APPLIED NUMERICAL LINEAR ALGEBRA

Applied Numerical Linear Algebra

James W. Demmel
University of California
Berkeley, California

siam Society for Industrial and Applied Mathematics

Philadelphia

10 9 8 7 6

Library of Congress Cataloging-in-Publication Data

Demmel, James W.
Applied numerical linear algebra / James W. Demmel.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-898713-89-3 (pbk.)
1. Algebras, Linear. 2. Numerical calculations. I. Title.
QA184.D455 1997
512'.5--dc21 97-17290

The four images on the cover show an original image of a baby as well as three versions compressed using the singular value decomposition. See Example 3.4 on pages 113–116 for details.

Contents

Preface ix

1 Introduction 1
- 1.1 Basic Notation . . . 1
- 1.2 Standard Problems of Numerical Linear Algebra . . . 1
- 1.3 General Techniques . . . 2
  - 1.3.1 Matrix Factorizations . . . 3
  - 1.3.2 Perturbation Theory and Condition Numbers . . . 4
  - 1.3.3 Effects of Roundoff Error on Algorithms . . . 5
  - 1.3.4 Analyzing the Speed of Algorithms . . . 5
  - 1.3.5 Engineering Numerical Software . . . 6
- 1.4 Example: Polynomial Evaluation . . . 7
- 1.5 Floating Point Arithmetic . . . 9
  - 1.5.1 Further Details . . . 12
- 1.6 Polynomial Evaluation Revisited . . . 15
- 1.7 Vector and Matrix Norms . . . 19
- 1.8 References and Other Topics for Chapter 1 . . . 23
- 1.9 Questions for Chapter 1 . . . 24

2 Linear Equation Solving 31
- 2.1 Introduction . . . 31
- 2.2 Perturbation Theory . . . 32
  - 2.2.1 Relative Perturbation Theory . . . 35
- 2.3 Gaussian Elimination . . . 38
- 2.4 Error Analysis . . . 44
  - 2.4.1 The Need for Pivoting . . . 45
  - 2.4.2 Formal Error Analysis of Gaussian Elimination . . . 46
  - 2.4.3 Estimating Condition Numbers . . . 50
  - 2.4.4 Practical Error Bounds . . . 54
- 2.5 Improving the Accuracy of a Solution . . . 60
  - 2.5.1 Single Precision Iterative Refinement . . . 62
  - 2.5.2 Equilibration . . . 62
- 2.6 Blocking Algorithms for Higher Performance . . . 63
  - 2.6.1 Basic Linear Algebra Subroutines (BLAS) . . . 66
  - 2.6.2 How to Optimize Matrix Multiplication . . . 67
  - 2.6.3 Reorganizing Gaussian Elimination to Use Level 3 BLAS 72
  - 2.6.4 More About Parallelism and Other Performance Issues . 75

2.7 Special Linear Systems 76
2.7.1 Real Symmetric Positive Definite Matrices 76
2.7.2 Symmetric Indefinite Matrices 79
2.7.3 Band Matrices 79
2.7.4 General Sparse Matrices 83
2.7.5 Dense Matrices Depending on Fewer Than $O(n^2)$ Parameters 90
2.8 References and Other Topics for Chapter 2 93
2.9 Questions for Chapter 2 93

3 Linear Least Squares Problems **101**
3.1 Introduction 101
3.2 Matrix Factorizations That Solve the Linear Least Squares Problem 105
3.2.1 Normal Equations 106
3.2.2 QR Decomposition 107
3.2.3 Singular Value Decomposition 109
3.3 Perturbation Theory for the Least Squares Problem 117
3.4 Orthogonal Matrices 118
3.4.1 Householder Transformations 119
3.4.2 Givens Rotations 121
3.4.3 Roundoff Error Analysis for Orthogonal Matrices 123
3.4.4 Why Orthogonal Matrices? 124
3.5 Rank-Deficient Least Squares Problems 125
3.5.1 Solving Rank-Deficient Least Squares Problems Using the SVD 128
3.5.2 Solving Rank-Deficient Least Squares Problems Using QR with Pivoting 130
3.6 Performance Comparison of Methods for Solving Least Squares Problems 132
3.7 References and Other Topics for Chapter 3 134
3.8 Questions for Chapter 3 134

4 Nonsymmetric Eigenvalue Problems **139**
4.1 Introduction 139
4.2 Canonical Forms 140
4.2.1 Computing Eigenvectors from the Schur Form 148
4.3 Perturbation Theory 148
4.4 Algorithms for the Nonsymmetric Eigenproblem 153
4.4.1 Power Method 154
4.4.2 Inverse Iteration 155
4.4.3 Orthogonal Iteration 156
4.4.4 QR Iteration 159
4.4.5 Making QR Iteration Practical 163
4.4.6 Hessenberg Reduction 164

4.4.7 Tridiagonal and Bidiagonal Reduction 166
4.4.8 QR Iteration with Implicit Shifts 167
4.5 Other Nonsymmetric Eigenvalue Problems 173
4.5.1 Regular Matrix Pencils and Weierstrass Canonical Form 173
4.5.2 Singular Matrix Pencils and the Kronecker Canonical Form 180
4.5.3 Nonlinear Eigenvalue Problems 183
4.6 Summary 184
4.7 References and Other Topics for Chapter 4 187
4.8 Questions for Chapter 4 187

5 The Symmetric Eigenproblem and Singular Value Decomposition 195
5.1 Introduction 195
5.2 Perturbation Theory 197
5.2.1 Relative Perturbation Theory 207
5.3 Algorithms for the Symmetric Eigenproblem 210
5.3.1 Tridiagonal QR Iteration 212
5.3.2 Rayleigh Quotient Iteration 214
5.3.3 Divide-and-Conquer 216
5.3.4 Bisection and Inverse Iteration 228
5.3.5 Jacobi's Method 232
5.3.6 Performance Comparison 235
5.4 Algorithms for the Singular Value Decomposition 237
5.4.1 QR Iteration and Its Variations for the Bidiagonal SVD 242
5.4.2 Computing the Bidiagonal SVD to High Relative Accuracy 245
5.4.3 Jacobi's Method for the SVD 248
5.5 Differential Equations and Eigenvalue Problems 254
5.5.1 The Toda Lattice 255
5.5.2 The Connection to Partial Differential Equations 259
5.6 References and Other Topics for Chapter 5 260
5.7 Questions for Chapter 5 260

6 Iterative Methods for Linear Systems 265
6.1 Introduction 265
6.2 On-line Help for Iterative Methods 266
6.3 Poisson's Equation 267
6.3.1 Poisson's Equation in One Dimension 267
6.3.2 Poisson's Equation in Two Dimensions 270
6.3.3 Expressing Poisson's Equation with Kronecker Products 274
6.4 Summary of Methods for Solving Poisson's Equation 277
6.5 Basic Iterative Methods 279
6.5.1 Jacobi's Method 281
6.5.2 Gauss–Seidel Method 282
6.5.3 Successive Overrelaxation 283

6.5.4 Convergence of Jacobi's, Gauss–Seidel, and SOR(ω) Methods on the Model Problem 285
6.5.5 Detailed Convergence Criteria for Jacobi's, Gauss–Seidel, and SOR(ω) Methods 286
6.5.6 Chebyshev Acceleration and Symmetric SOR (SSOR) . 294
6.6 Krylov Subspace Methods 299
6.6.1 Extracting Information about A via Matrix-Vector Multiplication 301
6.6.2 Solving $Ax = b$ Using the Krylov Subspace $\mathcal{K}_k$ 305
6.6.3 Conjugate Gradient Method 307
6.6.4 Convergence Analysis of the Conjugate Gradient Method 312
6.6.5 Preconditioning 316
6.6.6 Other Krylov Subspace Algorithms for Solving $Ax = b$. 319
6.7 Fast Fourier Transform 321
6.7.1 The Discrete Fourier Transform 323
6.7.2 Solving the Continuous Model Problem Using Fourier Series 324
6.7.3 Convolutions 325
6.7.4 Computing the Fast Fourier Transform 326
6.8 Block Cyclic Reduction 327
6.9 Multigrid 331
6.9.1 Overview of Multigrid on the Two-Dimensional Poisson's Equation 332
6.9.2 Detailed Description of Multigrid on the One-Dimensional Poisson's Equation 337
6.10 Domain Decomposition 347
6.10.1 Nonoverlapping Methods 348
6.10.2 Overlapping Methods 351
6.11 References and Other Topics for Chapter 6 356
6.12 Questions for Chapter 6 356

7 Iterative Methods for Eigenvalue Problems 361
7.1 Introduction 361
7.2 The Rayleigh–Ritz Method 362
7.3 The Lanczos Algorithm in Exact Arithmetic 366
7.4 The Lanczos Algorithm in Floating Point Arithmetic 375
7.5 The Lanczos Algorithm with Selective Orthogonalization 382
7.6 Beyond Selective Orthogonalization 383
7.7 Iterative Algorithms for the Nonsymmetric Eigenproblem . . . 384
7.8 References and Other Topics for Chapter 7 386
7.9 Questions for Chapter 7 386

Bibliography 389

Index 409

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

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$^{\mathrm{TM}}$ [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. $\diamond$

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. $\diamond$

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 $\text{alg}(x)$ is our algorithm for $f(x)$, including the effects of roundoff, we call $\text{alg}(x)$ a *backward stable algorithm* for $f(x)$ if for all x there is a "small" δx such that $\text{alg}(x) = f(x + \delta x)$. 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 $\text{alg}(\cdot)$ 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.

mic decisions automatically for the user, it may perform more slowly than a carefully chosen library routine.

The third programming paradigm is that of *templates*, or recipes for assembling complicated algorithms out of simpler building blocks. Templates are useful when there are a large number of ways to construct an algorithm but no simple rule for choosing the best construction for a particular input problem; therefore, much of the construction must be left to the user. An example of this may be found in *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* [24]; a similar set of templates for eigenproblems is currently under construction.

1.4. Example: Polynomial Evaluation

We illustrate the ideas of perturbation theory, condition numbers, backward stability, and roundoff error analysis with the example of *polynomial evaluation*:

$$p(x) = \sum_{i=0}^{d} a_i x^i.$$

Horner's rule for polynomial evaluation is

$p = a_d$
for $i = d - 1$ down to 0
    $p = x * p + a_i$
end for

Let us apply this to $p(x) = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$. In the bottom of Figure 1.1, we see that near the zero $x = 2$ the value of $p(x)$ computed by Horner's rule is quite unpredictable and may justifiably be called "noise." The top of Figure 1.1 shows an accurate plot.

To understand the implications of this figure, let us see what would happen if we tried to find a zero of $p(x)$ using a simple zero finder based on Bisection, shown below in Algorithm 1.1.

Bisection starts with an interval $[x_{low}, x_{high}]$ in which $p(x)$ changes sign ($p(x_{low}) \cdot p(x_{high}) < 0$) so that $p(x)$ must have a zero in the interval. Then the algorithm computes $p(x_{mid})$ at the interval midpoint $x_{mid} = (x_{low} + x_{high})/2$ and asks whether $p(x)$ changes sign in the bottom half interval $[x_{low}, x_{mid}]$ or top half interval $[x_{mid}, x_{high}]$. Either way, we find an interval of half the original length containing a zero of $p(x)$. We can continue bisecting until the interval is as short as desired.

So the decision between choosing the top half interval or bottom half interval depends on the sign of $p(x_{mid})$. Examining the graph of $p(x)$ in the bottom half of Figure 1.1, we see that this sign varies rapidly from plus to minus as

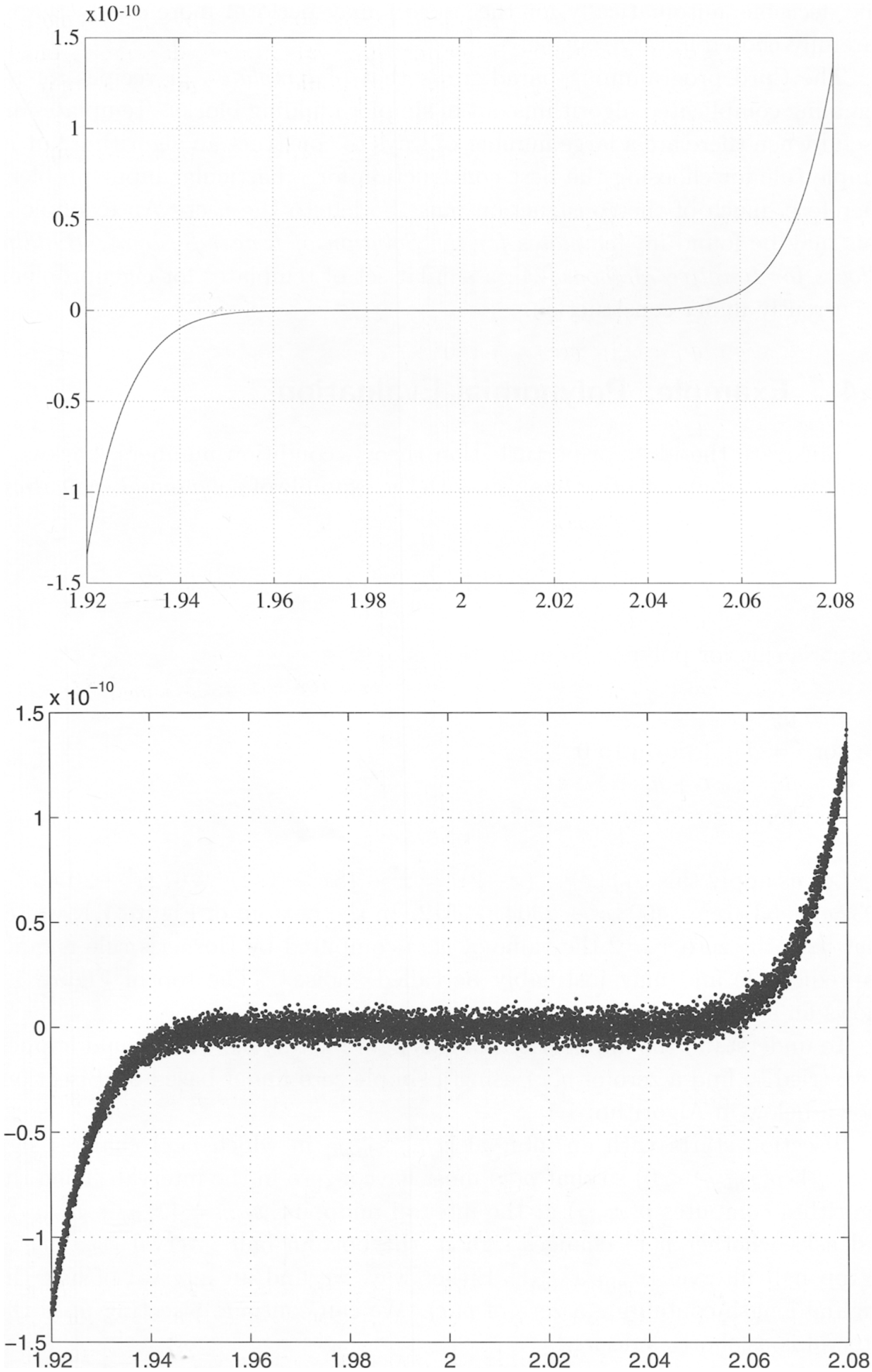


Fig. 1.1. *Plot of* $y = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$ *evaluated at* 8000 *equispaced points, using* $y = (x-2)^9$ *(top) and using Horner's rule (bottom).*

x varies. So changing x_{low} or x_{high} just slightly could completely change the sequence of sign decisions and also the final interval. Indeed, depending on the initial choices of x_{low} and x_{high}, the algorithm could converge *anywhere* inside the "noisy region" from 1.95 to 2.05 (see Question 1.21).

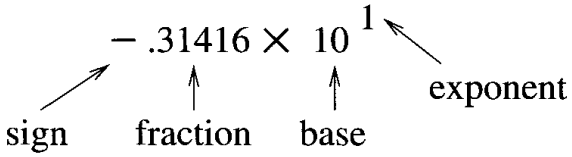
To explain this fully, we return to properties of floating point arithmetic.

ALGORITHM 1.1. *Finding zeros of $p(x)$ using Bisection.*

proc **bisect** $(p, x_{low}, x_{high}, tol)$
/* *find a root of* $p(x) = 0$ *in* $[x_{low}, x_{high}]$
assuming $p(x_{low}) \cdot p(x_{high}) < 0$ */
/* *stop if zero found to within* $\pm tol$ */
$p_{low} = p(x_{low})$
$p_{high} = p(x_{high})$
while $x_{high} - x_{low} > 2 \cdot tol$
$x_{mid} = (x_{low} + x_{high})/2$
$p_{mid} = p(x_{mid})$
if $p_{low} \cdot p_{mid} < 0$ *then /* there is a root in* $[x_{low}, x_{mid}]$ */
$x_{high} = x_{mid}$
$p_{high} = p_{mid}$
else if $p_{mid} \cdot p_{high} < 0$ *then /* there is a root in* $[x_{mid}, x_{high}]$ */
$x_{low} = x_{mid}$
$p_{low} = p_{mid}$
*else /** x_{mid} *is a root */*
$x_{low} = x_{mid}$
$x_{high} = x_{mid}$
end if
end while
$root = (x_{low} + x_{high})/2$

1.5. Floating Point Arithmetic

The number -3.1416 may be expressed in *scientific notation* as follows:



Computers use a similar representation called *floating point*, but generally the base is 2 (with exceptions, such as 16 for IBM 370 and 10 for some spreadsheets and most calculators). For example, $.10101_2 \times 2^3 = 5.25_{10}$.

A floating point number is called *normalized* if the leading digit of the fraction is nonzero. For example, $.10101_2 \times 2^3$ is normalized, but $.010101_2 \times 2^4$ is not. Floating point numbers are usually normalized, which has two advantages:

each nonzero floating point value has a unique representation as a bit string, and in binary the leading 1 in the fraction need not be stored explicitly (because it is always 1), leaving one extra bit for a longer, more accurate fraction.

The most important parameters describing floating point numbers are the base; the number of digits (bits) in the fraction, which determines the precision; and the number of digits (bits) in the exponent, which determines the exponent range and thus the largest and smallest representable numbers. Different floating point arithmetics also differ in how they round computed results, what they do about numbers that are too near zero (underflow) or too big (overflow), whether $\pm\infty$ is allowed, and whether useful nonnumbers (sometimes called NaNs, indefinites, or reserved operands) are provided. We discuss each of these below.

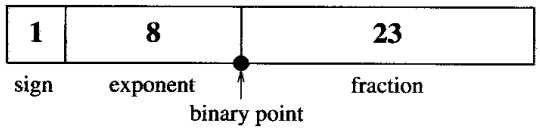
First we consider the precision with which numbers can be represented. For example, $.31416 \times 10^1$ has five decimal digits, so any information less than $.5 \times 10^{-4}$ may have been lost. This means that if x is a real number whose best five-digit approximation is $.31416 \times 10^1$, then the *relative representation error* in $.31416 \times 10^1$ is

$$\frac{|x - .31416 \times 10^1|}{.31416 \times 10^1} \leq \frac{.5 \times 10^{-4}}{.31416 \times 10^1} \approx .16 \times 10^{-4}.$$

The maximum relative representation error in a normalized number occurs for $.10000 \times 10^1$, which is the most accurate five-digit approximation of all numbers in the interval from .999995 to 1.00005. Its relative error is therefore bounded by $.5 \cdot 10^{-4}$. More generally, the *maximum relative representation error* in a floating point arithmetic with p digits and base β is $.5 \times \beta^{1-p}$. This is also half the distance between 1 and the next larger floating point number, $1 + \beta^{1-p}$.

Computers have historically used many different choices of base, number of digits, and range, but fortunately the *IEEE standard for binary arithmetic* is now most common. It is used on Sun, DEC, HP, and IBM workstations and all PCs. IEEE arithmetic includes two kinds of floating point numbers: *single precision* (32 bits long) and *double precision* (64 bits long).

IEEE single precision



If s, e, and $f < 1$ are the 1-bit sign, 8-bit exponent, and 23-bit fraction in the IEEE single precision format, respectively, then the number represented is $(-1)^s \cdot 2^{e-127} \cdot (1 + f)$. The maximum relative representation error is $2^{-24} \approx 6 \cdot 10^{-8}$, and the range of positive normalized numbers is from 2^{-126} (the *underflow threshold*) to $2^{127} \cdot (2 - 2^{-23}) \approx 2^{128}$ (the *overflow threshold*), or about 10^{-38} to 10^{38}. The positions of these floating point numbers on the real

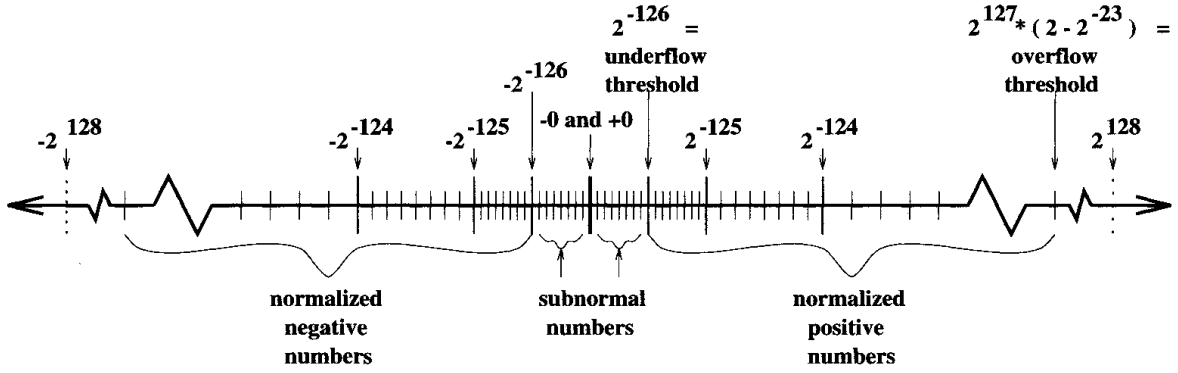
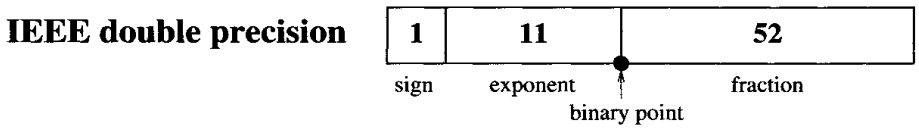


Fig. 1.2. *Real number line with floating point numbers indicated by solid tick marks. The range shown is correct for IEEE single precision, but a 3-bit fraction is assumed for ease of presentation so that there are only* $2^3 - 1 = 7$ *floating point numbers between consecutive powers of* 2, *not* $2^{23} - 1$. *The distance between consecutive tick marks is constant between powers of* 2 *and doubles/halves across powers of* 2 *(among the normalized floating point numbers).* $+2^{128}$ *and* -2^{128}, *which are one unit in the last place larger in magnitude than the overflow threshold (the largest finite floating point number,* $2^{127} \cdot (2 - 2^{-23})$), *are shown as dotted tick marks. The figure is symmetric about* 0; $+0$ *and* -0 *are distinct floating point bit strings but compare as numerically equal. Division by zero is the only binary operation that gives different results,* $+\infty$ *and* $-\infty$, *for different signed zero arguments.*

number line are shown in Figure 1.2 (where we use a 3-bit fraction for ease of presentation).



If s, e, and $f < 1$ are the 1-bit sign, 11-bit exponent, and 52-bit fraction in IEEE double precision format, respectively, then the number represented is $(-1)^s \cdot 2^{e-1023} \cdot (1 + f)$. The maximum relative representation error is $2^{-53} \approx 10^{-16}$, and the exponent range is 2^{-1022} (the *underflow threshold*) to $2^{1023} \cdot (2 - 2^{-52}) \approx 2^{1024}$ (the *overflow threshold*), or about 10^{-308} to 10^{308}.

When the true value of a computation $a \odot b$ (where $\odot$ is one of the four binary operations $+$, $-$, $*$, and $/$) cannot be represented exactly as a floating point number, it must be approximated by a nearby floating point number before it can be stored in memory or a register. We denote this approximation by $\mathrm{fl}(a \odot b)$. The difference $(a \odot b) - \mathrm{fl}(a \odot b)$ is called the *roundoff error*. If $\mathrm{fl}(a \odot b)$ is a nearest floating point number to $a \odot b$, we say that the arithmetic *rounds correctly* (or just *rounds*). IEEE arithmetic has this attractive property. (IEEE arithmetic breaks ties, when $a \odot b$ is exactly halfway between two adjacent floating point numbers, by choosing $\mathrm{fl}(a \odot b)$ to have its least significant bit zero; this is called *rounding to nearest even.*) When rounding correctly, if $a \odot b$ is within the exponent range (otherwise we get *overflow* or *underflow*), then

we can write

$$\mathrm{fl}(a \odot b) = (a \odot b)(1 + \delta), \tag{1.1}$$

where $|\delta|$ is bounded by ε, which is called variously *machine epsilon*, *machine precision*, or *macheps*. Since we are rounding as accurately as possible, ε is equal to the maximum relative representation error $.5 \cdot \beta^{1-p}$. IEEE arithmetic also guarantees that $\mathrm{fl}(\sqrt{a}) = \sqrt{a}(1 + \delta)$, with $|\delta| \leq \varepsilon$. This is the most common model for roundoff error analysis and the one we will use in this book. A nearly identical formula applies to complex floating point arithmetic; see Question 1.12. However, formula (1.1) does ignore some interesting details.

1.5.1. Further Details

IEEE arithmetic also includes *subnormal numbers*, i.e., unnormalized floating point numbers with the minimum possible exponent. These represent tiny numbers between zero and the smallest normalized floating point number; see Figure 1.2. Their presence means that a difference $\mathrm{fl}(x - y)$ can never be zero because of underflow, yielding the attractive property that the predicate $x = y$ is true if and only if $\mathrm{fl}(x - y) = 0$. To incorporate errors caused by underflow into formula (1.1) one would change it to

$$\mathrm{fl}(a \odot b) = (a \odot b)(1 + \delta) + \eta,$$

where $|\delta| \leq \varepsilon$ as before, and $|\eta|$ is bounded by a tiny number equal to the largest error caused by underflow ($2^{-150} \approx 10^{-45}$ in IEEE single precision and $2^{-1075} \approx 10^{-324}$ in IEEE double precision).

IEEE arithmetic includes the symbols $\pm\infty$ and NaN (*Not a Number*). $\pm\infty$ is returned when an operation overflows, and behaves according to the following arithmetic rules: $x/\pm\infty = 0$ for any finite floating point number x, $x/0 = \pm\infty$ for any nonzero floating point number x, $+\infty + \infty = +\infty$, etc. An NaN is returned by any operation with no well-defined finite or infinite result, such as $\infty - \infty$, $\frac{\infty}{\infty}$, $\frac{0}{0}$, $\sqrt{-1}$, $\mathrm{NaN} \odot x$, etc.

