \delta x$ and a bound on δx. The best that we can do (without more information) is to compute $f(x + \delta x)$ and to try to bound the absolute error $|f(x + \delta x) - f(x)|$. We may use a simple linear approximation to f to get the estimate $f(x + \delta x) \approx f(x) + \delta x f'(x)$, and so the error is $|f(x+\delta x) - f(x)| \approx |\delta x| \cdot |f'(x)|$. We call $|f'(x)|$ the *absolute condition number* of f at x. If $|f'(x)|$ is large enough, then the error may be large even if δx is small; in this case we call f *ill-conditioned* at x. ◇

We say *absolute* condition number because it provides a bound on the absolute error $|f(x + \delta x) - f(x)|$ given a bound on the absolute change $|\delta x|$ in the input. We will also often use the following essentially equivalent expression to bound the error:

$$\frac{|f(x + \delta x) - f(x)|}{|f(x)|} \approx \frac{|\delta x|}{|x|} \cdot \frac{|f'(x)| \cdot |x|}{|f(x)|}.$$

This expression bounds the *relative error* $|f(x + \delta x) - f(x)|/|f(x)|$ as a multiple of the *relative* change $|\delta x|/|x|$ in the input. The multiplier, $|f'(x)| \cdot |x|/|f(x)|$, is called the *relative condition number*, or often just *condition number* for short.

The condition number is all that we need to understand how error in the input data affects the computed answer: we simply multiply the condition number by a bound on the input error to bound the error in the computed solution.

For each problem we consider, we will derive its corresponding condition number.

1.3.3. Effects of Roundoff Error on Algorithms

To continue our analysis of the error caused by the algorithm itself, we need to study the effect of roundoff error in the arithmetic, or simply roundoff for short. We will do so by using a property possessed by most good algorithms: *backward stability*. We define it as follows.

> If alg(x) is our algorithm for $f(x)$, including the effects of roundoff, we call alg(x) a *backward stable algorithm* for $f(x)$ if for all x there is a "small" δx such that alg(x) $= f(x + \delta x)$. δx is called the *backward error*. Informally, we say that we get the exact answer ($f(x + \delta x)$) for a slightly wrong problem ($x + \delta x$).

This implies that we may bound the error as

$$\text{error} = |\text{alg}(x) - f(x)| = |f(x + \delta x) - f(x)| \approx |f'(x)| \cdot |\delta x|,$$

the product of the absolute condition number $|f'(x)|$ and the magnitude of the backward error $|\delta x|$. Thus, if alg($\cdot$) is backward stable, $|\delta x|$ is always small, so the error will be small unless the absolute condition number is large. Thus, backward stability is a desirable property for an algorithm, and most of the algorithms that we present will be backward stable. Combined with the corresponding condition numbers, we will have error bounds for all our computed solutions.

Proving that an algorithm is backward stable requires knowledge of the roundoff error of the basic floating point operations of the machine and how these errors propagate through an algorithm. This is discussed in section 1.5.

1.3.4. Analyzing the Speed of Algorithms

In choosing an algorithm to solve a problem, one must of course consider its speed (which is also called performance) as well as its backward stability. There are several ways to estimate speed. Given a particular problem instance, a particular implementation of an algorithm, and a particular computer, one can of course simply run the algorithm and see how long it takes. This may be difficult or time consuming, so we often want simpler estimates. Indeed, we typically want to estimate how long a particular algorithm would take *before* implementing it.

The traditional way to estimate the time an algorithm takes is to count the *flops*, or *floating point operations*, that it performs. We will do this for all the algorithms we present. However, this is often a misleading time estimate on modern computer architectures, because it can take significantly more time to move the data inside the computer to the place where it is to be multiplied, say, than it does to actually perform the multiplication. This is especially true on parallel computers but also is true on conventional machines such as workstations and PCs. For example, matrix multiplication on

the IBM RS6000/590 workstation can be sped up from 65 Mflops (millions of floating point operations per second) to 240 Mflops, nearly four times faster, by judiciously reordering the operations of the standard algorithm (and using the correct compiler optimizations). We discuss this further in section 2.6.

If an algorithm is *iterative*, i.e., produces a series of approximations converging to the answer rather than stopping after a fixed number of steps, then we must ask how many steps are needed to decrease the error to a tolerable level. To do this, we need to decide if the convergence is *linear* (i.e., the error decreases by a constant factor $0 < c < 1$ at each step so that $|\text{error}_i| \leq c \cdot |\text{error}_{i-1}|$) or faster, such as *quadratic* ($|\text{error}_i| \leq c \cdot |\text{error}_{i-1}|^2$). If two algorithms are both linear, we can ask which has the smaller constant c. Iterative linear equation solvers and their convergence analysis are the subject of Chapter 6.

1.3.5. Engineering Numerical Software

Three main issues in designing or choosing a piece of numerical software are *ease of use*, *reliability*, and *speed*. Most of the algorithms covered in this book have already been carefully programmed with these three issues in mind. If some of this existing software can solve your problem, its ease of use may well outweigh any other considerations such as speed. Indeed, if you need only to solve your problem once or a few times, it is often easier to use general purpose software written by experts than to write your own more specialized program.

There are three programming paradigms for exploiting other experts' software. The first paradigm is the traditional software library, consisting of a collection of subroutines for solving a fixed set of problems, such as solving linear systems, finding eigenvalues, and so on. In particular, we will discuss the LAPACK library [10], a state-of-the-art collection of routines available in Fortran and C. This library, and many others like it, are freely available in the public domain; see NETLIB on the World Wide Web.[2] LAPACK provides reliability and high speed (for example, making careful use of matrix multiplication, as described above) but requires careful attention to data structures and calling sequences on the part of the user. We will provide pointers to such software throughout the text.

The second programming paradigm provides a much easier-to-use environment than libraries like LAPACK, but at the cost of some performance. This paradigm is provided by the commercial system Matlab [184], among others. Matlab provides a simple interactive programming environment where all variables represent matrices (scalars are just 1-by-1 matrices), and most linear algebra operations are available as built-in functions. For example, "$C = A * B$" stores the product of matrices A and B in C, and "$A = \text{inv}(B)$" stores the inverse of matrix B in A. It is easy to quickly prototype algorithms in Matlab and to see how they work. But since Matlab makes a number of algorith-

[2]Recall that we abbreviate the URL prefix http://www.netlib.org to NETLIB in the text.