Whenever an arithmetic operation is invalid and so produces an NaN, or overflows or divides by zero to produce $\pm\infty$, or underflows, an *exception flag* is set and can later be tested by the user's program. These features permit one to write both more reliable programs (because the program can detect and correct its own exceptions, instead of simply aborting execution) and faster programs (by avoiding "paranoid" programming with many tests and branches to avoid possible but unlikely exceptions). For examples, see Question 1.19, the comments following Lemma 5.3, and [81].

The most expensive error known to have been caused by an improperly handled floating point exception is the crash of the Ariane 5 rocket of the European Space Agency on June 4, 1996. See HOME/ariane5rep.html for details.

Not all machines use IEEE arithmetic or round carefully, although nearly all do. The most important modern exceptions are those machines produced by

Cray Research,[3] although future generations of Cray machines may use IEEE arithmetic.[4] Since the difference between $\mathrm{fl}(a \odot b)$ computed on a Cray machine and $\mathrm{fl}(a \odot b)$ computed on an IEEE machine usually lies in the 14th decimal place or beyond, the reader may wonder whether the difference is important. Indeed, most algorithms in numerical linear algebra are insensitive to details in the way roundoff is handled. But it turns out that some algorithms are easier to design, or more reliable, when rounding is done properly. Here are two examples.

When the Cray C90 subtracts 1 from the next smaller floating point number, it gets -2^{-47}, which is twice the correct answer, -2^{-48}. Getting even tiny differences to high relative accuracy is essential for the correctness of the divide-and-conquer algorithm for finding eigenvalues and eigenvectors of symmetric matrices, currently the fastest algorithm available for the problem. This algorithm requires a rather nonintuitive modification to guarantee correctness on Cray machines (see section 5.3.3).

The Cray machine may also yield an error when computing $\arccos(x/\sqrt{x^2+y^2})$ because excessive roundoff causes the argument of arccos to be larger than 1. This cannot happen in IEEE arithmetic (see Question 1.17).

To accommodate error analysis on a Cray C90 or other Cray machines we may instead use the model $\mathrm{fl}(a \pm b) = a(1+\delta_1) \pm b(1+\delta_2)$, $\mathrm{fl}(a * b) = (a * b)(1+\delta_3)$, and $\mathrm{fl}(a/b) = (a/b)(1+\delta_3)$, with $|\delta_i| \leq \varepsilon$, where ε is a small multiple of the maximum relative representation error.

Briefly, we can say that correct rounding and other features of IEEE arithmetic are designed to preserve as many mathematical relationships used to derive formulas as possible. It is easier to design algorithms knowing that (barring overflow or underflow) $\mathrm{fl}(a-b)$ is computed with a small relative error (otherwise divide-and-conquer can fail), and that $-1 \leq c \equiv \mathrm{fl}(x/\sqrt{x^2+y^2}) \leq 1$ (otherwise $\arccos(c)$ can fail). There are many other such mathematical relationships that one relies on (often unwittingly) to design algorithms. For more details about IEEE arithmetic and its relationship to numerical analysis, see [159, 158, 81].

Given the variability in floating point across machines, how does one write portable software that depends on the arithmetic? For example, iterative algorithms that we will study in later chapters frequently have loops such as

repeat
    ...
    update e
until "e is negligible compared to f,"

[3] We include machines such as the NEC SX-4, which has a "Cray mode" in which it performs arithmetic the same way. We exclude the Cray T3D and T3E, which are parallel computers built from DEC Alpha processors, which use IEEE arithmetic very nearly (underflows are flushed to zero for speed's sake).

[4] Cray Research was purchased by Silicon Graphics in 1996.

where $e \geq 0$ is some error measure, and $f > 0$ is some comparison value (see section 4.4.5 for an example). By negligible we mean "is $e \leq c \cdot \varepsilon \cdot f$?," where $c \geq 1$ is some modest constant, chosen to trade off accuracy and speed of convergence. Since this test requires the machine-dependent constant ε, this test has in the past often been replaced by the *apparently* machine-independent test "is $e + cf = cf$?" The idea here is that adding e to cf and rounding will yield cf again if $e < c\varepsilon f$ or perhaps a little smaller. But this test can fail (by requiring e to be *much* smaller than necessary, or than attainable), depending on the machine and compiler used (see the next paragraph). So the best test indeed uses ε explicitly. It turns out that with sufficient care one can compute ε in a machine-independent way, and software for this is available in the LAPACK subroutines `slamch` (for single precision) and `dlamch` (for double precision). These routines also compute or estimate the overflow threshold (without overflowing!), the underflow threshold, and other parameters. Another portable program that uses these explicit machine parameters is discussed in Question 1.19.

Sometimes one needs higher precision than is available from IEEE single or double precision. For example, higher precision is of use in algorithms such as iterative refinement for improving the accuracy of a computed solution of $Ax = b$ (see section 2.5.1). So IEEE defines another, higher precision called *double extended.* For example, *all* arithmetic operations on an Intel Pentium (or its predecessors going back to the Intel 8086/8087) are performed in 80-bit double extended registers, providing 64-bit fractions and 15-bit exponents. Unfortunately, not all languages and compilers permit one to declare and compute with double-extended precision variables.

Few machines offer anything beyond double-extended arithmetic in hardware, but there are several ways in which more accurate arithmetic may be simulated in software. Some compilers on DEC Vax and DEC Alpha, Sun Sparc, and IBM RS6000 machines permit the user to declare *quadruple precision* (or *real*16* or *double double precision*) variables and to perform computations with them. Since this arithmetic is simulated using shorter precision, it may run several times slower than double. Cray's single precision is similar in precision to IEEE double, and so Cray double precision is about twice IEEE double; it too is simulated in software and runs relatively slowly. There are also algorithms and packages available for simulating much higher precision floating point arithmetic, using either integer arithmetic [20, 21] or the underlying floating point (see Question 1.18) [204, 218].

Finally, we mention *interval arithmetic*, a style of computation that automatically provides guaranteed error bounds. Each variable in an interval computation is represented by a pair of floating point numbers, one a lower bound and one an upper bound. Computation proceeds by rounding in such a way that lower bounds and upper bounds are propagated in a guaranteed fashion. For example, to add the intervals $a = [a_l, a_u]$ and $b = [b_l, b_u]$, one rounds $a_l + b_l$ *down* to the nearest floating point number, c_l, and rounds $a_u + b_u$

up to the nearest floating point number, c_u. This guarantees that the interval $c = [c_l, c_u]$ contains the sum of any pair of variables from a and from b. Unfortunately, if one naively takes a program and converts all floating point variables and operations to interval variables and operations, it is most likely that the intervals computed by the program will quickly grow so wide (such as $[-\infty, +\infty]$) that they provide no useful information at all. (A simple example is to repeatedly compute $x = x - x$ when x is an interval; instead of getting $x = 0$, the width $x_u - x_l$ of x doubles at each subtraction.) It is possible to modify old algorithms or design new ones that do provide useful guaranteed error bounds [4, 140, 162, 190], but these are often several times as expensive as the algorithms discussed in this book. The error bounds that we present in this book are not guaranteed in the same mathematical sense that interval bounds are, but they are reliable enough in almost all situations. (We discuss this in more detail later.) We will not discuss interval arithmetic further in this book.

1.6. Polynomial Evaluation Revisited

Let us now apply roundoff model (1.1) to evaluating a polynomial with Horner's rule. We take the original program,

$p = a_d$
for $i = d - 1$ down to 0
    $p = x \cdot p + a_i$
end for

Then we add subscripts to the intermediate results so that we have a unique symbol for each one (p_0 is the final result):

$p_d = a_d$
for $i = d - 1$ down to 0
    $p_i = x \cdot p_{i+1} + a_i$
end for

Then we insert a roundoff term $(1 + \delta_i)$ at each floating point operation to get

$p_d = a_d$
for $i = d - 1$ down to 0
    $p_i = ((x \cdot p_{i+1})(1 + \delta_i) + a_i)(1 + \delta_i'), \quad$ where $|\delta_i|, |\delta_i'| \leq \varepsilon$
end for

Expanding, we get the following expression for the final computed value of the polynomial:

$$p_0 = \sum_{i=0}^{d-1} \left[(1 + \delta_i') \prod_{j=0}^{i-1} (1 + \delta_j)(1 + \delta_j') \right] a_i x^i + \left[\prod_{j=0}^{d-1} (1 + \delta_j)(1 + \delta_j') \right] a_d x^d .$$

This is messy, a typical result when we try to keep track of every rounding error in an algorithm. We simplify it using the following upper and lower bounds:

$$\begin{aligned}(1+\delta_1)\cdots(1+\delta_j) &\leq (1+\varepsilon)^j \leq \frac{1}{1-j\varepsilon} = 1+j\varepsilon+O(\varepsilon^2),\\ (1+\delta_1)\cdots(1+\delta_j) &\geq (1-\varepsilon)^j \geq 1-j\varepsilon.\end{aligned}$$

These bounds are correct, provided that $j\varepsilon < 1$. Typically, we make the reasonable assumption that $j\varepsilon \ll 1$ ($j \ll 10^7$ in IEEE single precision) and make the approximations

$$1-j\varepsilon \leq (1+\delta_1)\cdots(1+\delta_j) \leq 1+j\varepsilon.$$

This lets us write

$$\begin{aligned}p_0 &= \sum_{i=0}^{d}(1+\overline{\delta}_i)a_i x^i, \quad \text{where } |\overline{\delta}_i| \leq 2d\varepsilon\\ &= \sum_{i=0}^{d}\overline{a}_i x^i\end{aligned}$$

So the computed value p_0 of $p(x)$ is the exact value of a slightly different polynomial with coefficients $\overline{a}_i$. This means that evaluating $p(x)$ is "backward stable," and the "backward error" is $2d\varepsilon$ measured as the maximum relative change of any coefficient of $p(x)$.

Using this backward error bound, we bound the error in the computed polynomial:

$$\begin{aligned}|p_0 - p(x)| &= \left|\sum_{i=0}^{d}(1+\overline{\delta}_i)a_i x^i - \sum_{i=0}^{d}a_i x^i\right|\\ &= \left|\sum_{i=0}^{d}\overline{\delta}_i a_i x^i\right| \leq \sum_{i=0}^{d}\varepsilon 2d|a_i \cdot x^i|\\ &\leq 2d\varepsilon\sum_{i=0}^{d}|a_i \cdot x^i|.\end{aligned}$$

Note that $\sum_i |a_i x^i|$ bounds the largest value that we could compute if there were no cancellation from adding positive and negative numbers, and the error bound is $2d\varepsilon$ times smaller. This is also the case for computing dot products and many other polynomial-like expressions.

By choosing $\overline{\delta}_i = \varepsilon \cdot \text{sign}(a_i x^i)$, we see that the error bound is attainable to within the modest factor $2d$. This means that we may use

$$\frac{\sum_{i=0}^{d}|a_i x^i|}{|\sum_{i=0}^{d}a_i x^i|}$$

as the *relative condition number* for polynomial evaluation.

We can easily compute this error bound, at the cost of doubling the number of operations:

$p = a_d$, $bp = |a_d|$
for $i = d - 1$ down to 0
  $p = x \cdot p + a_i$
  $bp = |x| \cdot bp + |a_i|$
end for
error bound $= bp = 2d \cdot \varepsilon \cdot bp$

so the true value of the polynomial is in the interval $[p - bp, p + bp]$, and the number of guaranteed correct decimal digits is $-\log_{10}(|\frac{bp}{p}|)$. These bounds are plotted in the top of Figure 1.3 for the polynomial discussed earlier, $(x-2)^9$. (The reader may wonder whether roundoff errors could make this computed error bound inaccurate. This turns out not to be a problem and is left to the reader as an exercise.)

The graph of $-\log_{10}|\frac{bp}{p}|$ in the bottom of Figure 1.3, a lower bound on the number of correct decimal digits, indicates that we expect difficulty computing $p(x)$ to high relative accuracy when $p(x)$ is near 0. What is special about $p(x) = 0$? An arbitrarily small error ε in computing $p(x) = 0$ causes an infinite relative error $\frac{\varepsilon}{p(x)} = \frac{\varepsilon}{0}$. In other words, our relative error bound $2d\varepsilon \sum_{i=0}^{d} |a_i x^i| / |\sum_{i=0}^{d} a_i x^i|$ is infinite.

DEFINITION 1.1. *A problem whose condition number is infinite is called* ill-posed. *Otherwise it is called* well-posed.[5]

There is a simple geometric interpretation of the condition number: it tells us how far $p(x)$ is from a polynomial which is ill-posed.

DEFINITION 1.2. *Let* $p(z) = \sum_{i=0}^{d} a_i z^i$ *and* $q(z) = \sum_{i=0}^{d} b_i z^i$. *Define the* relative distance $d(p,q)$ from p to q *as the smallest value satisfying* $|a_i - b_i| \leq d(p,q) \cdot |a_i|$ *for* $0 \leq i \leq d$. *(If all* $a_i \neq 0$, *then we can more simply write* $d(p,q) = \max_{0 \leq i \leq d} |\frac{a_i - b_i}{a_i}|$.)

Note that if $a_i = 0$, then b_i must also be zero for $d(p,q)$ to be finite.

[5]This definition is slightly nonstandard, because ill-posed problems include those whose solutions are continuous as long as they are nondifferentiable. Examples include multiple roots of polynomials and multiple eigenvalues of matrices (section 4.3). Another way to describe an ill-posed problem is one in which the number of correct digits in the solution is not always within a constant of the number of digits used in the arithmetic in the solution. For example, multiple roots of polynomials tend to lose *half* or more of the precision of the arithmetic.

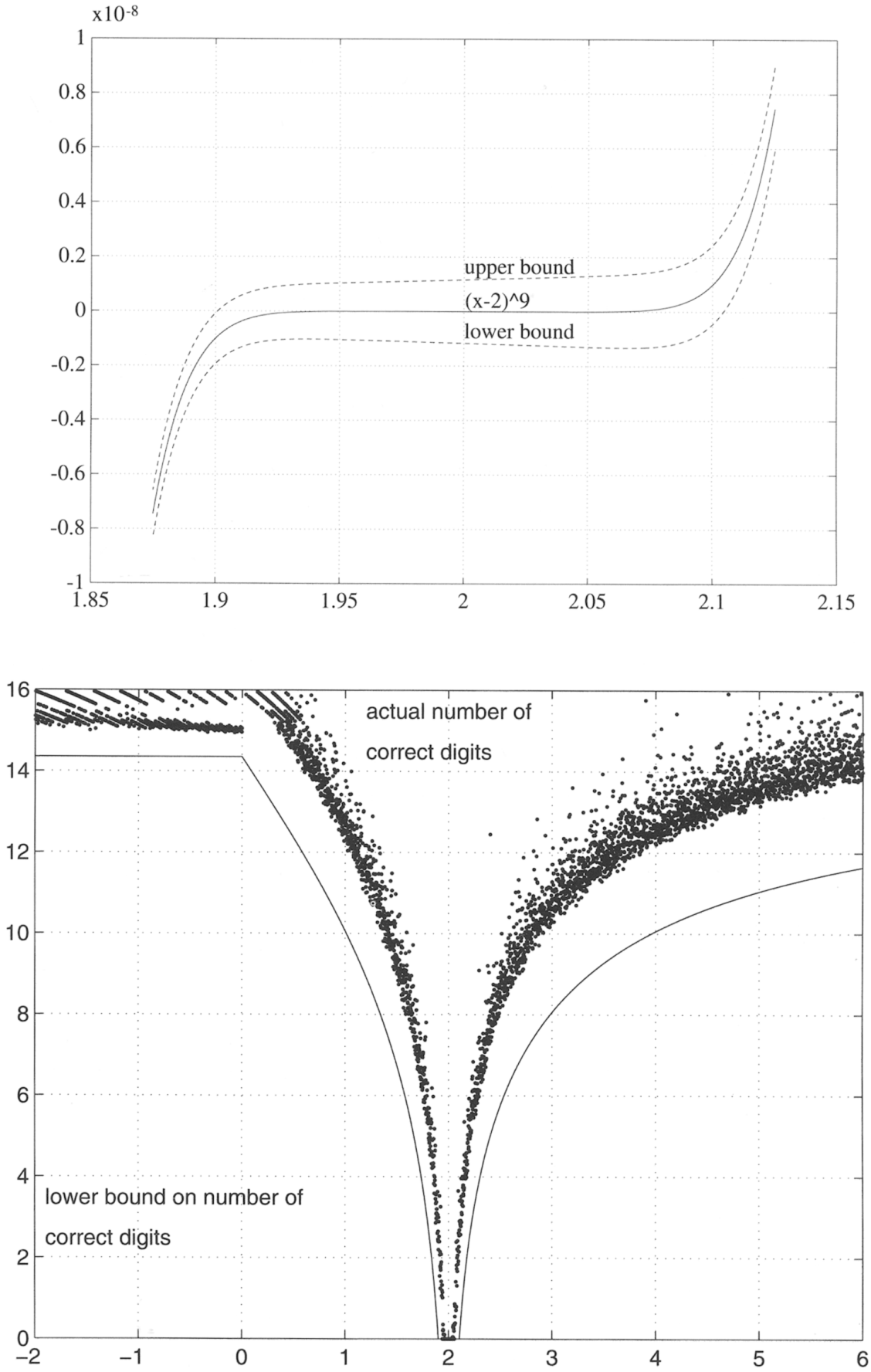


Fig. 1.3. *Plot of error bounds on the value of* $y = (x-2)^9$ *evaluated using Horner's rule.*

THEOREM 1.2. *Suppose that* $p(z) = \sum_{i=0}^{d} a_i z^i$ *is not identically zero.*

$$\min\{d(p,q) \text{ such that } q(x) = 0\} = \frac{|\sum_{i=0}^{d} a_i x^i|}{\sum_{i=0}^{d} |a_i x^i|}.$$

In other words, the distance from p *to the nearest polynomial* q *whose condition number at* x *is infinite, i.e.,* $q(x) = 0$*, is the reciprocal of the condition number of* $p(x)$*.*

Proof. Write $q(z) = \sum b_i z^i = \sum (1+\varepsilon_i) a_i z^i$ so that $d(p,q) = \max_i |\varepsilon_i|$. Then $q(x) = 0$ implies $|p(x)| = |q(x) - p(x)| = |\sum_{i=0}^{d} \varepsilon_i a_i x^i| \le \sum_{i=0}^{d} |\varepsilon_i a_i x^i| \le \max_i |\varepsilon_i| \sum_i |a_i x^i|$, which in turn implies $d(p,q) = \max |\varepsilon_i| \ge |p(x)| / \sum_i |a_i x^i|$. To see that there is a q this close to p, choose

$$\varepsilon_i = \frac{-p(x)}{\sum |a_i x^i|} \cdot \operatorname{sign}(a_i x^i). \quad \square$$

This simple reciprocal relationship between condition number and distance to the nearest ill-posed problem is very common in numerical analysis, and we shall encounter it again later.

At the beginning of the introduction we said that we would use canonical forms of matrices to help solve linear algebra problems. For example, knowing the exact Jordan canonical form makes computing exact eigenvalues trivial. There is an analogous canonical form for polynomials, which makes accurate polynomial evaluation easy: $p(x) = a_d \prod_{i=1}^{d} (x - r_i)$. In other words, we represent the polynomial by its leading coefficient a_d and its roots $r_1, \ldots, r_n$. To evaluate $p(x)$ we use the obvious algorithm

```
p = a_d
for i = 1 to d
    p = p · (x − r_i)
end for
```

It is easy to show the computed $p = p(x) \cdot (1+\delta)$, where $|\delta| \le 2d\varepsilon$; i.e., we always get $p(x)$ with high relative accuracy. But we need the roots of the polynomial to do this!

1.7. Vector and Matrix Norms

Norms are used to measure errors in matrix computations, so we need to understand how to compute and manipulate them.

Missing proofs are left as problems at the end of the chapter.

DEFINITION 1.3. *Let* $\mathcal{B}$ *be a real (complex) linear space* $\mathbb{R}^n$ (or $\mathbb{C}^n$). *It is* normed *if there is a function* $\|\cdot\| : \mathcal{B} \to \mathbb{R}$*, which we call a* norm*, satisfying all of the following:*

1) $\|x\| \geq 0$, *and* $\|x\| = 0$ *if and only if* $x = 0$ (positive definiteness),
2) $\|\alpha x\| = |\alpha| \cdot \|x\|$ *for any real (or complex) scalar* α (homogeneity),
3) $\|x + y\| \leq \|x\| + \|y\|$ (the triangle inequality).

EXAMPLE 1.4. The most common norms are $\|x\|_p = (\sum_i |x_i|^p)^{1/p}$ for $1 \leq p < \infty$, which we call *p-norms*, as well as $\|x\|_\infty = \max_i |x_i|$, which we call the ∞-*norm* or *infinity-norm*. Also, if $\|x\|$ is any norm and C is any nonsingular matrix, then $\|Cx\|$ is also a norm. $\diamond$

We see that there are many norms that we could use to measure errors; it is important to choose an appropriate one. For example, let $x_1 = [1, 2, 3]^T$ in meters and $x_2 = [1.01, 2.01, 2.99]^T$ in meters. Then x_2 is a good approximation to x_1 because the relative error $\frac{\|x_1 - x_2\|_\infty}{\|x_1\|_\infty} \approx .0033$, and $x_3 = [10, 2.01, 2.99]^T$ is a bad approximation because $\frac{\|x_1 - x_3\|_\infty}{\|x_1\|_\infty} = 3$. But suppose the first component is measured in kilometers instead of meters. Then in this norm $\hat{x}_1$ and $\hat{x}_3$ look close:

$$\hat{x}_1 = \begin{bmatrix} .001 \\ 2 \\ 3 \end{bmatrix}, \quad \hat{x}_3 = \begin{bmatrix} .01 \\ 2.01 \\ 2.99 \end{bmatrix}, \quad \text{and} \quad \frac{\|\hat{x}_1 - \hat{x}_3\|_\infty}{\|\hat{x}_1\|_\infty} \approx .0033.$$

To compare $\hat{x}_1$ and $\hat{x}_3$, we should use

$$\|\hat{x}\|_s \equiv \left\| \begin{bmatrix} 1000 & & \\ & 1 & \\ & & 1 \end{bmatrix} \hat{x} \right\|_\infty$$

to make the units the same or so that equally important errors make the norm equally large.

Now we define *inner products*, which are a generalization of the standard *dot product* $\sum_i x_i y_i$, and arise frequently in linear algebra.

DEFINITION 1.4. *Let* $\mathcal{B}$ *be a real (complex) linear space.* $\langle \cdot, \cdot \rangle : \mathcal{B} \times \mathcal{B} \to \mathbb{R}(\mathbb{C})$ *is an* inner product *if all of the following apply:*

1) $\langle x, y \rangle = \langle y, x \rangle$ (*or* $\overline{\langle y, x \rangle}$),
2) $\langle x, y + z \rangle = \langle x, y \rangle + \langle x, z \rangle$,
3) $\langle \alpha x, y \rangle = \alpha \langle x, y \rangle$ *for any real (or complex) scalar* α,
4) $\langle x, x \rangle \geq 0$, *and* $\langle x, x \rangle = 0$ *if and only if* $x = 0$.

EXAMPLE 1.5. Over $\mathbb{R}$, $\langle x, y \rangle = y^T x = \sum_i x_i y_i$, and over $\mathbb{C}$, $\langle x, y \rangle = y^* x = \sum_i x_i \bar{y}_i$ are inner products. (Recall that $y^* = \bar{y}^T$ is the conjugate transpose of y.) $\diamond$

DEFINITION 1.5. x *and* y *are* orthogonal *if* $\langle x, y \rangle = 0$.

The most important property of an inner product is that it satisfies the Cauchy–Schwartz inequality. This can be used in turn to show that $\sqrt{\langle x, x\rangle}$ is a norm, one that we will frequently use.

LEMMA 1.1. Cauchy–Schwartz inequality. $|\langle x, y\rangle| \leq \sqrt{\langle x, x\rangle \cdot \langle y, y\rangle}$.

LEMMA 1.2. $\sqrt{\langle x, x\rangle}$ *is a norm.*

There is a one-to-one correspondence between inner products and *symmetric (Hermitian) positive definite matrices*, as defined below. These matrices arise frequently in applications.

DEFINITION 1.6. *A real symmetric (complex Hermitian) matrix A is* positive definite *if $x^T Ax > 0$ ($x^* Ax > 0$) for all $x \neq 0$. We abbreviate symmetric positive definite to s.p.d., and Hermitian positive to h.p.d.*

LEMMA 1.3. *Let $\mathcal{B} = \mathbb{R}^n$ (or $\mathbb{C}^n$) and $\langle \cdot, \cdot \rangle$ be an inner product. Then there is an n-by-n s.p.d. (h.p.d.) matrix A such that $\langle x, y\rangle = y^T Ax$ ($y^* Ax$). Conversely, if A is s.p.d (h.p.d.), then $y^T Ax$ ($y^* Ax$) is an inner product.*

The following two lemmas are useful in converting error bounds in terms of one norm to error bounds in terms of another.

LEMMA 1.4. *Let $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ be two norms on $\mathbb{R}^n$ (or $\mathbb{C}^n$). There are constants $c_1, c_2 > 0$ such that, for all x, $c_1\|x\|_\alpha \leq \|x\|_\beta \leq c_2\|x\|_\alpha$. We also say that norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ are* equivalent *with respect to constants c_1 and c_2.*

LEMMA 1.5.

$$\begin{array}{rcccl} \|x\|_2 & \leq & \|x\|_1 & \leq & \sqrt{n}\|x\|_2, \\ \|x\|_\infty & \leq & \|x\|_2 & \leq & \sqrt{n}\|x\|_\infty, \\ \|x\|_\infty & \leq & \|x\|_1 & \leq & n\|x\|_\infty. \end{array}$$

In addition to vector norms, we will also need *matrix norms* to measure errors in matrices.

DEFINITION 1.7. *$\|\cdot\|$ is a* matrix norm *on m-by-n matrices if it is a vector norm on $m \cdot n$ dimensional space:*

1) $\|A\| \geq 0$ *and* $\|A\| = 0$ *if and only if* $A = 0$,
2) $\|\alpha A\| = |\alpha| \cdot \|A\|$,
3) $\|A + B\| \leq \|A\| + \|B\|$.

EXAMPLE 1.6. $\max_{ij} |a_{ij}|$ is called the *max norm*, and $(\sum |a_{ij}|^2)^{1/2} = \|A\|_F$ is called the *Frobenius norm*. $\diamond$

The following definition is useful for bounding the norm of a product of matrices, something we often need to do when deriving error bounds.

DEFINITION 1.8. *Let $\|\cdot\|_{m\times n}$ be a matrix norm on m-by-n matrices, $\|\cdot\|_{n\times p}$ be a matrix norm on n-by-p matrices, and $\|\cdot\|_{m\times p}$ be a matrix norm on m-by-p matrices. These norms are called* mutually consistent *if $\|A\cdot B\|_{m\times p} \leq \|A\|_{m\times n}\cdot\|B\|_{n\times p}$, where A is m-by-n and B is n-by-p.*

DEFINITION 1.9. *Let A be m-by-n, $\|\cdot\|_{\hat{m}}$ be a vector norm on $\mathbb{R}^m$, and $\|\cdot\|_{\hat{n}}$ be a vector norm on $\mathbb{R}^n$. Then*

$$\|A\|_{\hat{m}\hat{n}} \equiv \max_{\substack{x\neq 0\\ x\in\mathbb{R}^n}} \frac{\|Ax\|_{\hat{m}}}{\|x\|_{\hat{n}}}$$

is called an operator norm *or* induced norm *or* subordinate matrix norm.

The next lemma provides a large source of matrix norms, ones that we will use for bounding errors.

LEMMA 1.6. *An operator norm is a matrix norm.*

Orthogonal and *unitary matrices*, defined next, are essential ingredients of nearly all our algorithms for least squares problems and eigenvalue problems.

DEFINITION 1.10. *A real square matrix Q is* orthogonal *if $Q^{-1} = Q^T$. A complex square matrix is* unitary *if $Q^{-1} = Q^*$.*

All rows (or columns) of orthogonal (or unitary) matrices have unit 2-norms and are orthogonal to one another, since $QQ^T = Q^TQ = I$ ($QQ^* = Q^*Q = I$).

The next lemma summarizes the essential properties of the norms and matrices we have introduced so far. We will use these properties later in the book.

LEMMA 1.7.

1. *$\|Ax\| \leq \|A\|\cdot\|x\|$ for a vector norm and its corresponding operator norm, or the vector two-norm and matrix Frobenius norm.*
2. *$\|AB\| \leq \|A\|\cdot\|B\|$ for any operator norm or for the Frobenius norm. In other words, any operator norm (or the Frobenius norm) is mutually consistent with itself.*
3. *The max norm and Frobenius norm are not operator norms.*
4. *$\|QAZ\| = \|A\|$ if Q and Z are orthogonal or unitary for the Frobenius norm and for the operator norm induced by $\|\cdot\|_2$. This is really just the Pythagorean theorem.*
5. *$\|A\|_\infty \equiv \max_{x\neq 0}\frac{\|Ax\|_\infty}{\|x\|_\infty} = \max_i \sum_j |a_{ij}|$ = maximum absolute row sum.*

6. $\|A\|_1 \equiv \max_{x\neq 0} \frac{\|Ax\|_1}{\|x\|_1} = \|A^T\|_\infty = \max_j \sum_i |a_{ij}|$ = *maximum absolute column sum.*

7. $\|A\|_2 \equiv \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sqrt{\lambda_{\max}(A^*A)}$, *where* $\lambda_{\max}$ *denotes the largest eigenvalue.*

8. $\|A\|_2 = \|A^T\|_2$.

9. $\|A\|_2 = \max_i |\lambda_i(A)|$ *if* A *is* normal, *i.e.*, $AA^* = A^*A$.

10. *If* A *is* n*-by-*n, *then* $n^{-1/2}\|A\|_2 \leq \|A\|_1 \leq n^{1/2}\|A\|_2$.

11. *If* A *is* n*-by-*n, *then* $n^{-1/2}\|A\|_2 \leq \|A\|_\infty \leq n^{1/2}\|A\|_2$.

12. *If* A *is* n*-by-*n, *then* $n^{-1}\|A\|_\infty \leq \|A\|_1 \leq n\|A\|_\infty$.

13. *If* A *is* n*-by-*n, *then* $\|A\|_1 \leq \|A\|_F \leq n^{1/2}\|A\|_2$.

Proof. We prove part 7 only and leave the rest to Question 1.16.

Since A^*A is Hermitian, there exists an eigendecomposition $A^*A = Q\Lambda Q^*$, with Q a unitary matrix (the columns are eigenvectors), and $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$, a diagonal matrix containing the eigenvalues, which must all be real. Note that all $\lambda_i \geq 0$ since if one, say λ, were negative, we would take q as its eigenvector and get the contradiction $0 \leq \|Aq\|_2^2 = q^TA^TAq = q^T\lambda q = \lambda\|q\|_2^2 < 0$. Therefore

$$\begin{aligned}
\|A\|_2 &= \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \max_{x\neq 0} \frac{(x^*A^*Ax)^{1/2}}{\|x\|_2} = \max_{x\neq 0} \frac{(x^*Q\Lambda Q^*x)^{1/2}}{\|x\|_2} \\
&= \max_{x\neq 0} \frac{((Q^*x)^*\Lambda Q^*x)^{1/2}}{\|Q^*x)\|_2} = \max_{y\neq 0} \frac{(y^*\Lambda y)^{1/2}}{\|y\|_2} = \max_{y\neq 0} \sqrt{\frac{\sum \lambda_i y_i^2}{\sum y_i^2}} \\
&\leq \max_{y\neq 0} \sqrt{\lambda_{\max}} \sqrt{\frac{\sum y_i^2}{\sum y_i^2}} = \sqrt{\lambda_{\max}},
\end{aligned}$$

which is attainable by choosing y to be the appropriate column of the identity matrix. □

1.8. References and Other Topics for Chapter 1

At the end of each chapter we will list the references most relevant to that chapter. They are also listed alphabetically in the bibliography at the end. In addition we will give pointers to related topics not discussed in the main text.

The most modern comprehensive work in this area is by G. Golub and C. Van Loan [121], which also has an extensive bibliography. A recent undergraduate level or beginning graduate text in this material is by D. Watkins [252]. Another good graduate text is by L. Trefethen and D. Bau [243]. A classic

work that is somewhat dated but still an excellent reference is by J. Wilkinson [262]. An older but still excellent book at the same level as Watkins is by G. Stewart [235].

More detailed information on error analysis can be found in the recent book by N. Higham [149]. Older but still good general references are by J. Wilkinson [261] and W. Kahan [157].

"What every computer scientist should know about floating point arithmetic" by D. Goldberg is a good recent survey [119]. IEEE arithmetic is described formally in [11, 12, 159] as well as in the reference manuals published by computer manufacturers. Discussion of error analysis with IEEE arithmetic may be found in [54, 70, 159, 158] and the references cited therein.

A more general discussion of condition numbers and the distance to the nearest ill-posed problem is given by the author in [71] as well as in a series of papers by S. Smale and M. Shub [219, 220, 221, 222]. Vector and matrix norms are discussed at length in [121, sects. 2.2, 2.3].

1.9. Questions for Chapter 1

QUESTION 1.1. *(Easy; Z. Bai)* Let A be an orthogonal matrix. Show that $\det(A) = \pm 1$. Show that if B also is orthogonal and $\det(A) = -\det(B)$, then $A + B$ is singular.

QUESTION 1.2. *(Easy; Z. Bai)* The *rank* of a matrix is the dimension of the space spanned by its columns. Show that A has rank one if and only if $A = ab^T$ for some column vectors a and b.

QUESTION 1.3. *(Easy; Z. Bai)* Show that if a matrix is orthogonal and triangular, then it is diagonal. What are its diagonal elements?

QUESTION 1.4. *(Easy; Z. Bai)* A matrix is *strictly upper triangular* if it is upper triangular with zero diagonal elements. Show that if A is strictly upper triangular and n-by-n, then $A^n = 0$.

QUESTION 1.5. *(Easy; Z. Bai)* Let $\|\cdot\|$ be a vector norm on $\mathbb{R}^m$ and assume that $C \in \mathbb{R}^{m \times n}$. Show that if $\operatorname{rank}(C) = n$, then $\|x\|_C \equiv \|Cx\|$ is a vector norm.

QUESTION 1.6. *(Easy; Z. Bai)* Show that if $0 \neq s \in \mathbb{R}^n$ and $E \in \mathbb{R}^{n \times n}$, then

$$\left\| E\left(I - \frac{ss^{\mathrm{T}}}{s^T s}\right)\right\|_F^2 = \|E\|_F^2 - \frac{\|Es\|_2^2}{s^T s}.$$

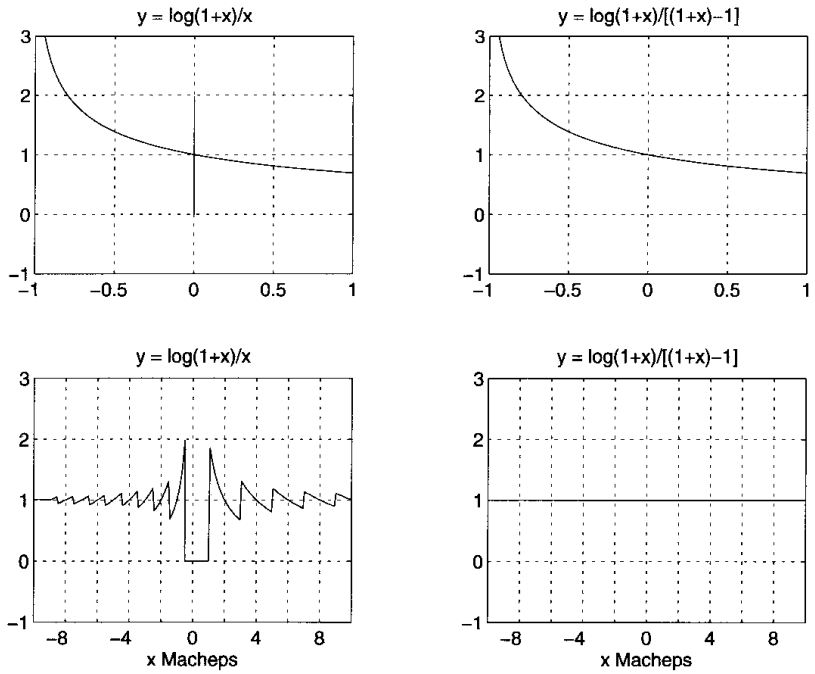
QUESTION 1.7. *(Easy; Z. Bai)* Verify that $\|xy^H\|_F = \|xy^H\|_2 = \|x\|_2\|y\|_2$ for any $x, y \in \mathbf{C}^n$.

QUESTION 1.8. *(Medium)* One can identify the degree d polynomials $p(x) = \sum_{i=0}^{d} a_i x^i$ with $\mathbb{R}^{d+1}$ via the vector of coefficients. Let x be fixed. Let S_x be the set of polynomials with an infinite relative condition number with respect to evaluating them at x (i.e., they are zero at x). In a few words, describe S_x geometrically as a subset of $\mathbb{R}^{d+1}$. Let $S_x(\kappa)$ be the set of polynomials whose relative condition number is κ or greater. Describe $S_x(\kappa)$ geometrically in a few words. Describe how $S_x(\kappa)$ changes geometrically as $\kappa \to \infty$.

QUESTION 1.9. *(Medium)* Consider the figure below. It plots the function $y = \log(1+x)/x$ computed in two different ways. Mathematically, y is a smooth function of x near $x = 0$, equaling 1 at 0. But if we compute y using this formula, we get the plots on the left (shown in the ranges $x \in [-1, 1]$ on the top left and $x \in [-10^{-15}, 10^{-15}]$ on the bottom left). This formula is clearly unstable near $x = 0$. On the other hand, if we use the algorithm

$d = 1 + x$
if $d = 1$ then
    $y = 1$
else
    $y = \log(d)/(d-1)$
end if

we get the two plots on the right, which are correct near $x = 0$. Explain this phenomenon, proving that the second algorithm must compute an accurate answer in floating point arithmetic. Assume that the log function returns an accurate answer for any argument. (This is true of any reasonable implementation of logarithm.) Assume IEEE floating point arithmetic if that makes your argument easier. (Both algorithms can malfunction on a Cray machine.)



QUESTION 1.10. *(Medium)* Show that, barring overflow or underflow, $\mathrm{fl}(\sum_{i=1}^{d} x_i y_i) = \sum_{i=1}^{d} x_i y_i(1+\delta_i)$, where $|\delta_i| \leq d\varepsilon$. Use this to prove the following fact. Let $A^{m\times n}$ and $B^{n\times p}$ be matrices, and compute their product in the usual way. Barring overflow or underflow show that $|\mathrm{fl}(A\cdot B) - A\cdot B| \leq n\cdot\varepsilon\cdot|A|\cdot|B|$. Here the absolute value of a matrix $|A|$ means the matrix with entries $(|A|)_{ij} = |a_{ij}|$, and the inequality is meant componentwise.

The result of this question will be used in section 2.4.2, where we analyze the roundoff errors in Gaussian elimination.

QUESTION 1.11. *(Medium)* Let L be a lower triangular matrix and solve $Lx = b$ by forward substitution. Show that barring overflow or underflow, the computed solution $\hat{x}$ satisfies $(L+\delta L)\hat{x} = b$, where $|\delta l_{ij}| \leq n\varepsilon|l_{ij}|$, where ε is the machine precision. This means that forward substitution is backward stable. Argue that backward substitution for solving upper triangular systems satisfies the same bound.

The result of this question will be used in section 2.4.2, where we analyze the roundoff errors in Gaussian elimination.

QUESTION 1.12. *(Medium)* In order to analyze the effects of rounding errors, we have used the following model (see equation (1.1)):

$$\mathrm{fl}(a \odot b) = (a \odot b)(1+\delta),$$

where $\odot$ is one of the four basic operations $+$, $-$, $*$, and $/$, and $|\delta| \leq \varepsilon$. To show that our analyses also work for *complex* data, we need to prove an analogous formula for the four basic complex operations. Now δ will be a tiny *complex* number bounded in absolute value by a small multiple of ϵ. Prove that this is true for complex addition, subtraction, multiplication, and division. Your algorithm for complex division should successfully compute $a/a \approx 1$, where $|a|$ is either very large (larger than the square root of the overflow threshold) or very small (smaller than the square root of the underflow threshold). Is it true that both the real and imaginary parts of the complex product are always computed to high relative accuracy?

QUESTION 1.13. *(Medium)* Prove Lemma 1.3.

QUESTION 1.14. *(Medium)* Prove Lemma 1.5.

QUESTION 1.15. *(Medium)* Prove Lemma 1.6.

QUESTION 1.16. *(Medium)* Prove all parts except 7 of Lemma 1.7. Hint for part 8: Use the fact that if X and Y are both n-by-n, then XY and YX have the same eigenvalues. Hint for part 9: Use the fact that a matrix is normal if [1] only if it has a complete set of orthonormal eigenvectors.

QUESTION 1.17. *(Hard; W. Kahan)* We mentioned that on a Cray machine the expression $\arccos(x/\sqrt{x^2+y^2})$ caused an error, because roundoff caused $(x/\sqrt{x^2+y^2})$ to exceed 1. Show that this is impossible using IEEE arithmetic, barring overflow or underflow. Hint: You will need to use more than the simple model $\mathrm{fl}(a \odot b) = (a \odot b)(1+\delta)$ with $|\delta|$ small. Think about evaluating $\sqrt{x^2}$, and show that, barring overflow or underflow, $\mathrm{fl}(\sqrt{x^2}) = x$ *exactly*; in numerical experiments done by A. Liu, this failed about 5% of the time on a Cray YMP. You might try some numerical experiments and explain them. Extra credit: Prove the same result using correctly rounded *decimal* arithmetic. (The proof is different.) This question is due to W. Kahan, who was inspired by a bug in a Cray program of J. Sethian.

QUESTION 1.18. *(Hard)* Suppose that a and b are normalized IEEE double precision floating point numbers, and consider the following algorithm, running with IEEE arithmetic:

if $(|a| < |b|)$, swap a and b
$s_1 = a + b$
$s_2 = (a - s_1) + b$

Prove the following facts:

1. Barring overflow or underflow, the only roundoff error committed in running the algorithm is computing $s_1 = fl(a+b)$. In other words, both subtractions $s_1 - a$ and $(s_1 - a) - b$ are computed *exactly.*

2. $s_1 + s_2 = a + b$, *exactly.* This means that s_2 is actually the roundoff error committed when rounding the exact value of $a + b$ to get s_1.

Thus, this program in effect simulates *quadruple* precision arithmetic, representing the true sum $a + b$ as the higher-order bits (s_1) and the lower-order bits (s_2).

Using this and similar tricks in a systematic way, it is possible to efficiently simulate all four basic floating point operations in *arbitrary* precision arithmetic, using only the underlying floating point instructions and no "bit-fiddling" [204]. 128-bit arithmetic is implemented this way on the IBM RS6000 and Cray (but much less efficiently on the Cray, which does not have IEEE arithmetic).

QUESTION 1.19. *(Hard; Programming)* This question illustrates the challenges in engineering highly reliable numerical software. Your job is to write a program to compute the two-norm $s \equiv \|x\|_2 = (\sum_{i=1}^{n} x_i^2)^{1/2}$ given $x_1, \ldots, x_n$. The most obvious (and inadequate) algorithm is

$s = 0$
for $i = 1$ to n
    $s = s + x_i^2$

endfor
$s = \text{sqrt}(s)$

This algorithm is inadequate because it does not have the following desirable properties:

1. It must compute the answer accurately (i.e., nearly all the computed digits must be correct) unless $\|x\|_2$ is (nearly) outside the range of normalized floating point numbers.

2. It must be nearly as fast as the obvious program above in most cases.

3. It must work on any "reasonable" machine, possibly including ones not running IEEE arithmetic. This means it may not cause an error condition, unless $\|x\|_2$ is (nearly) larger than the largest floating point number.

To illustrate the difficulties, note that the obvious algorithm fails when $n = 1$ and x_1 is larger than the square root of the largest floating point number (in which case x_1^2 overflows, and the program returns $+\infty$ in IEEE arithmetic and halts in most non-IEEE arithmetics) or when $n = 1$ and x_1 is smaller than the square root of the smallest normalized floating point number (in which case x_1^2 underflows, possibly to zero, and the algorithm may return zero). Scaling the x_i by dividing them all by $\max_i |x_i|$ does not have property 2), because division is usually many times more expensive than either multiplication or addition. Multiplying by $c = 1/\max_i |x_i|$ risks overflow in computing c, even when $\max_i |x_i| > 0$.

This routine is important enough that it has been standardized as a *Basic Linear Algebra Subroutine*, or *BLAS*, which should be available on all machines [169]. We discuss the BLAS at length in section 2.6.1, and documentation and sample implementations may be found at NETLIB/blas. In particular, see NETLIB/cgi-bin/netlibget.pl/blas/snrm2.f for a sample implementation that has properties 1) and 3) but not 2). These sample implementations are intended to be starting points for implementations specialized to particular architectures (an easier problem than producing a completely portable one, as requested in this problem). Thus, when writing your own numerical software, you should think of computing $\|x\|_2$ as a building block that should be available in a numerical library on each machine.

For another careful implementation of $\|x\|_2$, see [35].

You can extract test code from NETLIB/blas/sblat1 to see if your implementation is correct; all implementations turned in must be thoroughly tested as well as timed, with times compared to the obvious algorithm above on those cases where both run. See how close to satisfying the three conditions you can come; the frequent use of the word "nearly" in conditions (1), (2) and (3) shows where you may compromise in attaining one condition in order to more

nearly attain another. In particular, you might want to see how much easier the problem is if you limit yourself to machines running IEEE arithmetic.

Hint: Assume that the values of the overflow and underflow thresholds are available for your algorithm. Portable software for computing these values is available (see NETLIB/cgi-bin/netlibget.pl/lapack/util/slamch.f).

QUESTION 1.20. *(Easy; Medium)* We will use a Matlab program to illustrate how sensitive the roots of polynomial can be to small perturbations in the coefficients. The program is available[6] at HOMEPAGE/Matlab/polyplot.m. Polyplot takes an input polynomial specified by its roots r and then adds random perturbations to the polynomial coefficients, computes the perturbed roots, and plots them. The inputs are

r = vector of roots of the polynomial,
e = maximum relative perturbation to make to each coefficient of the polynomial,
m = number of random polynomials to generate, whose roots are plotted.

1. *(Easy)* The first part of your assignment is to run this program for the following inputs. In all cases choose m high enough that you get a fairly dense plot but don't have to wait too long. m = a few hundred or perhaps 1000 is enough. You may want to change the axes of the plot if the graph is too small or too large.
   - r=(1:10); e = 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8,
   - r=(1:20); e = 1e-9, 1e-11, 1e-13, 1e-15,
   - r=[2,4,8,16,..., 1024]; e=1e-1, 1e-2, 1e-3, 1e-4.

   Also try your own example with complex conjugate roots. Which roots are most sensitive?

2. *(Medium)* The second part of your assignment is to modify the program to compute the condition number c(i) for each root. In other words, a relative perturbation of e in each coefficient should change root r(i) by at most about e*c(i). Modify the program to plot circles centered at r(i) with radii e*c(i), and confirm that these circles enclose the perturbed roots (at least when e is small enough that the linearization used to derive the condition number is accurate). You should turn in a few plots with circles and perturbed eigenvalues, and some explanation of what you observe.

3. *(Medium)* In the last part, notice that your formula for c(i) "blows up" if $p'(\mathrm{r(i)}) = 0$. This condition means that r(i) is a *multiple root* of $p(x) = 0$. We can still expect some accuracy in the computed value of a multiple

[6]Recall that we abbreviate the URL prefix of the class homepage to HOMEPAGE in the text.

root, however, and in this part of the question, we will ask how sensitive a multiple root can be: First, write $p(x) = q(x) \cdot (x - \mathrm{r(i)})^m$, where $q(\mathrm{r(i)}) \neq 0$ and m is the multiplicity of the root r(i). Then compute the m roots nearest r(i) of the slightly perturbed polynomial $p(x) - q(x)\epsilon$, and show that they differ from r(i) by $|\epsilon|^{1/m}$. So that if $m = 2$, for instance, the root r(i) is perturbed by $\epsilon^{1/2}$, which is much larger than ϵ if $|\epsilon| \ll 1$. Higher values of m yield even larger perturbations. If ϵ is around machine epsilon and represents rounding errors in computing the root, this means an m-tuple root can lose all but $1/m$-th of its significant digits.

QUESTION 1.21. *(Medium)* Apply Algorithm 1.1, Bisection, to find the roots of $p(x) = (x - 2)^9 = 0$, where $p(x)$ is evaluated using Horner's rule. Use the Matlab implementation in HOMEPAGE/Matlab/bisect.m, or else write your own. Confirm that changing the input interval slightly changes the computed root drastically. Modify the algorithm to use the error bound discussed in the text to stop bisecting when the roundoff error in the computed value of $p(x)$ gets so large that its sign cannot be determined.

2

Linear Equation Solving

2.1. Introduction

This chapter discusses perturbation theory, algorithms, and error analysis for solving the linear equation $Ax = b$. The algorithms are all variations on Gaussian elimination. They are called *direct methods*, because in the absence of roundoff error they would give the exact solution of $Ax = b$ after a finite number of steps. In contrast, Chapter 6 discusses *iterative methods*, which compute a sequence $x_0, x_1, x_2, \ldots$ of ever better approximate solutions of $Ax = b$; one stops iterating (computing the next x_{i+1}) when x_i is accurate enough. Depending on the matrix A and the speed with which x_i converges to $x = A^{-1}b$, a direct method or an iterative method may be faster or more accurate. We will discuss the relative merits of direct and iterative methods at length in Chapter 6. For now, we will just say that direct methods are the methods of choice when the user has no special knowledge about the source[7] of matrix A or when a solution is required with guaranteed stability and in a guaranteed amount of time.

The rest of this chapter is organized as follows. Section 2.2 discusses perturbation theory for $Ax = b$; it forms the basis for the practical error bounds in section 2.4. Section 2.3 derives the Gaussian elimination algorithm for dense matrices. Section 2.4 analyzes the errors in Gaussian elimination and presents practical error bounds. Section 2.5 shows how to improve the accuracy of a solution computed by Gaussian elimination, using a simple and inexpensive iterative method. To get high speed from Gaussian elimination and other linear algebra algorithms on contemporary computers, care must be taken to organize the computation to respect the computer memory organization; this is discussed in section 2.6. Finally, section 2.7 discusses faster variations of Gaussian elimination for matrices with special properties commonly arising in practice, such as symmetry ($A = A^T$) or sparsity (when many entries of A are zero).

[7]For example, in Chapter 6 we consider the case when A arises from approximating the solution to a particular differential equation, Poisson's equation.

Sections 2.2.1 and 2.5.1 discuss recent innovations upon which the software in the LAPACK library depends.

There are a variety of open problems, which we shall mention as we go along.

2.2. Perturbation Theory

Suppose $Ax = b$ and $(A + \delta A)\hat{x} = b + \delta b$; our goal is to bound the norm of $\delta x \equiv \hat{x} - x$. Later, $\hat{x}$ will be the computed solution of $Ax = B$. We simply subtract these two equalities and solve for δx: one way to do this is to take

$$\begin{array}{rrcl} & (A+\delta A)(x+\delta x) & = & b+\delta b \\ - & [Ax & = & b] \\ \hline & \delta Ax + (A+\delta A)\delta x & = & \delta b \end{array}$$

and rearrange to get

$$\delta x = A^{-1}(-\delta A\hat{x} + \delta b). \tag{2.1}$$

Taking norms and using part 1 of Lemma 1.7 as well as the triangle inequality for vector norms, we get

$$\|\delta x\| \leq \|A^{-1}\|(\|\delta A\| \cdot \|\hat{x}\| + \|\delta b\|). \tag{2.2}$$

(We have assumed that the vector norm and matrix norm are consistent, as defined in section 1.7. For example, any vector norm and its induced matrix norm will do.) We can further rearrange this inequality to get

$$\frac{\|\delta x\|}{\|\hat{x}\|} \leq \|A^{-1}\| \cdot \|A\| \cdot \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\| \cdot \|\hat{x}\|}\right). \tag{2.3}$$

The quantity $\kappa(A) = \|A^{-1}\| \cdot \|A\|$ is the *condition number*[8] of the matrix A, because it measures the relative change $\frac{\|\delta x\|}{\|\hat{x}\|}$ in the answer as a multiple of the relative change $\frac{\|\delta A\|}{\|A\|}$ in the data. (To be rigorous, we need to show that inequality (2.2) is an equality for some nonzero choice of δA and δb; otherwise $\kappa(A)$ would only be an upper bound on the condition number. See Question 2.3.) The quantity multiplying $\kappa(A)$ will be small if δA and δb are small, yielding a small upper bound on the relative error $\frac{\|\delta x\|}{\|\hat{x}\|}$.

The upper bound depends on δx (via $\hat{x}$), which makes it seem hard to interpret, but it is actually quite useful in practice, since we know the computed solution $\hat{x}$ and so can straightforwardly evaluate the bound. We can also derive a theoretically more attractive bound that does not depend on δx as follows:

[8]More pedantically, it is the condition number with respect to the problem of matrix inversion. The problem of finding the eigenvalues of A, for example, has a different condition number.

LEMMA 2.1. *Let* $\|\cdot\|$ *satisfy* $\|AB\| \le \|A\| \cdot \|B\|$. *Then* $\|X\| < 1$ *implies that* $I - X$ *is invertible,* $(I-X)^{-1} = \sum_{i=0}^{\infty} X^i$, *and* $\|(I-X)^{-1}\| \le \frac{1}{1-\|X\|}$.

Proof. The sum $\sum_{i=0}^{\infty} X^i$ is said to converge if and only if it converges in each component. We use the fact (from applying Lemma 1.4 to Example 1.6) that for any norm, there is a constant c such that $|x_{jk}| \le c \cdot \|X\|$. We then get $|(X^i)_{jk}| \le c \cdot \|X^i\| \le c \cdot \|X\|^i$, so each component of $\sum X^i$ is dominated by a convergent geometric series $\sum c\|X\|^i = \frac{c}{1-\|X\|}$ and must converge. Therefore $S_n = \sum_{i=0}^{n} X^i$ converges to some S as $n \to \infty$, and $(I-X)S_n = (I-X)(I + X + X^2 + \cdots + X^n) = I - X^{n+1} \to I$ as $n \to \infty$, since $\|X^i\| \le \|X\|^i \to 0$. Therefore $(I-X)S = I$ and $S = (I-X)^{-1}$. The final bound is $\|(I-X)^{-1}\| = \|\sum_{i=0}^{\infty} X^i\| \le \sum_{i=0}^{\infty} \|X^i\| \le \sum_{i=0}^{\infty} \|X\|^i = \frac{1}{1-\|X\|}$. □

Solving our first equation $\delta A x + (A + \delta A)\delta x = \delta b$ for δx yields

$$\begin{aligned} \delta x &= (A+\delta A)^{-1}(-\delta A x + \delta b) \\ &= [A(I + A^{-1}\delta A)]^{-1}(-\delta A x + \delta b) \\ &= (I + A^{-1}\delta A)^{-1} A^{-1}(-\delta A x + \delta b). \end{aligned}$$

Taking norms, dividing both sides by $\|x\|$, using part 1 of Lemma 1.7 and the triangle inequality, and assuming that δA is small enough so that $\|A^{-1}\delta A\| \le \|A^{-1}\| \cdot \|\delta A\| < 1$, we get the desired bound:

$$\begin{aligned} \frac{\|\delta x\|}{\|x\|} &\le \|(I + A^{-1}\delta A)^{-1}\| \cdot \|A^{-1}\| \left(\|\delta A\| + \frac{\|\delta b\|}{\|x\|} \right) \\ &\le \frac{\|A^{-1}\|}{1 - \|A^{-1}\| \cdot \|\delta A\|} \left(\|\delta A\| + \frac{\|\delta b\|}{\|x\|} \right) \quad \text{by Lemma 2.1} \\ &= \frac{\|A^{-1}\| \cdot \|A\|}{1 - \|A^{-1}\| \cdot \|A\| \frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\| \cdot \|x\|} \right) \\ &\le \frac{\kappa(A)}{1 - \kappa(A)\frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right) \qquad (2.4) \\ &\quad \text{since } \|b\| = \|Ax\| \le \|A\| \cdot \|x\|. \end{aligned}$$

This bound expresses the relative error $\frac{\|\delta x\|}{\|x\|}$ in the solution as a multiple of the relative errors $\frac{\|\delta A\|}{\|A\|}$ and $\frac{\|\delta b\|}{\|b\|}$ in the input. The multiplier, $\kappa(A)/(1 - \kappa(A)\frac{\|\delta A\|}{\|A\|})$, is close to the condition number $\kappa(A)$ if $\|\delta A\|$ is small enough.

The next theorem explains more about the assumption that $\|A^{-1}\| \cdot \|\delta A\| = \kappa(A) \cdot \frac{\|\delta A\|}{\|A\|} < 1$: it guarantees that $A + \delta A$ is nonsingular, which we need for δx to exist. It also establishes a geometric characterization of the condition number.

THEOREM 2.1. *Let* A *be nonsingular. Then*

$$\min \left\{ \frac{\|\delta A\|_2}{\|A\|_2} : A + \delta A \text{ singular} \right\} = \frac{1}{\|A^{-1}\|_2 \cdot \|A\|_2} = \frac{1}{\kappa(A)}.$$

Therefore, the distance to the nearest singular matrix (ill-posed problem) $= \frac{1}{\text{condition number}}$.

Proof. It is enough to show $\min \{\|\delta A\|_2 : A + \delta A \text{ singular}\} = \frac{1}{\|A^{-1}\|_2}$.

To show this minimum is at least $\frac{1}{\|A^{-1}\|_2}$, note that if $\|\delta A\|_2 < \frac{1}{\|A^{-1}\|_2}$, then $1 > \|\delta A\|_2 \cdot \|A^{-1}\|_2 \geq \|A^{-1}\delta A\|_2$, so Lemma 2.1 implies that $I + A^{-1}\delta A$ is invertible, and so $A + \delta A$ is invertible.

To show the minimum equals $\frac{1}{\|A^{-1}\|_2}$, we construct a δA of norm $\frac{1}{\|A^{-1}\|_2}$ such that $A + \delta A$ is singular. Note that since $\|A^{-1}\|_2 = \max_{x \neq 0} \frac{\|A^{-1}x\|_2}{\|x\|_2}$, there exists an x such that $\|x\|_2 = 1$ and $\|A^{-1}\|_2 = \|A^{-1}x\|_2 > 0$. Now let $y = \frac{A^{-1}x}{\|A^{-1}x\|_2} = \frac{A^{-1}x}{\|A^{-1}\|_2}$ so $\|y\|_2 = 1$. Let $\delta A = \frac{-xy^T}{\|A^{-1}\|_2}$.

Then

$$\|\delta A\|_2 = \max_{z \neq 0} \frac{\|xy^T z\|_2}{\|A^{-1}\|_2 \, \|z\|_2} = \max_{z \neq 0} \frac{|y^T z|}{\|z\|_2} \frac{\|x\|_2}{\|A^{-1}\|_2} = \frac{1}{\|A^{-1}\|_2},$$

where the maximum is attained when z is any nonzero multiple of y, and $A+\delta A$ is singular because

$$(A + \delta A)y = Ay - \frac{xy^T y}{\|A^{-1}\|_2} = \frac{x}{\|A^{-1}\|_2} - \frac{x}{\|A^{-1}\|_2} = 0 \,. \quad \square$$

We have now seen that the distance to the nearest ill-posed problem equals the reciprocal of the condition number for two problems: polynomial evaluation and linear equation solving. This reciprocal relationship is quite common in numerical analysis [71].

Here is a slightly different way to do perturbation theory for $Ax = b$; we will need it to derive practical error bounds later in section 2.4.4. If $\hat{x}$ is any vector, we can bound the difference $\delta x \equiv \hat{x} - x = \hat{x} - A^{-1}b$ as follows. We let $r = A\hat{x} - b$ be the *residual* of $\hat{x}$; the residual r is zero if $\hat{x} = x$. This lets us write $\delta x = A^{-1}r$, yielding the bound

$$\|\delta x\| = \|A^{-1}r\| \leq \|A^{-1}\| \cdot \|r\|. \tag{2.5}$$

This simple bound is attractive to use in practice, since r is easy to compute, given an approximate solution $\hat{x}$. Furthermore, there is no apparent need to estimate δA and δb. In fact our two approaches are very closely related, as shown by the next theorem.

THEOREM 2.2. *Let $r = A\hat{x} - b$. Then there exists a δA such that $\|\delta A\| = \frac{\|r\|}{\|\hat{x}\|}$ and $(A + \delta A)\hat{x} = b$. No δA of smaller norm and satisfying $(A + \delta A)\hat{x} = b$ exists. Thus, δA is the smallest possible backward error (measured in norm). This is true for any vector norm and its induced norm (or $\|\cdot\|_2$ for vectors and $\|\cdot\|_F$ for matrices).*

Proof. $(A+\delta A)\hat{x} = b$ if and only if $\delta A \cdot \hat{x} = b - A\hat{x} = -r$, so $\|r\| = \|\delta A \cdot \hat{x}\| \leq \|\delta A\| \cdot \|\hat{x}\|$, implying $\|\delta A\| \geq \frac{\|r\|}{\|\hat{x}\|}$. We complete the proof only for the two-norm and its induced matrix norm. Choose $\delta A = \frac{-r \cdot \hat{x}^T}{\|\hat{x}\|_2^2}$. We can easily verify that $\delta A \cdot \hat{x} = -r$ and $\|\delta A\|_2 = \frac{\|r\|_2}{\|\hat{x}\|_2}$. □

Thus, the smallest $\|\delta A\|$ that could yield an $\hat{x}$ satisfying $(A+\delta A)\hat{x} = b$ and $r = A\hat{x} - b$ is given by Theorem 2.2. Applying error bound (2.2) (with $\delta b = 0$) yields

$$\|\delta x\| \leq \|A^{-1}\| \left(\frac{\|r\|}{\|\hat{x}\|} \cdot \|\hat{x}\| \right) = \|A^{-1}\| \cdot \|r\|,$$

the same bound as (2.5).

All our bounds depend on the ability to estimate the condition number $\|A\| \cdot \|A^{-1}\|$. We return to this problem in section 2.4.3. Condition number estimates are computed by LAPACK routines such as `sgesvx`.

2.2.1. Relative Perturbation Theory

In the last section we showed how to bound the norm of the error $\delta x = \hat{x} - x$ in the approximate solution $\hat{x}$ of $Ax = b$. Our bound on $\|\delta x\|$ was proportional to the condition number $\kappa(A) = \|A\| \cdot \|A^{-1}\|$ times the norms $\|\delta A\|$ and $\|\delta b\|$, where $\hat{x}$ satisfies $(A + \delta A)\hat{x} = b + \delta b$.

In many cases this bound is quite satisfactory, but not always. Our goal in this section is to show when it is too pessimistic and to derive an alternative perturbation theory that provides tighter bounds. We will use this perturbation theory later in section 2.5.1 to justify the error bounds computed by the LAPACK subroutines like `sgesvx`.

This section may be skipped on a first reading.

Here is an example where the error bound of the last section is much too pessimistic.

EXAMPLE 2.1. Let $A = \mathrm{diag}(\gamma, 1)$ (a diagonal matrix with entries $a_{11} = \gamma$ and $a_{22} = 1$) and $b = [\gamma, 1]^T$, where $\gamma > 1$. Then $x = A^{-1}b = [1, 1]^T$. Any reasonable direct method will solve $Ax = b$ very accurately (using two divisions b_i/a_{ii}) to get $\hat{x}$, yet the condition number $\kappa(A) = \gamma$ may be arbitrarily large. Therefore our error bound (2.3) may be arbitrarily large.

The reason that the condition number $\kappa(A)$ leads us to overestimate the error is that bound (2.2), from which it comes, assumes that δA is bounded in norm *but is otherwise arbitrary*; this is needed to prove that bound (2.2) is attainable in Question 2.3. In contrast, the δA corresponding to the actual rounding errors is not arbitrary but has a special structure not captured by its norm alone. We can determine the smallest δA corresponding to $\hat{x}$ for our problem as follows: A simple rounding error analysis shows that $\hat{x}_i = (b_i/a_{ii})/(1+\delta_i)$, where $|\delta_i| \leq \varepsilon$. Thus $(a_{ii} + \delta_i a_{ii})\hat{x}_i = b_i$. We may rewrite this

as $(A + \delta A)\hat{x} = b$, where $\delta A = \mathrm{diag}(\delta_1 a_{11}, \delta_2 a_{22})$. Then $\|\delta A\|$ can be as large $\max_i |\varepsilon a_{ii}| = \varepsilon\gamma$. Applying error bound (2.3) with $\delta b = 0$ yields

$$\frac{\|\delta x\|_\infty}{\|\hat{x}\|_\infty} \leq \gamma \left(\frac{\varepsilon\gamma}{\gamma}\right) = \varepsilon\gamma.$$

In contrast, the actual error satisfies

$$\begin{aligned} \|\delta x\|_\infty &= \|\hat{x} - x\|_\infty \\ &= \left\| \begin{bmatrix} (b_1/a_{11})/(1+\delta_1) - (b_1/a_{11}) \\ (b_2/a_{22})/(1+\delta_2) - (b_2/a_{22}) \end{bmatrix} \right\|_\infty \\ &= \left\| \begin{bmatrix} -\delta_1/(1+\delta_1) \\ -\delta_2/(1+\delta_2) \end{bmatrix} \right\|_\infty \\ &\leq \frac{\varepsilon}{1-\varepsilon} \end{aligned}$$

or

$$\frac{\|\delta x\|_\infty}{\|\hat{x}\|_\infty} \leq \varepsilon/(1-\varepsilon)^2,$$

which is about γ times smaller. $\diamond$

For this example, we can describe the structure of the actual δA as follows: $|\delta a_{ij}| \leq \epsilon |a_{ij}|$, where ϵ is a tiny number. We write this more succinctly as

$$|\delta A| \leq \epsilon |A| \tag{2.6}$$

(see section 1.1 for notation). We also say that δA is a *small componentwise relative perturbation in A*. Since δA can often be made to satisfy bound (2.6) in practice, along with $|\delta b| \leq \epsilon |b|$ (see section 2.5.1), we will derive perturbation theory using these bounds on δA and δb.

We begin with equation (2.1):

$$\delta x = A^{-1}(-\delta A \hat{x} + \delta b).$$

Now take absolute values, and repeatedly use the triangle inequality to get

$$\begin{aligned} |\delta x| &= |A^{-1}(-\delta A \hat{x} + \delta b)| \\ &\leq |A^{-1}|(|\delta A| \cdot |\hat{x}| + |\delta b|) \\ &\leq |A^{-1}|(\epsilon |A| \cdot |\hat{x}| + \epsilon |b|) \\ &= \epsilon(|A^{-1}|(|A| \cdot |\hat{x}| + |b|)). \end{aligned}$$

Now using any vector norm (like the infinity-, one-, or Frobenius norms), where $\| \, |z| \, \| = \|z\|$, we get the bound

$$\|\delta x\| \leq \epsilon \| |A^{-1}|(|A| \cdot |\hat{x}| + |b|) \|. \tag{2.7}$$

Assuming for the moment that $\delta b = 0$, we can weaken this bound to

$$\|\delta x\| \leq \epsilon \| |A^{-1}| \cdot |A| \| \cdot \|\hat{x}\|$$

or

$$\frac{\|\delta x\|}{\|x\|} \leq \epsilon \| \, |A^{-1}| \cdot |A| \, \|. \tag{2.8}$$

This leads us to define $\kappa_{CR}(A) \equiv \| \, |A^{-1}| \cdot |A| \, \|$ as the *componentwise relative condition number of* A, or just *relative condition number* for short. It is sometimes also called the Bauer condition number [26] or Skeel condition number [225, 226, 227]. For a proof that bounds (2.7) and (2.8) are attainable, see Question 2.4.

Recall that Theorem 2.1 related the condition number $\kappa(A)$ to the distance from A to the nearest singular matrix. For a similar interpretation of $\kappa_{CR}(A)$, see [72, 208].

EXAMPLE 2.2. Consider our earlier example with $A = \text{diag}(\gamma, 1)$ and $b = [\gamma, 1]^T$. It is easy to confirm that $\kappa_{CR}(A) = 1$, since $|A^{-1}| \cdot |A| = I$. Indeed, $\kappa_{CR}(A) = 1$ for any diagonal matrix A, capturing our intuition that a diagonal system of equations should be solvable quite accurately. $\diamond$

More generally, suppose that D is any nonsingular diagonal matrix and B is an arbitrary nonsingular matrix. Then

$$\begin{aligned} \kappa_{CR}(DB) &= \| \, |(DB)^{-1}| \cdot |(DB)| \, \| \\ &= \| \, |B^{-1}D^{-1}| \cdot |DB| \, \| \\ &= \| \, |B^{-1}| \cdot |B| \, \| \\ &= \kappa_{CR}(B). \end{aligned}$$

This means that if DB is *badly scaled*, i.e., B is well-conditioned but DB is badly conditioned (because D has widely varying diagonal entries), then we should hope to get an accurate solution of $(DB)x = b$ despite DB's ill-conditioning. This is discussed further in sections 2.4.4, 2.5.1, and 2.5.2.

Finally, as in the last section we provide an error bound using only the residual $r = A\hat{x} - b$:

$$\|\delta x\| = \|A^{-1}r\| \leq \| \, |A^{-1}| \cdot |r| \, \|, \tag{2.9}$$

where we have used the triangle inequality. In section 2.4.4 we will see that this bound can sometimes be much smaller than the similar bound (2.5), in particular when A is badly scaled. There is also an analogue to Theorem 2.2 [193].

THEOREM 2.3. *The smallest* $\epsilon > 0$ *such that there exist* $|\delta A| \leq \epsilon |A|$ *and* $|\delta b| \leq \epsilon |b|$ *satisfying* $(A+\delta A)\hat{x} = b+\delta b$ *is called the* componentwise relative backward error. *It may be expressed in terms of the residual* $r = A\hat{x} - b$ *as follows:*

$$\epsilon = \max_i \frac{|r_i|}{(|A| \cdot |\hat{x}| + |b|)_i}.$$

For a proof, see Question 2.5.

LAPACK routines like `sgesvx` compute the componentwise backward relative error ϵ (the LAPACK variable name for ϵ is `BERR`).

2.3. Gaussian Elimination

The basic algorithm for solving $Ax = b$ is *Gaussian elimination.* To state it, we first need to define a *permutation matrix.*

DEFINITION 2.1. *A* permutation matrix P *is an identity matrix with permuted rows.*

The most important properties of a permutation matrix are given by the following lemma.

LEMMA 2.2. *Let P, P_1, and P_2 be n-by-n permutation matrices and X be an n-by-n matrix. Then*

1. *PX is the same as X with its rows permuted. XP is the same as X with its columns permuted.*
2. $P^{-1} = P^T$.
3. $\det(P) = \pm 1$.
4. *$P_1 \cdot P_2$ is also a permutation matrix.*

For a proof, see Question 2.6.

Now we can state our overall algorithm for solving $Ax = b$.

ALGORITHM 2.1. *Solving $Ax = b$ using Gaussian elimination:*

1. *Factorize A into $A = PLU$, where*
   P = *permutation matrix,*
   L = *unit lower triangular matrix (i.e., with ones on the diagonal),*
   U = *nonsingular upper triangular matrix.*
2. *Solve $PLUx = b$ for LUx by permuting the entries of b: $LUx = P^{-1}b = P^Tb$.*
3. *Solve $LUx = P^{-1}b$ for Ux by forward substitution: $Ux = L^{-1}(P^{-1}b)$.*
4. *Solve $Ux = L^{-1}(P^{-1}b)$ for x by back substitution: $x = U^{-1}(L^{-1}P^{-1}b)$.*

We will derive the algorithm for factorizing $A = PLU$ in several ways. We begin by showing why the permutation matrix P is necessary.

DEFINITION 2.2. *The* leading j-by-j principal submatrix *of* A *is* $A(1:j, 1:j)$.

THEOREM 2.4. *The following two statements are equivalent:*

1. *There exists a unique unit lower triangular* L *and nonsingular upper triangular* U *such that* $A = LU$.
2. *All leading principal submatrices of* A *are nonsingular.*

Proof. We first show (1) implies (2). $A = LU$ may also be written

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix},$$

where A_{11} is a j-by-j leading principal submatrix, as are L_{11} and U_{11}. Therefore $\det A_{11} = \det(L_{11}U_{11}) = \det L_{11} \det U_{11} = 1 \cdot \prod_{k=1}^{j} (U_{11})_{kk} \neq 0$, since L is unit triangular and U is triangular.

We prove that (2) implies (1) by induction on n. It is easy for 1-by-1 matrices: $a = 1 \cdot a$. To prove it for n-by-n matrices $\tilde{A}$, we need to find unique $(n-1)$-by-$(n-1)$ triangular matrices L and U, unique $(n-1)$-by-1 vectors l and u, and a unique nonzero scalar η such that

$$\tilde{A} = \begin{bmatrix} A & b \\ c^T & \delta \end{bmatrix} = \begin{bmatrix} L & 0 \\ l^T & 1 \end{bmatrix} \begin{bmatrix} U & u \\ 0 & \eta \end{bmatrix} = \begin{bmatrix} LU & Lu \\ l^T U & l^T u + \eta \end{bmatrix}.$$

By induction, unique L and U exist such that $A = LU$. Now let $u = L^{-1}b$, $l^T = c^T U^{-1}$, and $\eta = \delta - l^T u$, all of which are unique. The diagonal entries of U are nonzero by induction, and $\eta \neq 0$ since $0 \neq \det(\tilde{A}) = \det(U) \cdot \eta$. □

Thus LU factorization without pivoting can fail on (well-conditioned) nonsingular matrices such as the permutation matrix

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix};$$

the 1-by-1 and 2-by-2 leading principal minors of P are singular. So we need to introduce permutations into Gaussian elimination.

THEOREM 2.5. *If* A *is nonsingular, then there exist permutations* P_1 *and* P_2, *a unit lower triangular matrix* L, *and a nonsingular upper triangular matrix* U *such that* $P_1 A P_2 = LU$. *Only one of* P_1 *and* P_2 *is necessary.*

Note: $P_1 A$ *reorders the rows of* A, AP_2 *reorders the columns, and* $P_1 A P_2$ *reorders both.*

Proof. As with many matrix factorizations, it suffices to understand block 2-by-2 matrices. More formally, we use induction on the dimension n. It is easy for 1-by-1 matrices: $P_1 = P_2 = L = 1$ and $U = A$. Assume that it is true for dimension $n-1$. If A is nonsingular, then it has a nonzero entry; choose permutations P_1' and P_2' so that the $(1,1)$ entry of $P_1'AP_2'$ is nonzero. (We need only one of P_1' and P_2' since nonsingularity implies that each row and each column of A has a nonzero entry.)

Now we write the desired factorization and solve for the unknown components:

$$\begin{aligned} P_1'AP_2' &= \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \cdot \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{A}_{22} \end{bmatrix} \\ &= \begin{bmatrix} u_{11} & U_{12} \\ L_{21}u_{11} & L_{21}U_{12} + \tilde{A}_{22} \end{bmatrix}, \end{aligned} \tag{2.10}$$

where A_{22} and $\tilde{A}_{22}$ are $(n-1)$-by-$(n-1)$ and L_{21} and U_{12}^T are $(n-1)$-by-1.

Solving for the components of this 2-by-2 block factorization we get $u_{11} = a_{11} \neq 0$, $U_{12} = A_{12}$, and $L_{21}u_{11} = A_{21}$. Since $u_{11} = a_{11} \neq 0$, we can solve for $L_{21} = \frac{A_{21}}{a_{11}}$. Finally, $L_{21}U_{12} + \tilde{A}_{22} = A_{22}$ implies $\tilde{A}_{22} = A_{22} - L_{21}U_{12}$.

We want to apply induction to $\tilde{A}_{22}$, but to do so we need to check that $\det \tilde{A}_{22} \neq 0$: Since $\det P_1'AP_2' = \pm \det A \neq 0$ and also

$$\det P_1'AP_2' = \det \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \cdot \det \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{A}_{22} \end{bmatrix} = 1 \cdot (u_{11} \cdot \det \tilde{A}_{22}),$$

then $\det \tilde{A}_{22}$ must be nonzero.

Therefore, by induction there exist permutations $\tilde{P}_1$ and $\tilde{P}_2$ so that $\tilde{P}_1\tilde{A}_{22}\tilde{P}_2 = \tilde{L}\tilde{U}$, with $\tilde{L}$ unit lower triangular and $\tilde{U}$ upper triangular and nonsingular. Substituting this in the above 2-by-2 block factorization yields

$$\begin{aligned} P_1'AP_2' &= \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{P}_1^T\tilde{L}\tilde{U}\tilde{P}_2^T \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_1^T\tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{U}\tilde{P}_2^T \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ L_{21} & \tilde{P}_1^T\tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12}\tilde{P}_2 \\ 0 & \tilde{U} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_2^T \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_1^T \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \tilde{P}_1L_{21} & \tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12}\tilde{P}_2 \\ 0 & \tilde{U} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_2^T \end{bmatrix}, \end{aligned}$$

so we get the desired factorization of A:

$$\begin{aligned} P_1AP_2 &= \left(\begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_1 \end{bmatrix} P_1' \right) A \left(P_2' \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_2 \end{bmatrix} \right) \\ &= \begin{bmatrix} 1 & 0 \\ \tilde{P}_1L_{21} & \tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12}\tilde{P}_2 \\ 0 & \tilde{U} \end{bmatrix}. \quad \square \end{aligned}$$

The next two corollaries state simple ways to choose P_1 and P_2 to guarantee that Gaussian elimination will succeed on a nonsingular matrix.

COROLLARY 2.1. *We can choose $P_2' = I$ and P_1' so that a_{11} is the largest entry in absolute value in its column, which implies $L_{21} = \frac{A_{21}}{a_{11}}$ has entries bounded by 1 in absolute value. More generally, at step i of Gaussian elimination, where we are computing the ith column of L, we reorder rows i through n so that the largest entry in the column is on the diagonal. This is called "Gaussian elimination with partial pivoting," or GEPP for short. GEPP guarantees that all entries of L are bounded by one in absolute value.*

GEPP is the most common way to implement Gaussian elimination in practice. We discuss its numerical stability in the next section. Another more expensive way to choose P_1 and P_2 is given by the next corollary. It is almost never used in practice, although there are rare examples where GEPP fails but the next method succeeds in computing an accurate answer (see Question 2.14). We discuss briefly it in the next section as well.

COROLLARY 2.2. *We can choose P_1' and P_2' so that a_{11} is the largest entry in absolute value in the whole matrix. More generally, at step i of Gaussian elimination, where we are computing the ith column of L, we reorder rows and columns i through n so that the largest entry in this submatrix is on the diagonal. This is called "Gaussian elimination with complete pivoting," or GECP for short.*

The following algorithm embodies Theorem 2.5, performing permutations, computing the first column of L and the first row of U, and updating A_{22} to get $\tilde{A}_{22} = A_{22} - L_{21}U_{12}$. We write the algorithm first in conventional programming language notation and then using Matlab notation.

ALGORITHM 2.2. *LU factorization with pivoting:*

for $i = 1$ *to* $n-1$
  apply permutations so $a_{ii} \neq 0$ *(permute L and U too)*
    / for example, for GEPP, swap rows j and i of A and of L*
      where $|a_{ji}|$ is the largest entry in $|A(i:n,i)|$;
      for GECP, swap rows j and i of A and of L,
      and columns k and i of A and of U,
      *where $|a_{jk}|$ is the largest entry in $|A(i:n,i:n)|$ */*
  / compute column i of L (L_{21} in (2.10)) */*
  for $j = i+1$ *to* n
    $l_{ji} = a_{ji}/a_{ii}$
  end for
  / compute row i of U (U_{12} in (2.10)) */*
  for $j = i$ *to* n

```
        u_ij = a_ij
    end for
    /* update A_22 (to get Ã_22 = A_22 − L_21 U_12 in (2.10)) */
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk − l_ji * u_ik
        end for
    end for
end for
```

Note that once column i of A is used to compute column i of L, it is never used again. Similarly, row i of A is never used again after computing row i of U. This lets us overwrite L and U on top of A as they are computed, so we need no extra space to store them; L occupies the (strict) lower triangle of A (the ones on the diagonal of L are not stored explicitly), and U occupies the upper triangle of A. This simplifies the algorithm to the following algorithm.

ALGORITHM 2.3. *LU factorization with pivoting, overwriting L and U on A:*

```
for i = 1 to n − 1
    apply permutations (see Algorithm 2.2 for details)
    for j = i + 1 to n
        a_ji = a_ji / a_ii
    end for
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk − a_ji * a_ik
        end for
    end for
end for
```

Using Matlab notation this further reduces to the following algorithm.

ALGORITHM 2.4. *LU factorization with pivoting, overwriting L and U on A:*

```
for i = 1 to n − 1
    apply permutations (see Algorithm 2.2 for details)
    A(i + 1 : n, i) = A(i + 1 : n, i)/A(i, i)
    A(i + 1 : n, i + 1 : n) =
        A(i + 1 : n, i + 1 : n) − A(i + 1 : n, i) * A(i, i + 1 : n)
end for
```

In the last line of the algorithm, $A(i+1:n,i)*A(i,i+1:n)$ is the product of an $(n-i)$-by-1 matrix (L_{21}) by a 1-by-$(n-i)$ matrix (U_{12}), which yields an $(n-i)$-by-$(n-i)$ matrix.

We now rederive this algorithm from scratch starting from perhaps the most familiar description of Gaussian elimination: "Take each row and subtract multiples of it from later rows to zero out the entries below the diagonal." Translating this directly into an algorithm yields

```
for i = 1 to n - 1                  /* for each row i */
    for j = i + 1 to n              /* subtract a multiple of
                                       row i from row j ... */
        for k = i to n              /* ... in columns i through n ... */
            a_jk = a_jk - (a_ji/a_ii) a_ik   /* ... to zero out column i
                                                below the diagonal */
        end for
    end for
end for
```

We will now make some improvements to this algorithm, modifying it until it becomes identical to Algorithm 2.3 (except for pivoting, which we omit). First, we recognize that we need not compute the zero entries below the diagonal, because we know they are zero. This shortens the k loop to yield

```
for i = 1 to n - 1
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk - (a_ji/a_ii) a_ik
        end for
    end for
end for
```

The next performance improvement is to compute $\frac{a_{ji}}{a_{ii}}$ outside the inner loop, since it is constant within the inner loop.

```
for i = 1 to n - 1
    for j = i + 1 to n
        l_ji = a_ji/a_ii
    end for
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk - l_ji a_ik
        end for
    end for
end for
```

Finally, we store the multipliers l_{ji} in the subdiagonal entries a_{ji} that we originally zeroed out; they are not needed for anything else. This yields Algorithm 2.3 (except for pivoting).

The operation count of LU is done by replacing loops by summations over the same range, and inner loops by their operation counts:

$$\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1 + \sum_{j=i+1}^{n}\sum_{k=i+1}^{n} 2\right)$$
$$= \sum_{i=1}^{n-1}((n-i) + 2(n-i)^2) = \frac{2}{3}n^3 + O(n^2).$$

The forward and back substitutions with L and U to complete the solution of $Ax = b$ cost $O(n^2)$, so overall solving $Ax = b$ with Gaussian elimination costs $\frac{2}{3}n^3 + O(n^2)$ operations. Here we have used the fact that $\sum_{i=1}^{m} i^k = m^{k+1}/(k+1) + O(m^k)$. This formula is enough to get the high-order term in the operation count.

There is more to implementing Gaussian elimination than writing the nested loops of Algorithm 2.2. Indeed, depending on the computer, programming language, and matrix size, merely interchanging the last two loops on j and k can change the execution time by orders of magnitude. We discuss this at length in section 2.6.

2.4. Error Analysis

Recall our two-step paradigm for obtaining error bounds for the solution of $Ax = b$:

1. Analyze roundoff errors to show that the result of solving $Ax = b$ is the exact solution $\hat{x}$ of the perturbed linear system $(A+\delta A)\hat{x} = b+\delta b$, where δA and δb are small. This is an example of *backward error analysis*, and δA and δb are called the *backward errors.*

2. Apply the perturbation theory of section 2.2 to bound the error, for example by using bound (2.3) or (2.5).

We have two goals in this section. The first is to show how to implement Gaussian elimination in order to keep the backward errors δA and δb small. In particular, we would like to keep $\frac{\|\delta A\|}{\|A\|}$ and $\frac{\|\delta b\|}{\|b\|}$ as small as $O(\varepsilon)$. This is as small as we can expect to make them, since merely rounding the largest entries of A (or b) to fit into the floating point format can make $\frac{\|\delta A\|}{\|A\|} \geq \varepsilon$ (or $\frac{\|\delta b\|}{\|b\|} \geq \varepsilon$). It turns out that unless we are careful about pivoting, δA and δb need not be small. We discuss this in the next section.

The second goal is to derive practical error bounds which are simultaneously cheap to compute and "tight," i.e., close to the true errors. It turns out that the best bounds for $\|\delta A\|$ that we can formally prove are generally much larger than the errors encountered in practice. Therefore, our practical error bounds

(in section 2.4.4) will rely on the computed residual $r = A\hat{x} - b$ and bound (2.5), instead of bound (2.3). We also need to be able to estimate $\kappa(A)$ inexpensively; this is discussed in section 2.4.3.

Unfortunately, we do not have error bounds that *always* satisfy our twin goals of cheapness and tightness, i.e., that simultaneously

1. cost a negligible amount compared to solving $Ax = b$ in the first place (for example, that cost $O(n^2)$ flops versus Gaussian elimination's $O(n^3)$ flops),

2. provide an error bound that is always at least as large as the true error and never more than a constant factor larger (100 times larger, say).

The practical bounds in section 2.4.4 will cost $O(n^2)$ but will on very rare occasions provide error bounds that are much too small or much too large. The probability of getting a bad error bound is so small that these bounds are widely used in practice. The only truly guaranteed bounds use either interval arithmetic, very high precision arithmetic, or both, and are several times more expensive than just solving $Ax = b$ (see section 1.5).

It has in fact been conjectured that no bound satisfying our twin goals of cheapness and tightness exist, but this remains an open problem.

2.4.1. The Need for Pivoting

Let us apply LU factorization without pivoting to $A = \begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix}$ in three-decimal-digit floating point arithmetic and see why we get the wrong answer. Note that $\kappa(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty \approx 4$, so A is well conditioned and thus we should expect to be able to solve $Ax = b$ accurately.

$$
\begin{aligned}
L &= \begin{bmatrix} 1 & 0 \\ \mathrm{fl}(1/10^{-4}) & 1 \end{bmatrix}, \ \mathrm{fl}(1/10^{-4}) \text{ rounds to } 10^4, \\
U &= \begin{bmatrix} 10^{-4} & 1 \\ & \mathrm{fl}(1 - 10^4 \cdot 1) \end{bmatrix}, \ \mathrm{fl}(1 - 10^4 \cdot 1) \text{ rounds to } -10^4, \\
\text{so } LU &= \begin{bmatrix} 1 & 0 \\ 10^4 & 1 \end{bmatrix} \begin{bmatrix} 10^{-4} & 1 \\ & -10^4 \end{bmatrix} = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 0 \end{bmatrix} \\
\text{but } A &= \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix}.
\end{aligned}
$$

Note that the original a_{22} has been entirely "lost" from the computation by subtracting 10^4 from it. We would have gotten the same LU factors whether a_{22} had been 1, 0, -2, or any number such that $\mathrm{fl}(a_{22} - 10^4) = -10^4$. Since the algorithm proceeds to work only with L and U, it will get the same answer for all these different a_{22}, which correspond to completely different A and so completely different $x = A^{-1}b$; there is no way to guarantee an accurate answer. This is called *numerical instability*, since L and U are *not* the exact

factors of a matrix close to A. (Another way to say this is that $\|A - LU\|$ is about as large as $\|A\|$, rather than $\varepsilon\|A\|$.)

Let us see what happens when we go on to solve $Ax = [1,2]^T$ for x using this LU factorization. The correct answer is $x \approx [1,1]^T$. Instead we get the following. Solving $Ly = [1,2]^T$ yields $y_1 = \mathrm{fl}(1/1) = 1$ and $y_2 = \mathrm{fl}(2 - 10^4 \cdot 1) = -10^4$; note that the value 2 has been "lost" by subtracting 10^4 from it. Solving $U\hat{x} = y$ yields $\hat{x}_2 = \mathrm{fl}((-10^4)/(-10^4)) = 1$ and $\hat{x}_1 = \mathrm{fl}((1-1)/10^{-4}) = 0$, a completely erroneous solution.

Another warning of the loss of accuracy comes from comparing the condition number of A to the condition numbers of L and U. Recall that we transform the problem of solving $Ax = b$ into solving two other systems with L and U, so we do not want the condition numbers of L or U to be much larger than that of A. But here, the condition number of A is about 4, whereas the condition numbers of L and U are about 10^8.

In the next section we will show that doing GEPP nearly always eliminates the instability just illustrated. In the above example, GEPP would have reversed the order of the two equations before proceeding. The reader is invited to confirm that in this case we would get

$$L = \begin{bmatrix} 1 & 0 \\ \mathrm{fl}(.0001/1) & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ .0001 & 1 \end{bmatrix}$$

and

$$U = \begin{bmatrix} 1 & 1 \\ 0 & \mathrm{fl}(1 - .0001 \cdot 1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

so that LU approximates A quite accurately. Both L and U are quite well-conditioned, as is A. The computed solution vector is also quite accurate.

2.4.2. Formal Error Analysis of Gaussian Elimination

Here is the intuition behind our error analysis of LU decomposition. If intermediate quantities arising in the product $L \cdot U$ are very large compared to $\|A\|$, the information in entries of A will get "lost" when these large values are subtracted from them. This is what happened to a_{22} in the example in section 2.4.1. If the intermediate quantities in the product $L \cdot U$ were instead comparable to those of A, we would expect a tiny backward error $A - LU$ in the factorization. Therefore, we want to bound the largest intermediate quantities in the product $L \cdot U$. We will do this by bounding the entries of the matrix $|L| \cdot |U|$ (see section 1.1 for notation).

Our analysis is analogous to the one we used for polynomial evaluation in section 1.6. There we considered $p = \sum_i a_i x^i$ and showed that if $|p|$ were comparable to the sum of absolute values $\sum_i |a_i x^i|$, then p would be computed accurately.

After presenting a general analysis of Gaussian elimination, we will use it to show that GEPP (or, more expensively, GECP) will keep the entries of $|L| \cdot |U|$ comparable to $\|A\|$ in almost all practical circumstances.

Unfortunately, the best bounds on $\|\delta A\|$ that we can prove in general are still much larger than the errors encountered in practice. Therefore, the error bounds that we use in practice will be based on the computed residual r and bound (2.5) (or bound (2.9)) instead of the rigorous but pessimistic bound in this section.

Now suppose that matrix A has already been pivoted, so the notation is simpler. We simplify Algorithm 2.2 to two equations, one for a_{jk} with $j \leq k$ and one for $j > k$. Let us first trace what Algorithm 2.2 does to a_{jk} when $j \leq k$: this element is repeatedly updated by subtracting $l_{ji}u_{ik}$ for $i = 1$ to $j-1$ and is finally assigned to u_{jk} so that

$$u_{jk} = a_{jk} - \sum_{i=1}^{j-1} l_{ji}u_{ik}.$$

When $j > k$, a_{jk} again has $l_{ji}u_{ik}$ subtracted for $i = 1$ to $k-1$, and then the resulting sum is divided by u_{kk} and assigned to l_{jk}:

$$l_{jk} = \frac{a_{jk} - \sum_{i=1}^{k-1} l_{ji}u_{ik}}{u_{kk}}.$$

To do the roundoff error analysis of these two formulas, we use the result from Question 1.10 that a dot product computed in floating point arithmetic satisfies

$$\mathrm{fl}\left(\sum_{i=1}^{d} x_i y_i\right) = \sum_{i=1}^{d} x_i y_i(1+\delta_i) \quad \text{with} \quad |\delta_i| \leq d\varepsilon.$$

We apply this to the formula for u_{jk}, yielding[9]

$$u_{jk} = \left(a_{jk} - \sum_{i=1}^{j-1} l_{ji}u_{ik}(1+\delta_i)\right)(1+\delta')$$

with $|\delta_i| \leq (j-1)\varepsilon$ and $|\delta'| \leq \varepsilon$. Solving for a_{jk} we get

$$\begin{aligned}
a_{jk} &= \tfrac{1}{1+\delta'}u_{jk} \cdot l_{jj} + \sum_{i=1}^{j-1} l_{ji}u_{ik}(1+\delta_i) \quad \text{since } l_{jj} = 1 \\
&= \sum_{i=1}^{j} l_{ji}u_{ik} + \sum_{i=1}^{j} l_{ji}u_{ik}\delta_i \\
&\quad \text{with } |\delta_i| \leq (j-1)\varepsilon \text{ and } 1+\delta_j \equiv \tfrac{1}{1+\delta'} \\
&\equiv \sum_{i=1}^{j} l_{ji}u_{ik} + E_{jk},
\end{aligned}$$

[9]Strictly speaking, the next formula assumes that we compute the sum first and then subtract from a_{jk}. But the final bound does not depend on the order of summation.

where we can bound E_{jk} by

$$|E_{jk}| = \left| \sum_{i=1}^{j} l_{ji} \cdot u_{ik} \cdot \delta_i \right| \leq \sum_{i=1}^{j} |l_{ji}| \cdot |u_{ik}| \cdot n\varepsilon = n\varepsilon(|L| \cdot |U|)_{jk}.$$

Doing the same analysis for the formula for l_{jk} yields

$$l_{jk} = (1+\delta'') \left(\frac{(1+\delta')(a_{jk} - \sum_{i=1}^{k-1} l_{ji}u_{ik}(1+\delta_i))}{u_{kk}} \right)$$

with $|\delta_i| \leq (k-1)\varepsilon$, $|\delta'| \leq \varepsilon$, and $|\delta''| \leq \varepsilon$. We solve for a_{jk} to get

$$\begin{aligned} a_{jk} &= \frac{1}{(1+\delta')(1+\delta'')} u_{kk} l_{jk} + \sum_{i=1}^{k-1} l_{ji}u_{ik}(1+\delta_i) \\ &= \sum_{i=1}^{k} l_{ji}u_{ik} + \sum_{i=1}^{k} l_{ji}u_{ik}\delta_i \quad \text{with } 1+\delta_k \equiv \frac{1}{(1+\delta')(1+\delta'')} \\ &\equiv \sum_{i=1}^{k} l_{ji}u_{ik} + E_{jk} \end{aligned}$$

with $|\delta_i| \leq n\varepsilon$, and so $|E_{jk}| \leq n\varepsilon(|L| \cdot |U|)_{jk}$ as before.

Altogether, we can summarize this error analysis with the simple formula $A = LU + E$ where $|E| \leq n\varepsilon|L| \cdot |U|$. Taking norms we get $\|E\| \leq n\varepsilon \| \, |L| \, \| \cdot \| \, |U| \, \|$. If the norm does not depend on the signs of the matrix entries (true for the Frobenius, infinity-, and one-norms but not the two-norm), we can simplify this to $\|E\| \leq n\varepsilon \|L\| \cdot \|U\|$.

Now we consider the rest of the problem: solving $LUx = b$ via $Ly = b$ and $Ux = y$. The result of Question 1.11 shows that solving $Ly = b$ by forward substitution yields a computed solution $\hat{y}$ satisfying $(L+\delta L)\hat{y} = b$ with $|\delta L| \leq n\varepsilon|L|$. Similarly when solving $Ux = \hat{y}$ we get $\hat{x}$ satisfying $(U+\delta U)\hat{x} = \hat{y}$ with $|\delta U| \leq n\varepsilon|U|$.

Combining these yields

$$\begin{aligned} b &= (L+\delta L)\hat{y} \\ &= (L+\delta L)(U+\delta U)\hat{x} \\ &= (LU + L\delta U + \delta LU + \delta L\delta U)\hat{x} \\ &= (A - E + L\delta U + \delta LU + \delta L\delta U)\hat{x} \\ &\equiv (A+\delta A)\hat{x}, \quad \text{where} \quad \delta A = -E + L\delta U + \delta LU + \delta L\delta U. \end{aligned}$$

Now we combine our bounds on E, δL, and δU and use the triangle inequality to bound δA:

$$|\delta A| = |-E + L\delta U + \delta LU + \delta L\delta U|$$

$$\begin{aligned} &\leq |E| + |L\delta U| + |\delta L U| + |\delta L \delta U| \\ &\leq |E| + |L| \cdot |\delta U| + |\delta L| \cdot |U| + |\delta L| \cdot |\delta U| \\ &\leq n\varepsilon |L| \cdot |U| + n\varepsilon |L| \cdot |U| + n\varepsilon |L| \cdot |U| + n^2\varepsilon^2 |L| \cdot |U| \\ &\approx 3n\varepsilon |L| \cdot |U|. \end{aligned}$$

Taking norms and assuming $\| \, |X| \, \| = \|X\|$ (true as before for the Frobenius, infinity-, and one-norms but not the two-norm) we get $\|\delta A\| \leq 3n\varepsilon \|L\| \cdot \|U\|$.

Thus, to see when Gaussian elimination is backward stable, we must ask when $3n\varepsilon \|L\| \cdot \|U\| = O(\varepsilon)\|A\|$; then the $\frac{\|\delta A\|}{\|A\|}$ in the perturbation theory bounds will be $O(\varepsilon)$ as we desire (note that $\delta b = 0$).

The main empirical observation, justified by decades of experience, is that GEPP *almost* always keeps $\|L\| \cdot \|U\| \approx \|A\|$. GEPP guarantees that each entry of L is bounded by 1 in absolute value, so we need consider only $\|U\|$. We define the *pivot growth factor for GEPP*[10] as $g_{\text{PP}} = \|U\|_{\max} / \|A\|_{\max}$, where $\|A\|_{\max} = \max_{ij} |a_{ij}|$, so stability is equivalent to g_{PP} being small or growing slowly as a function of n. In practice, g_{PP} is almost always n or less. The average behavior seems to be $n^{2/3}$ or perhaps even just $n^{1/2}$ [242]. (See Figure 2.1.) This makes GEPP the algorithm of choice for many problems. Unfortunately, there are rare examples in which g_{PP} can be as large as 2^{n-1}.

PROPOSITION 2.1. *GEPP guarantees that $g_{\text{PP}} \leq 2^{n-1}$. This bound is attainable.*

Proof. The first step of GEPP updates $\tilde{a}_{jk} = a_{jk} - l_{ji} \cdot u_{ik}$, where $|l_{ji}| \leq 1$ and $|u_{ik}| = |a_{ik}| \leq \max_{rs} |a_{rs}|$, so $|\tilde{a}_{jk}| \leq 2 \cdot \max_{rs} |a_{rs}|$. So each of the $n-1$ major steps of GEPP can double the size of the remaining matrix entries, and we get 2^{n-1} as the overall bound. See the example in Question 2.14 to see that this is attainable. □

Putting all these bounds together, we get

$$\|\delta A\|_\infty \leq 3 g_{\text{PP}} n^3 \varepsilon \|A\|_\infty, \tag{2.11}$$

since $\|L\|_\infty \leq n$ and $\|U\|_\infty \leq n g_{\text{PP}} \|A\|_\infty$. The factor $3 g_{\text{PP}} n^3$ in the bound causes it to almost always greatly overestimate the true $\|\delta A\|$, even if $g_{\text{PP}} = 1$. For example, if $\varepsilon = 10^{-7}$ and $n = 150$, a very modest-sized matrix, then $3n^3\varepsilon > 1$, meaning that all precision is potentially lost. Example 2.3 graphs $3 g_{\text{PP}} n^3 \varepsilon$ along with the true backward error to show how it can be pessimistic; $\|\delta A\|$ is usually $O(\varepsilon)\|A\|$, so we can say that GEPP is *backward stable in practice*, even though we can construct examples where it fails. Section 2.4.4 presents practical error bounds for the computed solution of $Ax = b$ that are much smaller than what we get from using $\|\delta A\|_\infty \leq 3 g_{\text{PP}} n^3 \varepsilon \|A\|_\infty$.

[10] This definition is slightly different from the usual one in the literature but essentially equivalent [121, p. 115].

It can be shown that GECP is even more stable than GEPP, with its pivot growth g_{CP} satisfying the worst-case bound [262, p. 213]

$$g_{\text{CP}} = \frac{\max_{ij} |u_{ij}|}{\max_{ij} |a_{ij}|} \leq \sqrt{n \cdot 2 \cdot 3^{1/2} \cdot 4^{1/3} \cdots n^{1/(n-1)}} \approx n^{1/2 + \log_e n/4}.$$

This upper bound is also much too large in practice. The average behavior of g_{CP} is $n^{1/2}$. It was an old open conjecture that $g_{\text{CP}} \leq n$, but this was recently disproved [99, 122]. It remains an open problem to find a good upper bound for g_{CP} (which is still widely suspected to be $O(n)$.)

The extra $O(n^3)$ comparisons that GECP uses to find the pivots ($O(n^2)$ comparisons per step, versus $O(n)$ for GEPP) makes GECP significantly slower than GEPP, especially on high-performance machines that perform floating point operations about as fast as comparisons. Therefore, using GECP is seldom warranted (but see sections 2.4.4, 2.5.1, and 5.4.3).

EXAMPLE 2.3. Figures 2.1 and 2.2 illustrate these backward error bounds. For both figures, five random matrices A of each dimension were generated, with independent normally distributed entries, of mean 0 and standard deviation 1. (Testing such random matrices can sometimes be misleading about the behavior on some real problems, but it is still informative.) For each matrix, a similarly random vector b was generated. Both GEPP and GECP were used to solve $Ax = b$. Figure 2.1 plots the pivot growth factors g_{PP} and g_{CP}. In both cases they grow slowly with dimension, as expected. Figure 2.2 shows our two upper bounds for the backward error, $3n^3 \varepsilon g_{\text{PP}}$ (or $3n^3 \varepsilon g_{\text{CP}}$) and $3n\varepsilon \frac{\||L| \cdot |U|\|_\infty}{\|A\|_\infty}$. It also shows the true backward error, computed as described in Theorem 2.2. Machine epsilon is indicated by a solid horizontal line at $\varepsilon = 2^{-53} \approx 1.1 \cdot 10^{-16}$. Both bounds are indeed bounds on the true backward error but are too large by several order of magnitude. For the Matlab program that produced these plots, see HOMEPAGE/Matlab/pivot.m. ◇

2.4.3. Estimating Condition Numbers

To compute a practical error bound based on a bound like (2.5), we need to estimate $\|A^{-1}\|$. This is also enough to estimate the condition number $\kappa(A) = \|A^{-1}\| \cdot \|A\|$, since $\|A\|$ is easy to compute. One approach is to compute A^{-1} explicitly and compute its norm. However, this would cost $2n^3$, more than the original $\frac{2}{3}n^3$ for Gaussian elimination. (Note that this implies that it is not cheaper to solve $Ax = b$ by computing A^{-1} and then multiplying it by b. This is true even if one has many different b vectors. See Question 2.2.) It is a fact that most users will not bother to compute error bounds if they are expensive.

So instead of computing A^{-1} we will devise a much cheaper algorithm to *estimate* $\|A^{-1}\|$. Such an algorithm is called a *condition estimator* and should have the following properties:

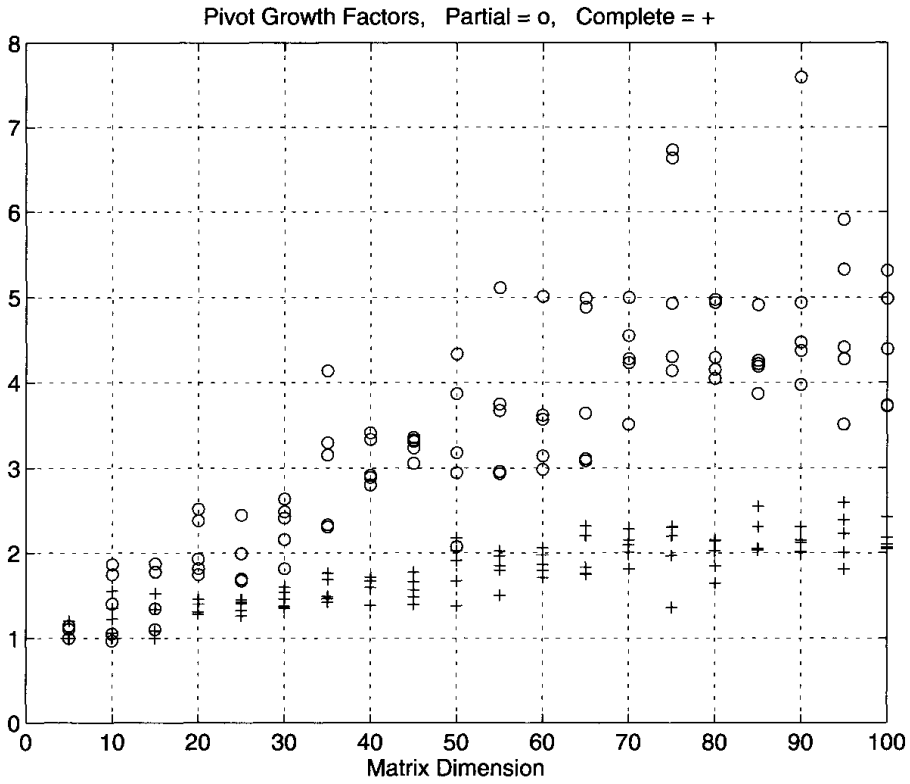


Fig. 2.1. *Pivot growth for random matrices,* $\circ = g_{\mathrm{PP}}$, $+ = g_{\mathrm{CP}}$.

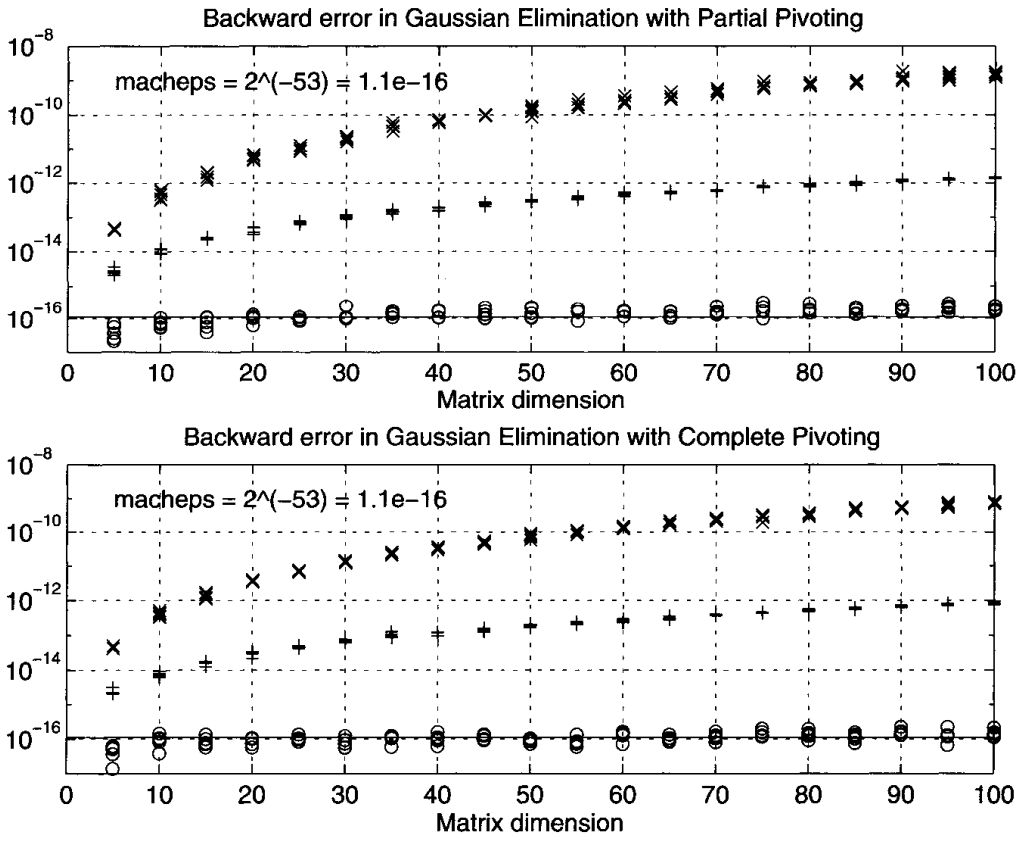


Fig. 2.2. *Backward error in Gaussian elimination on random matrices,* $\times = 3n^3\varepsilon g$, $+ = 3n\||L| \cdot |U|\|_\infty / \|A\|_\infty$, $\circ = \|Ax - b\|_\infty / (\|A\|_\infty \|x\|_\infty)$.

1. Given the L and U factors of A, it should cost $O(n^2)$, which for large enough n is negligible compared to the $\frac{2}{3}n^3$ cost of GEPP.

2. It should provide an estimate which is almost always within a factor of 10 of $\|A^{-1}\|$. This is all one needs for an error bound which tells you about how many decimal digits of accuracy that you have. (A factor-of-10 error is one decimal digit.[11])

There are a variety of such estimators available (see [146] for a survey). We choose to present one that is widely applicable to problems besides solving $Ax = b$, at the cost of being slightly slower than algorithms specialized for $Ax = b$ (but it is still reasonably fast). Our estimator, like most others, is guaranteed to produce only a *lower* bound on $\|A^{-1}\|$, not an upper bound. Empirically, it is almost always within a factor of 10, and usually 2 to 3, of $\|A^{-1}\|$. For the matrices in Figures 2.1 and 2.2, where the condition numbers varied from 10 to 10^5, the estimator equaled the condition number to several decimal places 83% of the time and was .43 times too small at worst. This is more than accurate enough to estimate the number of correct decimal digits in the final answer.

The algorithm estimates the one-norm $\|B\|_1$ of a matrix B, provided that we can compute Bx and $B^T y$ for arbitrary x and y. We will apply the algorithm to $B = A^{-1}$, so we need to compute $A^{-1}x$ and $A^{-T}y$, i.e., solve linear systems. This costs just $O(n^2)$ given the LU factorization of A. The algorithm was developed in [138, 146, 148], with the latest version in [147]. Recall that $\|B\|_1$ is defined by

$$\|B\|_1 \neq \max_{x \neq 0} \frac{\|Bx\|_1}{\|x\|_1} = \max_j \sum_{i=1}^{n} |b_{ij}|.$$

It is easy for us to show that the maximum over $x \neq 0$ is attained at $x = e_{j_0} = [0, \ldots, 0, 1, 0, \ldots, 0]^T$. (The single nonzero entry is component j_0, where $\max_j \sum_i |b_{ij}|$ occurs at $j = j_0$.)

Searching over all $e_j, j = 1, \ldots, n$, means computing all columns of $B = A^{-1}$; this is too expensive. Instead, since $\|B\|_1 = \max_{\|x\|_1 \leq 1} \|Bx\|_1$, we can use *hill climbing* or *gradient ascent* on $f(x) \equiv \|Bx\|_1$ inside the set $\|x\|_1 \leq 1$. $\|x\|_1 \leq 1$ is clearly a convex set of vectors, and $f(x)$ is a convex function, since $0 \leq \alpha \leq 1$ implies $f(\alpha x + (1-\alpha)y) = \|\alpha Bx + (1-\alpha)By\|_1 \leq \alpha\|Bx\|_1 + (1-\alpha)\|By\|_1 = \alpha f(x) + (1-\alpha)f(y)$.

Doing gradient ascent to maximize $f(x)$ means moving x in the direction of the gradient $\nabla f(x)$ (if it exists) as long as $f(x)$ increases. The convexity of $f(x)$ means $f(y) \geq f(x) + \nabla f(x) \cdot (y - x)$ (if $\nabla f(x)$ exists). To compute ∇f we *assume* all $\sum_j b_{ij}x_j \neq 0$ in $f(x) = \sum_i |\sum_j b_{ij}x_j|$ (this is almost always

[11] As stated earlier, no one has ever found an estimator that approximates $\|A^{-1}\|$ with some guaranteed accuracy and is simultaneously significantly cheaper than explicitly computing A^{-1}. It has been been conjectured that no such estimator exists, but this has not been proven.

true). Let $\zeta_i = \text{sign}(\sum_j b_{ij}x_j)$, so $\zeta_i = \pm 1$ and $f(x) = \sum_i \sum_j \zeta_i b_{ij} x_j$. Then $\frac{\partial f}{\partial x_k} = \sum_i \zeta_i b_{ik}$ and $\nabla f = \zeta^T B = (B^T \zeta)^T$.

In summary, to compute $\nabla f(x)$ takes three steps: $w = Bx$, $\zeta = \text{sign}(w)$, and $\nabla f = \zeta^T B$.

ALGORITHM 2.5. *Hager's condition estimator returns a lower bound* $\|w\|_1$ *on* $\|B\|_1$*:*

```
choose any x such that ||x||_1 = 1          /* e.g. x_i = 1/n */
repeat
    w = Bx, ζ = sign(w), z = B^T ζ          /* z^T = ∇f */
    if ||z||_∞ ≤ z^T x then
        return ||w||_1
    else
        x = e_j where |z_j| = ||z||_∞
    endif
end repeat
```

THEOREM 2.6. 1. *When* $\|w\|_1$ *is returned,* $\|w\|_1 = \|Bx\|_1$ *is a local maximum of* $\|Bx\|_1$.

2. *Otherwise,* $\|Be_j\|$ *(at end of loop)* $> \|Bx\|$ *(at start), so the algorithm has made progress in maximizing* $f(x)$.

Proof.

1. In this case, $\|z\|_\infty \le z^T x$. Near x, $f(x) = \|Bx\|_1 = \sum_i \sum_j \zeta_i b_{ij} x_j$ is linear in x so $f(y) = f(x) + \nabla f(x) \cdot (y - x) = f(x) + z^T(y - x)$, where $z^T = \nabla f(x)$. To show x is a local maximum we want $z^T(y-x) \le 0$ when $\|y\|_1 = 1$. We compute

$$\begin{aligned} z^T(y-x) &= z^T y - z^T x = \sum_i z_i \cdot y_i - z^T x \le \sum_i |z_i| \cdot |y_i| - z^T x \\ &\le \|z\|_\infty \cdot \|y\|_1 - z^T x = \|z\|_\infty - z^T x \le 0 \quad \text{as desired.} \end{aligned}$$

2. In this case $\|z\|_\infty > z^T x$. Choose $\tilde{x} = e_j \cdot \text{sign}(z_j)$, where j is chosen so that $|z_j| = \|z\|_\infty$. Then

$$\begin{aligned} f(\tilde{x}) &\ge f(x) + \nabla f \cdot (\tilde{x} - x) = f(x) + z^T(\tilde{x} - x) \\ &= f(x) + z^T\tilde{x} - z^T x = f(x) + |z_j| - z^T x > f(x), \end{aligned}$$

where the last inequality is true by construction. □

Higham [147, 148] tested a slightly improved version of this algorithm by trying many random matrices of sizes $10, 25, 50$ and condition numbers $\kappa = 10, 10^3, 10^6, 10^9$; in the worst case the computed κ underestimated the

true κ by a factor .44. The algorithm is available in LAPACK as subroutine `slacon`. LAPACK routines like `sgesvx` call `slacon` internally and return the estimated condition number. (They actually return the reciprocal of the estimated condition number, to avoid overflow on exactly singular matrices.) A different condition estimator is available in Matlab as `rcond`. The Matlab routine `cond` computes the exact condition number $\|A^{-1}\|_2\|A\|_2$, using algorithms discussed in section 5.4; it is much more expensive than `rcond`.

Estimating the Relative Condition Number

We can also use the algorithm from the last section to estimate the relative condition number $\kappa_{CR}(A) = \| \, |A^{-1}| \cdot |A| \, \|_\infty$ from bound (2.8) or to evaluate the bound $\| \, |A^{-1}| \cdot |r| \, \|_\infty$ from (2.9). We can reduce both to the same problem, that of estimating $\| \, |A^{-1}| \cdot g \, \|_\infty$, where g is a vector of nonnegative entries. To see why, let e be the vector of all ones. From part 5 of Lemma 1.7, we see that $\|X\|_\infty = \|Xe\|_\infty$ if the matrix X has nonnegative entries. Then

$$\| \, |A^{-1}| \cdot |A| \, \|_\infty = \| \, |A^{-1}| \cdot |A|e \, \|_\infty = \| \, |A^{-1}| \cdot g \, \|_\infty, \quad \text{where} \quad g = |A|e.$$

Here is how we estimate $\| \, |A^{-1}| \cdot g \, \|_\infty$. Let $G = \mathrm{diag}(g_1, \ldots, g_n)$; then $g = Ge$. Thus

$$\begin{aligned} \| \, |A^{-1}| \cdot g \, \|_\infty &= \| \, |A^{-1}| \cdot Ge \, \|_\infty = \| \, |A^{-1}| \cdot G \, \|_\infty = \| \, |A^{-1}G| \, \|_\infty \\ &= \|A^{-1}G\|_\infty. \end{aligned} \tag{2.12}$$

The last equality is true because $\|Y\|_\infty = \| \, |Y| \, \|_\infty$ for any matrix Y. Thus, it suffices to estimate the infinity norm of the matrix $A^{-1}G$. We can do this by applying Hager's algorithm, Algorithm 2.5, to the matrix $(A^{-1}G)^T = GA^{-T}$, to estimate $\|(A^{-1}G)^T\|_1 = \|A^{-1}G\|_\infty$ (see part 6 of Lemma 1.7). This requires us to multiply by the matrix GA^{-T} and its transpose $A^{-1}G$. Multiplying by G is easy since it is diagonal, and we multiply by A^{-1} and A^{-T} using the LU factorization of A, as we did in the last section.

2.4.4. Practical Error Bounds

We present two practical error bounds for our approximate solution $\hat{x}$ of $Ax = b$. For the first bound we use inequality (2.5) to get

$$\text{error} = \frac{\|\hat{x} - x\|_\infty}{\|\hat{x}\|_\infty} \leq \|A^{-1}\|_\infty \cdot \frac{\|r\|_\infty}{\|\hat{x}\|_\infty}, \tag{2.13}$$

where $r = A\hat{x} - b$ is the residual. We estimate $\|A^{-1}\|_\infty$ by applying Algorithm 2.5 to $B = A^{-T}$, estimating $\|B\|_1 = \|A^{-T}\|_1 = \|A^{-1}\|_\infty$ (see parts 5 and 6 of Lemma 1.7).

Our second error bound comes from the tighter inequality (2.9):

$$\text{error} = \frac{\|\hat{x} - x\|_\infty}{\|\hat{x}\|_\infty} \leq \frac{\| \, |A^{-1}| \cdot |r| \, \|_\infty}{\|\hat{x}\|_\infty}. \tag{2.14}$$

We estimate $\| |A^{-1}| \cdot |r| \|_\infty$ using the algorithm based on equation (2.12). Error bound (2.14) (modified as described below in the subsection "What can go wrong") is computed by LAPACK routines like `sgesvx`. The LAPACK variable name for the error bound is `FERR`, for Forward ERRor.

EXAMPLE 2.4. We have computed the first error bound (2.13) and the true error for the same set of examples as in Figures 2.1 and 2.2, plotting the result in Figure 2.3. For each problem $Ax = b$ solved with GEPP we plot a $\circ$ at the point (true error, error bound), and for each problem $Ax = b$ solved with GECP we plot a $+$ at the point (true error, error bound). If the error bound were equal to the true error, the $\circ$ or $+$ would lie on the solid diagonal line. Since the error bound always exceeds the true error, the $\circ$s and $+$s lie above this diagonal. When the error bound is less than 10 times larger than the true error, the $\circ$ or $+$ appears between the solid diagonal line and the first superdiagonal dashed line. When the error bound is between 10 and 100 times larger than the true error, the $\circ$ or $+$ appears between the first two superdiagonal dashed lines. Most error bounds are in this range, with a few error bounds as large as 1000 times the true error. Thus, our computed error bound underestimates the number of correct decimal digits in the answer by one or two and in rare cases by as much as three. The Matlab code for producing these graphs is the same as before, HOMEPAGE/Matlab/pivot.m. $\diamond$

EXAMPLE 2.5. We present an example chosen to illustrate the difference between the two error bounds (2.13) and (2.14). This example will also show that GECP can sometimes be more accurate than GEPP. We choose a set of badly scaled examples constructed as follows. Each test matrix is of the form $A = DB$, with the dimension running from 5 to 100. B is equal to an identity matrix plus very small random offdiagonal entries, around 10^{-7}, so it is very well-conditioned. D is a diagonal matrix with entries scaled geometrically from 1 up to 10^{14}. (In other words, $d_{i+1,i+1}/d_{i,i}$ is the same for all i.) The A matrices have condition numbers $\kappa(A) = \|A^{-1}\|_\infty \cdot \|A\|_\infty$ nearly equal to 10^{14}, which is very ill-conditioned, although their relative condition numbers $\kappa_{CR}(A) = \| |A^{-1}| \cdot |A| \|_\infty = \| |B^{-1}| \cdot |B| \|_\infty$ are all nearly 1. As before, machine precision is $\varepsilon = 2^{-53} \approx 10^{-16}$. The examples were computed using the same Matlab code HOMEPAGE/Matlab/pivot.m.

The pivot growth factors g_{PP} and g_{CP} were never larger than about 1.33 for any example, and the backward error from Theorem 2.2 never exceeded 10^{-15} in any case. Hager's estimator was very accurate in all cases, returning the true condition number 10^{14} to many decimal places.

Figure 2.4 plots the error bounds (2.13) and (2.14) for these examples, along with the componentwise relative backward error, as given by the formula in Theorem 2.3. The cluster of plus signs in the upper left corner of Figure 2.4(a) shows that while GECP computes the answer with a tiny error near 10^{-15}, the error bound (2.13) is usually closer to 10^{-2}, which is very pessimistic. This

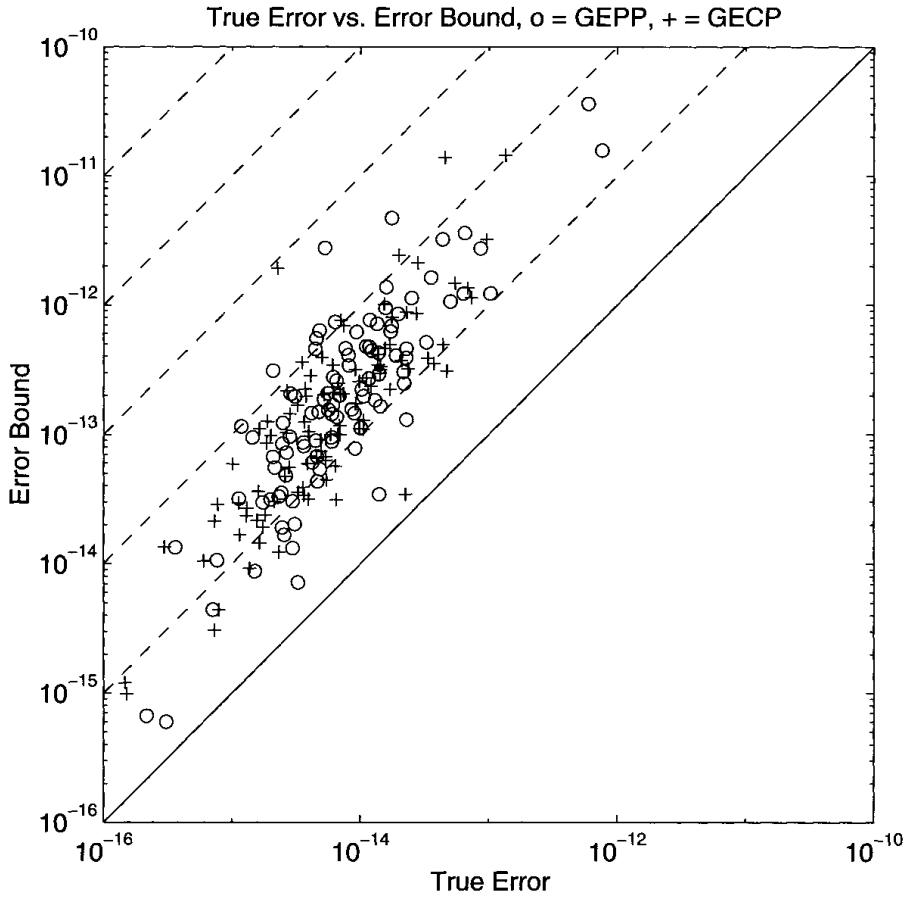


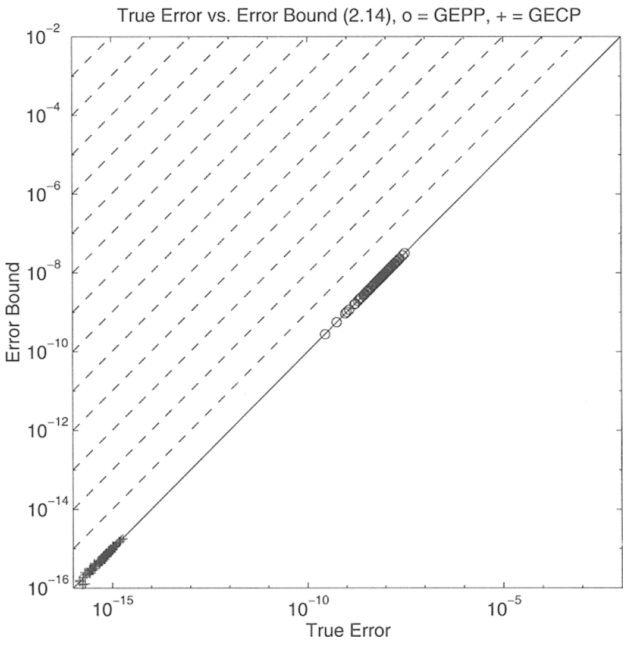
Fig. 2.3. *Error bound* (2.13) *plotted versus true error,* $\circ$ = *GEPP,* $+$ = *GECP.*

is because the condition number is 10^{14}, and so unless the backward error is much smaller than $\varepsilon \approx 10^{-16}$, which is unlikely, the error bound will be close to $10^{-16}10^{14} = 10^{-2}$. The cluster of circles in the middle top of the same figure shows that GEPP gets a larger error of about 10^{-8}, while the error bound (2.13) is again usually near 10^{-2}.

In contrast, the error bound (2.14) is nearly perfectly accurate, as illustrated by the pluses and circles on the diagonal in Figure 2.4(b). This graph again illustrates that GECP is nearly perfectly accurate, whereas GEPP loses about half the accuracy. This difference in accuracy is explained by Figure 2.4(c), which shows the componentwise relative backward error from Theorem 2.3 for GEPP and GECP. This graph makes it clear that GECP has nearly perfect backward error in the componentwise relative sense, so since the corresponding componentwise relative condition number is 1, the accuracy is perfect. GEPP on the other hand is not completely stable in this sense, losing from 5 to 10 decimal digits.

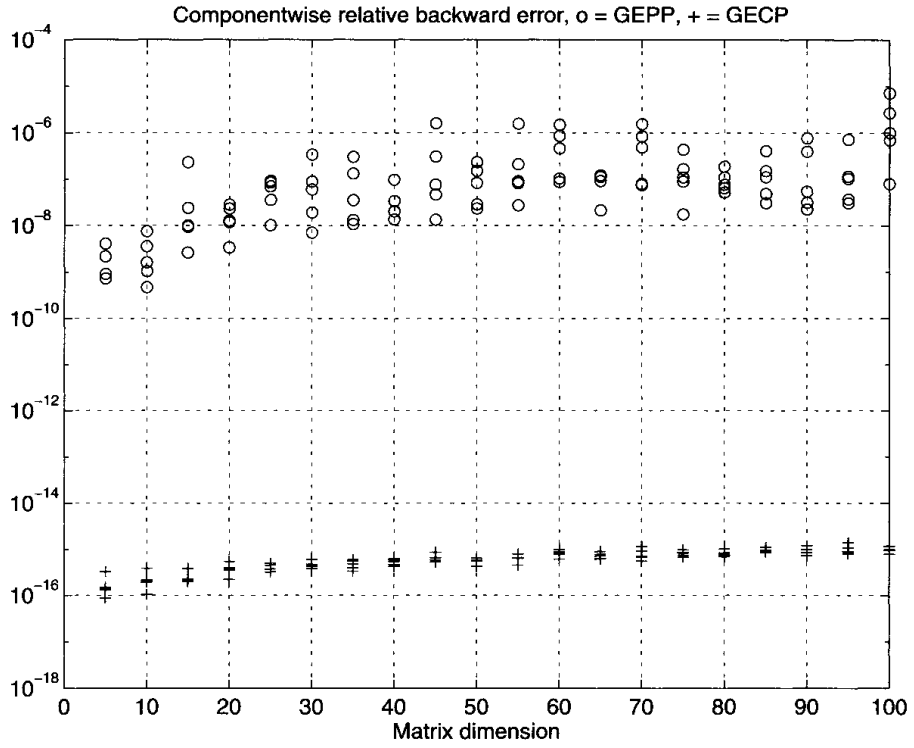
In section 2.5 we show how to iteratively improve the computed solution $\hat{x}$. One step of this method will make the solution computed by GEPP as accurate as the solution from GECP. Since GECP is significantly more expensive than GEPP in practice, it is very rarely used. $\diamond$

(a)



(b)

Fig. 2.4. (a) *plots the error bound* (2.13) *versus the true error.* (b) *plots the error bound* (2.14) *versus the true error.*



(c)

Fig. 2.4. *Continued.* (c) *plots the componentwise relative backward error from Theorem* 2.3.

What Can Go Wrong

Unfortunately, as mentioned in the beginning of section 2.4, error bounds (2.13) and (2.14) are *not* guaranteed to provide tight bounds in all cases when implemented in practice. In this section we describe the (rare!) ways they can fail, and the partial remedies used in practice.

First, as described in section 2.4.3, the estimate of $\|A^{-1}\|$ from Algorithm 2.5 (or similar algorithms) provides only a lower bound, although the probability is very low that it is more than 10 times too small.

Second, there is a small but nonnegligible probability that roundoff in the evaluation of $r = A\hat{x} - b$ might make $\|r\|$ artificially small, in fact zero, and so also make our computed error bound too small. To take this possibility into account, one can add a small quantity to $|r|$ to account for it: From Question 1.10 we know that the roundoff in evaluating r is bounded by

$$|(A\hat{x} - b) - \mathrm{fl}(A\hat{x} - b)| \leq (n+1)\varepsilon(|A| \cdot |\hat{x}| + |b|), \tag{2.15}$$

so we can replace $|r|$ with $|r| + (n+1)\varepsilon(|A| \cdot |\hat{x}| + |b|)$ in bound (2.14) (this is done in the LAPACK code `sgesvx`) or $\|r\|$ with $\|r\| + (n+1)\varepsilon(\|A\| \cdot \|\hat{x}\| + \|b\|)$ in bound (2.13). The factor $n+1$ is usually much too large and can be omitted if desired.

Third, roundoff in performing Gaussian elimination on very ill-conditioned matrices can yield such inaccurate L and U that bound (2.14) is much too low.

EXAMPLE 2.6. We present an example, discovered by W. Kahan, that illustrates the difficulties in getting truly guaranteed error bounds. In this example the matrix A will be *exactly* singular. Therefore any error bound on $\frac{\|x-\hat{x}\|}{\|x\|}$ should be one or larger to indicate that no digits in the computed solution are correct, since the true solution does not exist.

Roundoff error during Gaussian elimination will yield nonsingular but very ill-conditioned factors L and U. With this example, computing using Matlab with IEEE double precision arithmetic, the computed residual r turns out to be *exactly* zero because of roundoff, so both error bounds (2.13) and (2.14) return zero. If we repair bound (2.13) by adding $4\varepsilon(\|A\| \cdot \|\hat{x}\| + \|b\|)$, it will be larger than 1 as desired.

Unfortunately our second, "tighter" error bound (2.14) is about 10^{-7}, erroneously indicating that seven digits of the computed solution are correct.

Here is how the example is constructed. Let $\chi = 3/2^{29}$, $\zeta = 2^{14}$,

$$A = \begin{bmatrix} \chi\cdot\zeta & -\zeta & \zeta \\ \zeta^{-1} & \zeta^{-1} & 0 \\ \zeta^{-1} & -\chi\cdot\zeta^{-1} & \zeta^{-1} \end{bmatrix}$$
$$\approx \begin{bmatrix} 9.1553\cdot 10^{-5} & -1.6384\cdot 10^{4} & 1.6384\cdot 10^{4} \\ 6.1035\cdot 10^{-5} & 6.1035\cdot 10^{-5} & 0 \\ 6.1035\cdot 10^{-5} & -3.4106\cdot 10^{-13} & 6.1035\cdot 10^{-5} \end{bmatrix},$$

and $b = A \cdot [1, 1+\varepsilon, 1]^T$. A can be computed without any roundoff error, but b has a bit of roundoff, which means that it is not exactly in the space spanned by the columns of A, so $Ax = b$ has no solution. Performing Gaussian elimination, we get

$$L \approx \begin{bmatrix} 1 & 0 & 0 \\ .66666 & 1 & 0 \\ .66666 & 1.0000 & 1 \end{bmatrix}$$

and

$$U \approx \begin{bmatrix} 9.1553\cdot 10^{-5} & -1.6384\cdot 10^{4} & 1.6384\cdot 10^{4} \\ 0 & 1.0923\cdot 10^{4} & -1.0923\cdot 10^{4} \\ 0 & 0 & 1.8190\cdot 10^{-12} \end{bmatrix},$$

yielding a computed value of

$$A^{-1} \approx \begin{bmatrix} 2.0480\cdot 10^{3} & 5.4976\cdot 10^{11} & -5.4976\cdot 10^{11} \\ -2.0480\cdot 10^{3} & -5.4976\cdot 10^{11} & 5.4976\cdot 10^{11} \\ -2.0480\cdot 10^{3} & -5.4976\cdot 10^{11} & 5.4976\cdot 10^{11} \end{bmatrix}.$$

This means the computed value of $|A^{-1}| \cdot |A|$ has all entries approximately equal to $6.7109 \cdot 10^7$, so $\kappa_{CR}(A)$ is computed to be $O(10^7)$. In other words, the

error bound indicates that about $16 - 7 = 9$ digits of the computed solution are accurate, whereas none are.

Barring large pivot growth, one can prove that bound (2.13) (with $\|r\|$ appropriately increased) cannot be made artificially small by the phenomenon illustrated here.

Similarly, Kahan has found a family of n-by-n singular matrices, where changing one tiny entry (about 2^{-n}) to zero lowers $\kappa_{CR}(A)$ to $O(n^3)$. One could similarly construct examples where A was not exactly singular, so that bounds (2.13) and (2.14) were correct in exact arithmetic, but where roundoff made them much too small. $\diamond$

2.5. Improving the Accuracy of a Solution

We have just seen that the error in solving $Ax = b$ may be as large as $\kappa(A)\varepsilon$. If this error is too large, what can we do? One possibility is to rerun the entire computation in higher precision, but this may be quite expensive in time and space. Fortunately, as long as $\kappa(A)$ is not too large, there are much cheaper methods available for getting a more accurate solution.

To solve any equation $f(x) = 0$, we can try to use Newton's method to improve an approximate solution x_i to get $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. Applying this to $f(x) = Ax - b$ yields one step of *iterative refinement*:

$$\begin{aligned} r &= Ax_i - b \\ &\text{solve } Ad = r \text{ for } d \\ x_{i+1} &= x_i - d \end{aligned}$$

If we could compute $r = Ax_i - b$ exactly and solve $Ad = r$ exactly, we would be done in one step, which is what we expect from Newton applied to a linear problem. Roundoff error prevents this immediate convergence. The algorithm is interesting and of use precisely when A is so ill-conditioned that solving $Ad = r$ (and $Ax_0 = b$) is rather inaccurate.

THEOREM 2.7. *Suppose that r is computed in* double *precision and $\kappa(A) \cdot \varepsilon < c \equiv \frac{1}{3n^3g+1} < 1$, where n is the dimension of A and g is the pivot growth factor. Then repeated iterative refinement converges with*

$$\frac{\|x_i - A^{-1}b\|_\infty}{\|A^{-1}b\|_\infty} = O(\varepsilon).$$

Note that the condition number does not appear in the final error bound. This means that we compute the answer accurately independent *of the condition number, provided that $\kappa(A)\varepsilon$ is sufficiently less than* 1. *(In practice, c is too conservative an upper bound, and the algorithm often succeeds even when $\kappa(A)\varepsilon$ is greater than c.)*

Sketch of Proof. In order to keep the proof transparent, we will take only the most important rounding errors into account. For brevity, we abbreviate $\|\cdot\|_\infty$ by $\|\cdot\|$. Our goal is to show that

$$\|x_{i+1} - x\| \leq \frac{\kappa(A)\varepsilon}{c}\|x_i - x\| \equiv \zeta\|x_i - x\|.$$

By assumption, $\zeta < 1$, so this inequality implies that the error $\|x_{i+1} - x\|$ decreases monotonically to zero. (In practice it will not decrease all the way to zero because of rounding error in the assignment $x_{i+1} = x_i - d$, which we are ignoring.)

We begin by estimating the error in the computed residual r. We get $r = \mathrm{fl}(Ax_i - b) = Ax_i - b + f$, where by the result of Question 1.10 $|f| \leq n\varepsilon^2(|A| \cdot |x_i| + |b|) + \varepsilon|Ax_i - b| \approx \varepsilon|Ax_i - b|$. The ε^2 term comes from the double precision computation of r, and the ε term comes from rounding the double precision result back to single precision. Since $\varepsilon^2 \ll \varepsilon$, we will neglect the ε^2 term in the bound on $|f|$.

Next we get $(A+\delta A)d = r$, where from bound (2.11) we know that $\|\delta A\| \leq \gamma \cdot \varepsilon \cdot \|A\|$, where $\gamma = 3n^3g$, although this is usually much too large. As mentioned earlier, we simplify matters by assuming $x_{i+1} = x_i - d$ exactly.

Continuing to ignore all ε^2 terms, we get

$$\begin{aligned} d &= (A+\delta A)^{-1}r = (I + A^{-1}\delta A)^{-1}A^{-1}r \\ &= (I + A^{-1}\delta A)^{-1}A^{-1}(Ax_i - b + f) \\ &= (I + A^{-1}\delta A)^{-1}(x_i - x + A^{-1}f) \\ &\approx (I - A^{-1}\delta A)(x_i - x + A^{-1}f) \\ &\approx x_i - x - A^{-1}\delta A(x_i - x) + A^{-1}f. \end{aligned}$$

Therefore $x_{i+1} - x = x_i - d - x = A^{-1}\delta A(x_i - x) - A^{-1}f$ and so

$$\begin{aligned} \|x_{i+1} - x\| &\leq \|A^{-1}\delta A(x_i - x)\| + \|A^{-1}f\| \\ &\leq \|A^{-1}\| \cdot \|\delta A\| \cdot \|x_i - x\| + \|A^{-1}\| \cdot \varepsilon \cdot \|Ax_i - b\| \\ &\leq \|A^{-1}\| \cdot \|\delta A\| \cdot \|x_i - x\| + \|A^{-1}\| \cdot \varepsilon \cdot \|A(x_i - x)\| \\ &\leq \|A^{-1}\| \cdot \gamma\varepsilon \cdot \|A\| \cdot \|x_i - x\| \\ &\quad + \|A^{-1}\| \cdot \|A\| \cdot \varepsilon \cdot \|x_i - x\| \\ &= \|A^{-1}\| \cdot \|A\| \cdot \varepsilon \cdot (\gamma + 1) \cdot \|x_i - x\|, \end{aligned}$$

so if

$$\zeta = \|A^{-1}\| \cdot \|A\| \cdot \varepsilon(\gamma + 1) = \kappa(A)\varepsilon/c < 1,$$

then we have convergence. □

Iterative refinement (or other variations of Newton's method) can be used to improve accuracy for many other problems of linear algebra as well.

2.5.1. Single Precision Iterative Refinement

This section may be skipped on a first reading.

Sometimes double precision is not available to run iterative refinement. For example, if the input data is already in double precision, we would need to compute the residual r in *quadruple* precision, which may not be available. On some machines, such as the Intel Pentium, double-extended precision is available, which provides 11 more bits of fraction than double precision (see section 1.5). This is not as accurate as quadruple precision (which would need at least $2 \cdot 53 = 106$ fraction bits) but still improves the accuracy noticeably.

But if none of these options are available, one could still run iterative refinement while computing the residual r in single precision (i.e., the same precision as the input data). In this case, Theorem 2.7 does not hold any more. On the other hand, the following theorem shows that under certain technical assumptions, one step of iterative refinement in single precision is still worth doing because it reduces the componentwise relative backward error as defined in Theorem 2.3 to $O(\varepsilon)$. If the corresponding relative condition number $\kappa_{CR}(A) = \| \, |A^{-1}| \cdot |A| \, \|_\infty$ from section 2.2.1 is significantly smaller than the usual condition number $\kappa(A) = \|A^{-1}\|_\infty \cdot \|A\|_\infty$, then the answer will also be more accurate.

THEOREM 2.8. *Suppose that r is computed in single precision and*

$$\|A^{-1}\|_\infty \cdot \|A\|_\infty \cdot \frac{\max_i(|A| \cdot |x|)_i}{\min_i(|A| \cdot |x|)_i} \cdot \varepsilon < 1.$$

Then one step of iterative refinement yields x_1 such that $(A + \delta A)x_1 = b + \delta b$ with $|\delta a_{ij}| = O(\varepsilon)|a_{ij}|$ and $|\delta b_i| = O(\varepsilon)|b_i|$. In other words, the componentwise relative backward error is as small as possible. For example, this means that if A and b are sparse, then δA and δb have the same sparsity structures as A and b, respectively.

For a proof, see [149] as well as [14, 225, 226, 227] for more details.

Single precision iterative refinement and the error bound (2.14) are implemented in LAPACK routines like `sgesvx`.

EXAMPLE 2.7. We consider the same matrices as in Example 2.5 and perform one step of iterative refinement in the same precision as the rest of the computation ($\varepsilon \approx 10^{-16}$). For these examples, the usual condition number is $\kappa(A) \approx 10^{14}$, whereas $\kappa_{CR}(A) \approx 1$, so we expect a large accuracy improvement. Indeed, the componentwise relative error for GEPP is driven below 10^{-15}, and the corresponding error from (2.14) is driven below 10^{-15} as well. The Matlab code for this example is HOMEPAGE/Matlab/pivot.m. ◇

2.5.2. Equilibration

There is one more common technique for improving the error in solving a linear system: *equilibration*. This refers to choosing an appropriate diagonal matrix

D and solving $DAx = Db$ instead of $Ax = b$. D is chosen to try to make the condition number of DA smaller than that of A. In Example 2.7 for instance, choosing d_{ii} to be the reciprocal of the two-norm of row i of A would make DA nearly equal to the identity matrix, reducing its condition number from 10^{14} to 1. It is possible to show that choosing D this way reduces the condition number of DA to within a factor of $\sqrt{n}$ of its smallest possible value for any diagonal D [244]. In practice we may also choose two diagonal matrices D_{row} and D_{col} and solve $(D_{row}AD_{col})\bar{x} = D_{row}b$, $x = D_{col}\bar{x}$.

The techniques of iterative refinement and equilibration are implemented in the LAPACK subroutines like `sgerfs` and `sgeequ`, respectively. These are in turn used by driver routines like `sgesvx`.

2.6. Blocking Algorithms for Higher Performance

At the end of section 2.3, we said that changing the order of the three nested loops in the implementation of Gaussian elimination in Algorithm 2.2 could change the execution speed by orders of magnitude, depending on the computer and the problem being solved. In this section we will explore why this is the case and describe some carefully written linear algebra software which takes these matters into account. These implementations use so-called *block algorithms*, because they operate on square or rectangular subblocks of matrices in their innermost loops rather than on entire rows or columns. These codes are available in public-domain software libraries such as LAPACK (in Fortran, at NETLIB/lapack)[12] and ScaLAPACK (at NETLIB/scalapack). LAPACK (and its versions in other languages) are suitable for PCs, workstations, vector computers, and shared-memory parallel computers. These include the Sun SPARCcenter 2000 [238], SGI Power Challenge [223], DEC AlphaServer 8400 [103], and Cray C90/J90 [253, 254]. ScaLAPACK is suitable for distributed-memory parallel computers, such as the IBM SP-2 [256], Intel Paragon [257], Cray T3 series [255], and networks of workstations [9]. These libraries are available on NETLIB, including comprehensive manuals [10, 34].

A more comprehensive discussion of algorithms for high performance (especially parallel) machines may be found on the World Wide Web at PARALLEL_HOMEPAGE.

LAPACK was originally motivated by the poor performance of its predecessors LINPACK and EISPACK (also available on NETLIB) on some high-performance machines. For example, consider the table below, which presents the speed in Mflops of LINPACK's Cholesky routine `spofa` on a Cray YMP, a supercomputer of the late 1980s. Cholesky is a variant of Gaussian elimination suitable for symmetric positive definite matrices. It is discussed in depth in

[12] A C translation of LAPACK, called CLAPACK (at NETLIB/clapack), is also available. LAPACK++ (at NETLIB/c++/lapack++)) and LAPACK90 (at NETLIB/lapack90)) are C++ and Fortran 90 interfaces to LAPACK, respectively.

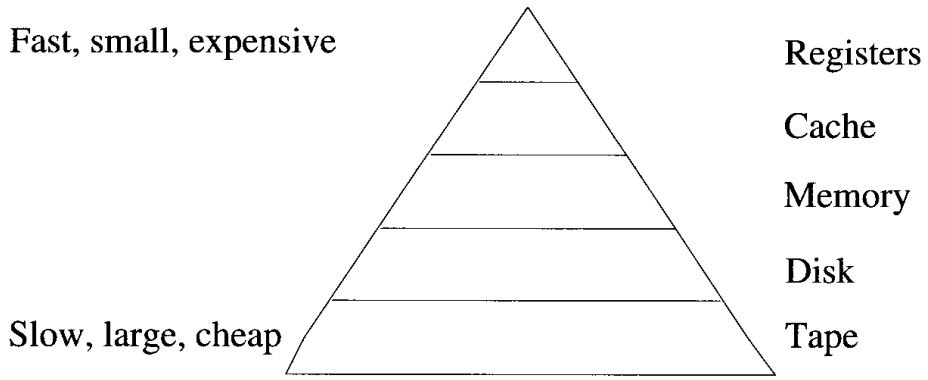
section 2.7; here it suffices to know that it is very similar to Algorithm 2.2. The table also includes the speed of several other linear algebra operations. The Cray YMP is a parallel computer with up to 8 processors that can be used simultaneously, so we include one column of data for 1 processor and another column where all 8 processors are used.

	1 Proc.	8 Procs.
Maximum speed	330	2640
Matrix-matrix multiply ($n = 500$)	312	2425
Matrix-vector multiply ($n = 500$)	311	2285
Solve $TX = B$ ($n = 500$)	309	2398
Solve $Tx = b$ ($n = 500$)	272	584
LINPACK (Cholesky, $n = 500$)	72	72
LAPACK (Cholesky, $n = 500$)	290	1414
LAPACK (Cholesky, $n = 1000$)	301	2115

The top line, the maximum speed of the machine, is an upper bound on the numbers that follow. The basic linear algebra operations on the next four lines have been measured using subroutines especially designed for high speed on the Cray YMP. They all get reasonably close to the maximum possible speed, except for solving $Tx = b$, a single triangular system of linear equations, which does not use 8 processors effectively. Solving $TX = B$ refers to solving triangular systems with many right-hand sides (B is a square matrix). These numbers are for large matrices and vectors ($n = 500$).

The Cholesky routine from LINPACK in the sixth line of the table executes significantly more slowly than these other operations, even though it is working on as large a matrix as the previous operations and doing mathematically similar operations. This poor performance leads us to try to reorganize Cholesky and other linear algebra routines to go as fast as their simpler counterparts like matrix-matrix multiplication. The speeds of these reorganized codes from LAPACK are given in the last two lines of the table. It is apparent that the LAPACK routines come much closer to the maximum speed of the machine. We emphasize that the LAPACK and LINPACK Cholesky routines perform the same floating operations, but in a different order.

To understand how these speedups were attained, we must understand how the time is spent by the computer while executing. This in turn requires us to understand how computer memories operate. It turns out that all computer memories, from the cheapest personal computer to the biggest supercomputer, are built as *hierarchies*, with a series of different kinds of memories ranging from very fast, expensive, and therefore small memory at the top of the hierarchy down to slow, cheap, and very large memory at the bottom.



For example, registers form the fastest memory, then cache, main memory, disks, and finally tape as the slowest, largest, and cheapest. Useful arithmetic and logical operations can be done *only* on data at the top of the hierarchy, in the registers. Data at one level of the memory hierarchy can move to adjacent levels—for example, moving between main memory and disk. The speed at which data move is high near the top of the hierarchy (between registers and cache) and slow near the bottom (between and disk and main memory). In particular, the speed at which arithmetic is done is much faster than the speed at which data is transferred between lower levels in the memory hierarchy, by factors of 10s or even 10000s, depending on the level. This means that an ill-designed algorithm may spend most of its time moving data from the bottom of the memory hierarchy to the registers in order to perform useful work rather than actually doing the work.

Here is an example of a simple algorithm which unfortunately cannot avoid spending most of its time moving data rather than doing useful arithmetic. Suppose that we want to add two large n-by-n matrices, large enough so that they fit only in a large, slow level of the memory hierarchy. To add them, they must be transferred a piece at a time up to the registers to do the additions, and the sums must be transferred back down. Thus, there are exactly 3 memory transfers between fast and slow memory (reading 2 summands into fast memory and writing 1 sum back to slow memory) for every addition performed. If the time to do a floating point operation is t_{arith} seconds and the time to move a word of data between memory levels is t_{mem} seconds, where $t_{\text{mem}} \gg t_{\text{arith}}$, then the execution time of this algorithm is $n^2(t_{\text{arith}} + 3t_{\text{mem}})$, which is much larger than than the time $n^2 t_{\text{arith}}$ required for the arithmetic alone. This means that matrix addition is doomed to run at the speed of the slowest level of memory in which the matrices reside, rather than the much higher speed of addition. In contrast, we will see later that other operations, such as matrix-matrix multiplication, can be made to run at the speed of the fastest level of the memory, even if the data are originally stored in the slowest.

LINPACK's Cholesky routine runs so slowly because it was *not* designed to minimize memory movement on machines such as the Cray YMP.[13] In contrast, matrix-matrix multiplication and the three other basic linear algebra

[13]It was designed to reduce another kind of memory movement, *page faults* between main memory and disk.

algorithms measured in the table were specialized to minimize data movement on a Cray YMP.

2.6.1. Basic Linear Algebra Subroutines (BLAS)

Since it is not cost-effective to write a special version of every routine like Cholesky for every new computer, we need a more systematic approach. Since operations like matrix-matrix multiplication are so common, computer manufacturers have standardized them as the *Basic Linear Algebra Subroutines*, or *BLAS* [169, 89, 87], and optimized them for their machines. In other words, a library of subroutines for matrix-matrix multiplication, matrix-vector multiplication, and other similar operations is available with a standard Fortran or C interface on high performance machines (and many others), but underneath they have been optimized for each machine. Our goal is to take advantage of these optimized BLAS by reorganizing algorithms like Cholesky so that they call the BLAS to perform most of their work.

In this section we will discuss the BLAS in general. In section 2.6.2, we will describe how to optimize matrix multiplication in particular. Finally, in section 2.6.3, we show how to reorganize Gaussian elimination so that most of its work is performed using matrix multiplication.

Let us examine the BLAS more carefully. Table 2.1 counts the number of memory references and floating points operations performed by three related BLAS. For example, the number of memory references needed to implement the `saxpy` operation in line 1 of the table is $3n + 1$, because we need to read n values of x_i, n values of y_i, and 1 value of α from slow memory to registers, and then write n values of y_i back to slow memory. The last column gives the ratio q of flops to memory references (its highest-order term in n only).

The significance of q is that it tells us roughly how many flops that we can perform per memory reference or how much useful work we can do compared to the time moving data. This tells us how fast the algorithm can *potentially* run. For example, suppose that an algorithm performs f floating points operations, each of which takes t_{arith} seconds, and m memory references, each of which takes t_{mem} seconds. Then the total running time is as large as

$$f \cdot t_{\text{arith}} + m \cdot t_{\text{mem}} = f \cdot t_{\text{arith}} \cdot \left(1 + \frac{m}{f} \frac{t_{\text{mem}}}{t_{\text{arith}}}\right) = f \cdot t_{\text{arith}} \cdot \left(1 + \frac{1}{q} \frac{t_{\text{mem}}}{t_{\text{arith}}}\right),$$

assuming that the arithmetic and memory references are not performed in parallel. Therefore, the larger the value of q, the closer the running time is to the best possible running time $f \cdot t_{\text{arith}}$, which is how long the algorithm would take if all data were in registers. This means that algorithms with the larger q values are better building blocks for other algorithms.

Table 2.1 reflects a hierarchy of operations: Operations such as `saxpy` perform $O(n^1)$ flops on vectors and offer the worst q values; these are called Level 1 BLAS, or BLAS1 [169], and include inner products, multiplying a

Operation	Definition	f	m	$q = f/m$
`saxpy` (BLAS1)	$y = \alpha \cdot x + y$ or $y_i = \alpha x_i + y_i$ $i = 1, \ldots, n$	$2n$	$3n + 1$	$2/3$
Matrix-vector mult (BLAS2)	$y = A \cdot x + y$ or $y_i = \sum_{j=1}^{n} a_{ij} x_j + y_i$ $i = 1, \ldots, n$	$2n^2$	$n^2 + 3n$	2
Matrix-matrix mult (BLAS3)	$C = A \cdot B + C$ or $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{jk} + c_{ij}$ $i, j = 1, \ldots, n$	$2n^3$	$4n^2$	$n/2$

Table 2.1. *Counting floating point operations and memory references for the BLAS. f is the number of floating point operations, and m is the number of memory references.*

scalar times a vector and other simple operations. Operations such as matrix-vector multiplication perform $O(n^2)$ flops on matrices and vectors and offer slightly better q values; these are called Level 2 BLAS, or BLAS2 [89, 88], and include solving triangular systems of equations and rank-1 updates of matrices ($A + xy^T$, x and y column vectors). Operations such as matrix-matrix multiplication perform $O(n^3)$ flops on pairs of matrices and offer the best q values; these are called Level 3 BLAS, or BLAS3 [87, 86], and include solving triangular systems of equations with many right-hand sides.

The directory NETLIB/blas includes documentation and (unoptimized) implementations of all the BLAS. For a quick summary of all the BLAS, see NETLIB/blas/blasqr.ps. This summary also appears in [10, App. C] (or NETLIB/lapack/lug/lapack_lug.html).

Since the Level 3 BLAS have the highest q values, we endeavor to reorganize our algorithms in terms of operations such as matrix-matrix multiplication rather than `saxpy` or matrix-vector multiplication. (LINPACK's Cholesky is constructed in terms of calls to `saxpy`.) We emphasize that such reorganized algorithms will only be faster when using BLAS that have been optimized.

2.6.2. How to Optimize Matrix Multiplication

Let us examine in detail how to implement matrix multiplication $C = A \cdot B + C$ to minimize the number of memory moves and so optimize its performance. We will see that the performance is sensitive to the implementation details. To simplify our discussion, we will use the following machine model. We assume that matrices are stored columnwise, as in Fortran. (It is easy to modify the examples below if matrices are stored rowwise as in C.) We assume that there are two levels of memory hierarchy, fast and slow, where the slow memory is large enough to contain the three $n \times n$ matrices A, B, and C, but the fast memory contains only M words where $2n < M \ll n^2$; this means that

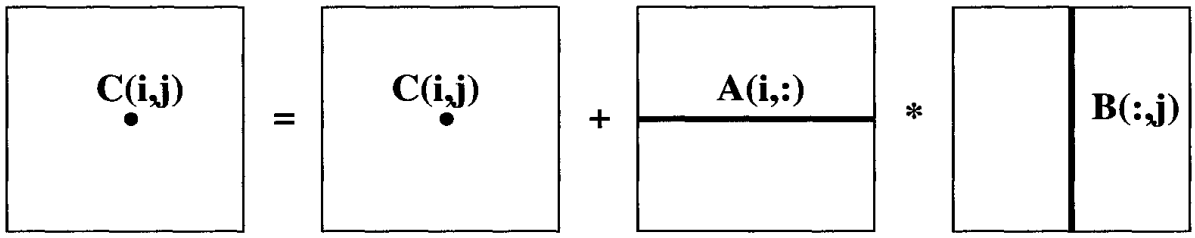
the fast memory is large enough to hold two matrix columns or rows but not a whole matrix. We further assume that the data movement is under programmer control. (In practice, data movement may be done automatically by hardware, such as the cache controller. Nonetheless, the basic optimization scheme remains the same.)

The simplest matrix-multiplication algorithm that one might try consists of three nested loops, which we have annotated to indicate the data movements.

ALGORITHM 2.6. *Unblocked matrix multiplication (annotated to indicate memory activity):*

```
for i = 1 to n
    { Read row i of A into fast memory }
    for j = 1 to n
        { Read C_ij into fast memory }
        { Read column j of B into fast memory }
        for k = 1 to n
            C_ij = C_ij + A_ik · B_kj
        end for
        { Write C_ij back to slow memory }
    end for
end for
```

The innermost loop is doing a dot product of row i of A and column j of B to compute C_{ij}, as shown in the following figure:

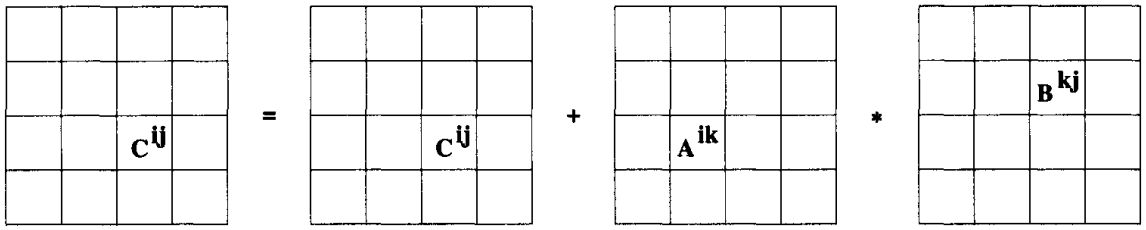


One can also describe the two innermost loops (on j and k) as doing a vector-matrix multiplication of the ith row of A times the matrix B to get the ith row of C. This is a hint that we will not perform any better than these BLAS1 and BLAS2 operations, since they are within the innermost loops.

Here is the detailed count of memory references: n^3 for reading B n times (once for each value of i); n^2 for reading A one row at a time and keeping it in fast memory until it is no longer needed; and $2n^2$ for reading one entry of C at a time, keeping it in fast memory until it is completely computed, and then moving it back to slow memory. This comes to $n^3 + 3n^2$ memory moves, or $q = 2n^3/(n^2+3n^2) \approx 2$, which is no better than the Level 2 BLAS and far from the maximum possible $n/2$ (see Table 2.1). If $M \ll n$, so that we cannot keep a full row of A in fast memory, q further decreases to 1, since the algorithm reduces to a sequence of inner products, which are Level 1 BLAS. For every

permutation of the three loops on i, j, and k, one gets another algorithm with q about the same.

Our preferred algorithm uses *blocking*, where C is broken into an $N \times N$ block matrix with $n/N \times n/N$ blocks C^{ij}, and A and B are similarly partitioned, as shown below for $N = 4$. The algorithm becomes



$$\boxed{C^{ij}} = \boxed{C^{ij}} + \sum_{k=1}^{N} \boxed{A^{ik}} * \boxed{B^{kj}}$$

ALGORITHM 2.7. *Blocked matrix multiplication (annotated to indicate memory activity):*

for $i = 1$ *to* N
  for $j = 1$ *to* N
  { *Read* C^{ij} *into fast memory* }
    for $k = 1$ *to* N
      { *Read* A^{ik} *into fast memory* }
      { *Read* B^{kj} *into fast memory* }
      $C^{ij} = C^{ij} + A^{ik} \cdot B^{kj}$
    end for
  { *Write* C^{ij} *back to slow memory* }
  end for
end for

Our memory reference count is as follows: $2n^2$ for reading and writing each block of C once, Nn^2 for reading A N times (reading each n/N-by-n/N submatrix A^{ik} N^3 times), and Nn^2 for reading B N times (reading each n/N-by-n/N submatrix B^{kj} N^3 times), for a total of $(2N + 2)n^2 \approx 2Nn^2$ memory references. So we want to choose N as small as possible to minimize the number of memory references. But N is subject to the constraint $M \geq 3(n/N)^2$, which means that one block each from A, B, and C must fit in fast memory simultaneously. This yields $N \approx n\sqrt{3/M}$, and so $q \approx (2n^3)/(2Nn^2) \approx \sqrt{M/3}$, which is much better than the previous algorithm. In particular q grows independently of n as M grows, which means that we expect the algorithm to be fast for any matrix size n and to go faster if the fast memory size M is increased. These are both attractive properties.

In fact, it can be shown that Algorithm 2.7 is asymptotically optimal [151]. In other words, no reorganization of matrix-matrix multiplication (that performs the same $2n^3$ arithmetic operations) can have a q larger than $O(\sqrt{M})$.

On the other hand, this brief analysis ignores a number of practical issues:

1. A real code will have to deal with nonsquare matrices, for which the optimal block sizes may not be square.

2. The cache and register structure of a machine will strongly affect the best shapes of submatrices.

3. There may be special hardware instructions that perform both a multiplication and an addition in one cycle. It may also be possible to execute several multiply-add operations simultaneously if they do not interfere.

For a detailed discussion of these issues for one high-performance workstation, the IBM RS6000/590, see [1], PARALLEL_HOMEPAGE, or http://www.rs6000.ibm.com/resource/technology/essl.html. Figure 2.5 shows the speeds of the three basic BLAS for this machine. The horizontal axis is matrix size, and the vertical axis is speed in Mflops. The peak machine speed is 266 Mflops. The top curve (peaking near 250 Mflops) is square matrix-matrix multiplication. The middle curve (peaking near 100 Mflops) is square matrix-vector multiplication, and the bottom curve (peaking near 75 Mflops) is **saxpy**. Note that the speed increases for larger matrices. This is a common phenomenon and means that we will try to develop algorithms whose internal matrix-multiplications use as large matrices as reasonable.

Both the above matrix-matrix multiplication algorithms perform $2n^3$ arithmetic operations. It turns out that there are other implementations of matrix-matrix multiplication that use far fewer operations. Strassen's method [3] was the first of these algorithms to be discovered and is the simplest to explain. This algorithm multiplies matrices recursively by dividing them into 2×2 block matrices and multiplying the subblocks using seven matrix multiplications (recursively) and 18 matrix additions of half the size; this leads to an asymptotic complexity of $n^{\log_2 7} \approx n^{2.81}$ instead of n^3.

ALGORITHM 2.8. *Strassen's matrix multiplication algorithm:*

C = *Strassen(A,B,n)*
  /* *Return* $C = A * B$, *where* A *and* B *are* n*-by-*n*;*
    Assume n *is a power of* 2 */
  if $n = 1$
    return $C = A * B$ /* *scalar multiplication* */
  else
    Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ *and* $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$
      where the subblocks A_{ij} *and* B_{ij} *are* $n/2$*-by-*$n/2$

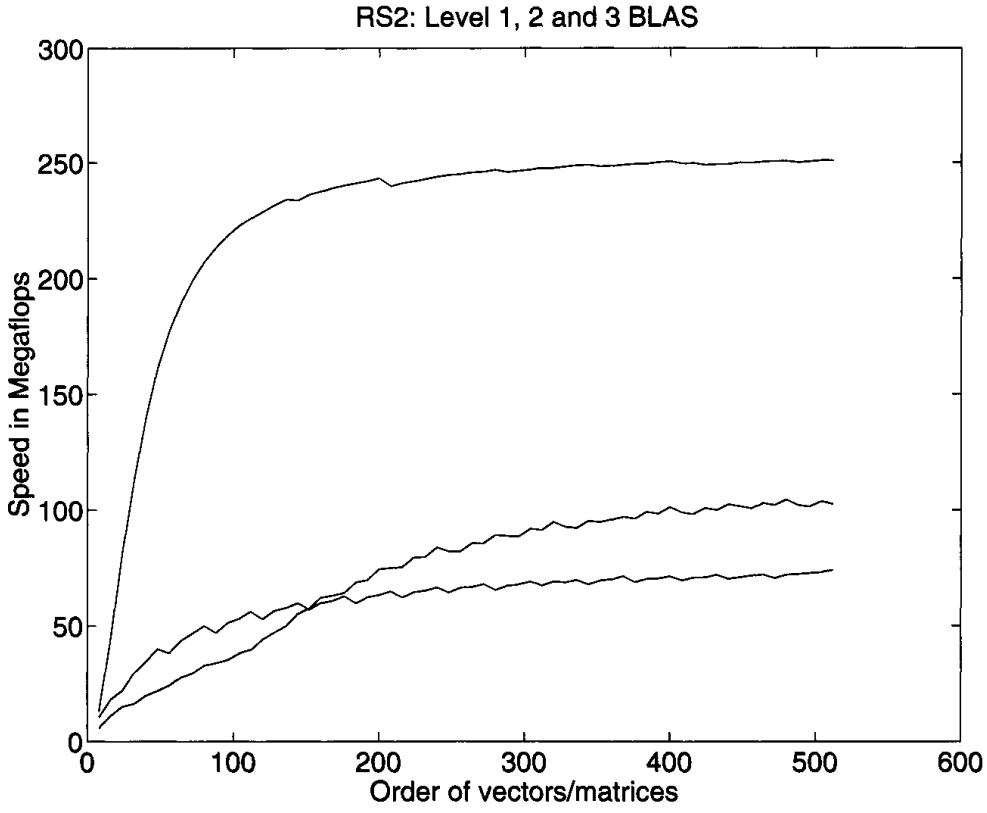


Fig. 2.5. *BLAS speed on the IBM RS* 6000/590.

$$
\begin{aligned}
P_1 &= \mathit{Strassen}(\, A_{12} - A_{22},\ B_{21} + B_{22},\ n/2\,) \\
P_2 &= \mathit{Strassen}(\, A_{11} + A_{22},\ B_{11} + B_{22},\ n/2\,) \\
P_3 &= \mathit{Strassen}(\, A_{11} - A_{21},\ B_{11} + B_{12},\ n/2\,) \\
P_4 &= \mathit{Strassen}(\, A_{11} + A_{12},\ B_{22},\ n/2\,) \\
P_5 &= \mathit{Strassen}(\, A_{11},\ B_{12} - B_{22},\ n/2\,) \\
P_6 &= \mathit{Strassen}(\, A_{22},\ B_{21} - B_{11},\ n/2\,) \\
P_7 &= \mathit{Strassen}(\, A_{21} + A_{22},\ B_{11},\ n/2\,) \\
C_{11} &= P_1 + P_2 - P_4 + P_6 \\
C_{12} &= P_4 + P_5 \\
C_{21} &= P_6 + P_7 \\
C_{22} &= P_2 - P_3 + P_5 - P_7 \\
\mathit{return}\ C &= \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}
\end{aligned}
$$

end if

It is tedious but straightforward to confirm by induction that this algorithm multiplies matrices correctly (see Question 2.21). To show that its complexity is $O(n^{\log_2 7})$, we let $T(n)$ be the number of additions, subtractions, and multiplications performed by the algorithm. Since the algorithm performs seven recursive calls on matrices of size $n/2$, and 18 additions of $n/2$-by-$n/2$ matrices, we can write down the recurrence $T(n) = 7T(n/2) + 18(n/2)^2$. Changing variables

from n to $m = \log_2 n$, we get a new recurrence $\bar{T}(m) = 7\bar{T}(m-1) + 18(2^{m-1})^2$, where $\bar{T}(m) = T(2^m)$. We can confirm that this linear recurrence for $\bar{T}$ has a solution $\bar{T}(m) = O(7^m) = O(n^{\log_2 7})$.

The value of Strassen's algorithm is not just this asymptotic complexity but its reduction of the problem to smaller subproblems which eventually fit in fast memory; once the subproblems fit in fast memory, standard matrix multiplication may be used. This approach has led to speedups on relatively large matrices on some machines [22]. A drawback is the need for significant workspace and somewhat lower numerical stability, although it is adequate for many purposes [77]. There are a number of other even faster matrix multiplication algorithms; the current record is about $O(n^{2.376})$, due to Winograd and Coppersmith [263]. But these algorithms only perform fewer operations than Strassen for impractically large values of n. For a survey see [195].

2.6.3. Reorganizing Gaussian Elimination to Use Level 3 BLAS

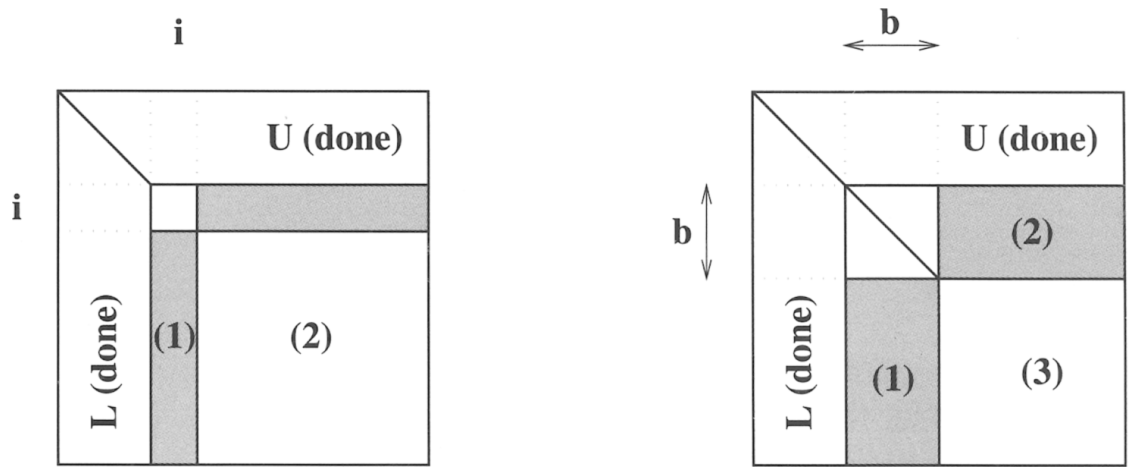
We will reorganize Gaussian elimination to use, first, the Level 2 BLAS and, then, the Level 3 BLAS. For simplicity, we assume that no pivoting is necessary.

Indeed, Algorithm 2.4 is already a Level 2 BLAS algorithm, because most of the work is done in the second line, $A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) * A(i, i+1:n)$, which is a *rank-1 update* of the submatrix $A(i+1:n, i+1:n)$. The other arithmetic in the algorithm, $A(i+1:n, i) = A(i+1:n, i)/A(i, i)$, is actually done by multiplying the vector $A(i+1:n, i)$ by the scalar $1/A(i, i)$, since multiplication is much faster than division; this is also a Level 1 BLAS operation. We need to modify Algorithm 2.4 slightly because we will use it within the Level 3 version.

ALGORITHM 2.9. *Level 2 BLAS implementation of LU factorization without pivoting for an m-by-n matrix A, where $m \geq n$: Overwrite A by the m-by-n matrix L and m-by-m matrix U. We have numbered the important lines for later reference.*

for $i = 1$ *to* $\min(m-1, n)$
(1) $A(i+1:m, i) = A(i+1:m, i)/A(i, i)$
if $i < n$
(2) $A(i+1:m, i+1:n) = A(i+1:m, i+1:n) -$
$A(i+1:m, i) \cdot A(i, i+1:n)$
end for

The left side of Figure 2.6 illustrates Algorithm 2.9 applied to a square matrix. At step i of the algorithm, columns 1 to $i-1$ of L and rows 1 to $i-1$ of U are already done, column i of L and row i of U are to be computed, and the trailing submatrix of A is to be updated by a rank-1 update. On the left side of Figure 2.6, the submatrices are labeled by the lines of the algorithm ((1) or (2)) that update them. The rank-1 update in line (2) is to subtract the



Step i of Level 2 BLAS Implementation of LU

Step i of Level 3 BLAS Implementation of LU

Fig. 2.6. *Level 2 and Level 3 BLAS implementations of LU factorization.*

product of the shaded column and the shaded row from the submatrix labeled (2).

The Level 3 BLAS algorithm will reorganize this computation by *delaying the update* of submatrix (2) for b steps, where b is a small integer called the *block size*, and later applying b rank-1 updates all at once in a single matrix-matrix multiplication. To see how to do this, suppose that we have already computed the first $i-1$ columns of L and rows of U, yielding

$$
\begin{aligned}
A &= \begin{matrix} \\ i-1 \\ b \\ n-b-i+1 \end{matrix}
\begin{matrix} \begin{matrix} i-1 & b & n-b-i+1 \end{matrix} \\
\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \end{matrix} \\
&= \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & 0 & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{21} & U_{31} \\ 0 & \tilde{A}_{22} & \tilde{A}_{23} \\ 0 & \tilde{A}_{32} & \tilde{A}_{33} \end{bmatrix},
\end{aligned}
$$

where all the matrices are partitioned the same way. This is shown on the right side of Figure 2.6. Now apply Algorithm 2.9 to the submatrix $\begin{bmatrix} \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix}$ to get

$$
\begin{bmatrix} \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} \cdot U_{22} = \begin{bmatrix} L_{22}U_{22} \\ L_{32}U_{22} \end{bmatrix}.
$$

This lets us write

$$
\begin{bmatrix} \tilde{A}_{22} & \tilde{A}_{23} \\ \tilde{A}_{32} & \tilde{A}_{33} \end{bmatrix} = \begin{bmatrix} L_{22}U_{22} & \tilde{A}_{23} \\ L_{32}U_{22} & \tilde{A}_{33} \end{bmatrix}
$$

$$
\begin{aligned}
&= \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{22} & L_{22}^{-1}\tilde{A}_{23} \\ 0 & \tilde{A}_{33} - L_{32} \cdot (L_{22}^{-1}\tilde{A}_{23}) \end{bmatrix} \\
&\equiv \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{22} & U_{23} \\ 0 & \tilde{A}_{33} - L_{32} \cdot U_{23} \end{bmatrix} \\
&\equiv \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{22} & U_{23} \\ 0 & \tilde{\tilde{A}}_{33} \end{bmatrix}.
\end{aligned}
$$

Altogether, we get an updated factorization with b more columns of L and rows of U completed:

$$
\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{23} & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{21} & U_{31} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & \tilde{\tilde{A}}_{33} \end{bmatrix}.
$$

This defines an algorithm with the following three steps, which are illustrated on the right of Figure 2.6:

(1) Use Algorithm 2.9 to factorize $\begin{bmatrix} \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} \cdot U_{22}$.

(2) Form $U_{23} = L_{22}^{-1}\tilde{A}_{23}$. This means solving a triangular linear system with many right-hand sides ($\tilde{A}_{23}$), a single Level 3 BLAS operation.

(3) Form $\tilde{\tilde{A}}_{33} = \tilde{A}_{33} - L_{32} \cdot U_{23}$, a matrix-matrix multiplication.

More formally, we have the following algorithm.

ALGORITHM 2.10. *Level 3 BLAS implementation of LU factorization without pivoting for an n-by-n matrix A. Overwrite L and U on A. The lines of the algorithm are numbered as above and to correspond to the right part of Figure* 2.6.

for $i = 1$ *to* $n - 1$ *step* b

(1) *Use Algorithm* 2.9 *to factorize* $A(i:n, i:i+b-1) = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22}$

(2) $A(i:i+b-1, i+b:n) = L_{22}^{-1} \cdot A(i:i+b-1, i+b:n)$
/ form* U_{23} **/*

(3) $A(i+b:n, i+b:n) = A(i+b:n, i+b:n)$
$-A(i+b:n, i:i+b-1) \cdot A(i:i+b-1, i+b:n)$
/ form* $\tilde{\tilde{A}}_{33}$ **/*

end for

We still need to choose the block size b in order to maximize the speed of the algorithm. On the one hand, we would like to make b large because we have seen that speed increases when multiplying larger matrices. On the other hand, we can verify that the number of floating point operations performed

by the slower Level 2 and Level 1 BLAS in line (1) of the algorithm is about $n^2b/2$ for small b, which grows as b grows, so we do not want to pick b too large. The optimal value of b is machine dependent and can be tuned for each machine. Values of $b = 32$ or $b = 64$ are commonly used.

To see detailed implementations of Algorithms 2.9 and 2.10, see subroutines `sgetf2` and `sgetrf`, respectively, in LAPACK (NETLIB/lapack). For more information on block algorithms, including detailed performance number on a variety of machines, see also [10] or the course notes at PARALLEL_HOMEPAGE.

2.6.4. More About Parallelism and Other Performance Issues

In this section we briefly survey other issues involved in implementing Gaussian elimination (and other linear algebra routines) as efficiently as possible.

A *parallel computer* contains $p > 1$ processors capable of simultaneously working on the same problem. One may hope to solve any given problem p times faster on such a machine than on a conventional uniprocessor. But such "perfect efficiency" is rarely achieved, even if there are always at least p independent tasks available to do, because of the overhead of coordinating p processors and the cost of sending data from the processor that may store it to the processor that needs it. This last problem is another example of a *memory hierarchy*: from the point of view of processor i, its own memory is fast, but getting data from the memory owned by processor j is slower, sometimes thousands of times slower.

Gaussian elimination offers many opportunities for parallelism, since each entry of the trailing submatrix may be updated independently and in parallel at each step. But some care is needed to be as efficient as possible. Two standard pieces of software are available. The LAPACK routine `sgetrf` described in the last section [10] runs on *shared-memory parallel machines*, provided that one has available implementations of the BLAS that run in parallel. A related library called ScaLAPACK, for *Scalable LAPACK* [34, 53], is designed for *distributed-memory parallel machines*, i.e., those that require special operations to move data between different processors. All software is available on NETLIB in the LAPACK and ScaLAPACK subdirectories. ScaLAPACK is described in more detail in the notes at PARALLEL_HOMEPAGE. Extensive performance data for linear equation solvers are available as the LINPACK Benchmark [85], with an up-to-date version available at NETLIB/benchmark/performance.ps, or in the Performance Database Server.[14] As of May 1997, the fastest that any linear system had been solved using Gaussian elimination was one with $n = 215000$ on an Intel ASCI Option Red with $p = 7264$ processors; the problem ran at just over 1068 Gflops (gigaflops), out of a maximum 1453 Gflops.

[14] http://performance.netlib.org/performance/html/PDStop.html

There are some matrices too large to fit in the main memory of any available machine. These matrices are stored on disk and must be read into main memory piece by piece in order to perform Gaussian elimination. The organization of such routines is largely similar to the technique described above, and they are included in ScaLAPACK.

Finally, one might hope that compilers would become sufficiently clever to take the simplest implementation of Gaussian elimination using three nested loops and automatically "optimize" the code to look like the blocked algorithm discussed in the last subsection. While there is much current research on this topic (see the bibliography in the recent compiler textbook [264]), there is still no reliably fast alternative to optimized libraries such as LAPACK and ScaLAPACK.

2.7. Special Linear Systems

As mentioned in section 1.2, it is important to exploit any special structure of the matrix to increase speed of solution and decrease storage. In practice, of course, the cost of the extra programming effort required to exploit this structure must be taken into account. For example, if our only goal is to minimize the time to get the desired solution, and it takes an extra week of programming effort to decrease the solution time from 10 seconds to 1 second, it is worth doing only if we are going to use the routine more than (1 week * 7 days/week * 24 hours/day * 3600 seconds/hour) / (10 seconds − 1 second) = 67200 times. Fortunately, there are some special structures that turn up frequently enough that standard solutions exist, and we should certainly use them. The ones we consider here are

1. s.p.d. matrices,
2. symmetric indefinite matrices,
3. band matrices,
4. general sparse matrices,
5. dense matrices depending on fewer than n^2 independent parameters.

We will consider only real matrices; extensions to complex matrices are straightforward.

2.7.1. Real Symmetric Positive Definite Matrices

Recall that a real matrix A is s.p.d. if and only if $A = A^T$ and $x^T Ax > 0$ for all $x \neq 0$. In this section we will show how to solve $Ax = b$ in half the time and half the space of Gaussian elimination when A is s.p.d.

PROPOSITION 2.2. 1. *If X is nonsingular, then A is s.p.d. if and only if X^TAX is s.p.d.*

2. *If A is s.p.d. and H is any principal submatrix of A ($H = A(j:k, j:k)$ for some $j \leq k$), then H is s.p.d.*

3. *A is s.p.d. if and only if $A = A^T$ and all its eigenvalues are positive.*

4. *If A is s.p.d., then all $a_{ii} > 0$, and $\max_{ij} |a_{ij}| = \max_i a_{ii} > 0$.*

5. *A is s.p.d. if and only if there is a unique lower triangular nonsingular matrix L, with positive diagonal entries, such that $A = LL^T$. $A = LL^T$ is called the* Cholesky factorization *of A, and L is called the* Cholesky factor *of A.*

Proof.

1. X nonsingular implies $Xx \neq 0$ for all $x \neq 0$, so $x^TX^TAXx > 0$ for all $x \neq 0$. So A s.p.d. implies X^TAX is s.p.d. Use X^{-1} to deduce the other implication.

2. Suppose first that $H = A(1:m, 1:m)$. Then given any m-vector y, the n-vector $x = [y^T, 0]^T$ satisfies $y^THy = x^TAx$. So if $x^TAx > 0$ for all nonzero x, then $y^THy > 0$ for all nonzero y, and so H is s.p.d. If H does not lie in the upper left corner of A, let P be a permutation so that H does lie in the upper left corner of P^TAP and apply Part 1.

3. Let X be the real, orthogonal eigenvector matrix of A so that $X^TAX = \Lambda$ is the diagonal matrix of real eigenvalues λ_i. Since $x^T\Lambda x = \sum_i \lambda_i x_i^2$, Λ is s.p.d if and only if each $\lambda_i > 0$. Now apply Part 1.

4. Let e_i be the ith column of the identity matrix. Then $e_i^TAe_i = a_{ii} > 0$ for all i. If $|a_{kl}| = \max_{ij} |a_{ij}|$ but $k \neq l$, choose $x = e_k - \text{sign}(a_{kl})e_l$. Then $x^TAx = a_{kk} + a_{ll} - 2|a_{kl}| \leq 0$, contradicting positive-definiteness.

5. Suppose $A = LL^T$ with L nonsingular. Then $x^TAx = (x^TL)(L^Tx) = \|L^Tx\|_2^2 > 0$ for all $x \neq 0$, so A is s.p.d. If A is s.p.d., we show that L exists by induction on the dimension n. If we choose each $l_{ii} > 0$, our construction will determine L uniquely. If $n = 1$, choose $l_{11} = \sqrt{a_{11}}$, which exists since $a_{11} > 0$. As with Gaussian elimination, it suffices to understand the block 2-by-2 case. Write

$$
\begin{aligned}
A &= \begin{bmatrix} a_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} \\
&= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{A_{12}^T}{\sqrt{a_{11}}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \frac{A_{12}}{\sqrt{a_{11}}} \\ 0 & I \end{bmatrix}
\end{aligned}
$$

$$= \begin{bmatrix} a_{11} & A_{12} \\ A_{12}^T & \tilde{A}_{22} + \frac{A_{12}^T A_{12}}{a_{11}} \end{bmatrix},$$

so the $(n-1)$-by-$(n-1)$ matrix $\tilde{A}_{22} = A_{22} - \frac{A_{12}^T A_{12}}{a_{11}}$ is symmetric.

By Part 1 above, $\begin{bmatrix} 1 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix}$ is s.p.d, so by Part 2 $\tilde{A}_{22}$ is s.p.d.

Thus by induction there exists an $\tilde{L}$ such that $\tilde{A}_{22} = \tilde{L}\tilde{L}^T$ and

$$\begin{aligned} A &= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{A_{12}^T}{\sqrt{a_{11}}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{L}\tilde{L}^T \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \frac{A_{12}}{\sqrt{a_{11}}} \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{A_{12}^T}{\sqrt{a_{11}}} & \tilde{L} \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \frac{A_{12}}{\sqrt{a_{11}}} \\ 0 & \tilde{L}^T \end{bmatrix} \equiv LL^T. \quad \square \end{aligned}$$

We may rewrite this induction as the following algorithm.

ALGORITHM 2.11. *Cholesky algorithm:*

for $j = 1$ *to* n
  $l_{jj} = (a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2)^{1/2}$
  for $i = j+1$ *to* n
    $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk})/l_{jj}$
  end for
end for

If A is not positive definite, then (in exact arithmetic) this algorithm will fail by attempting to compute the square root of a negative number or by dividing by zero; this is the cheapest way to test if a symmetric matrix is positive definite.

As with Gaussian elimination, L can overwrite the lower half of A. Only the lower half of A is referred to by the algorithm, so in fact only $n(n+1)/2$ storage is needed instead of n^2. The number of flops is

$$\sum_{j=1}^{n} \left(2j + \sum_{i=j+1}^{n} 2j \right) = \frac{1}{3}n^3 + O(n^2),$$

or just half the flops of Gaussian elimination. Just as with Gaussian elimination, Cholesky may be reorganized to perform most of its floating point operations using Level 3 BLAS; see LAPACK routine `spotrf`.

Pivoting is not necessary for Cholesky to be numerically stable (equivalently, we could also say any diagonal pivot order is numerically stable). We show this as follows. The same analysis as for Gaussian elimination in section 2.4.2 shows that the computed solution $\hat{x}$ satisfies $(A + \delta A)\hat{x} = b$ with

$|\delta A| \leq 3n\varepsilon|L| \cdot |L^T|$. But by the Cauchy–Schwartz inequality and Part 4 of Proposition 2.2

$$\begin{aligned}(|L| \cdot |L^T|)_{ij} &= \sum_k |l_{ik}| \cdot |l_{jk}| \\ &\leq \sqrt{\sum l_{ik}^2}\sqrt{\sum l_{jk}^2} \\ &= \sqrt{a_{ii}} \cdot \sqrt{a_{jj}} \\ &\leq \max_{ij} |a_{ij}|, \end{aligned} \tag{2.16}$$

so $\| \, |L| \cdot |L^T| \, \|_\infty \leq n\|A\|_\infty$ and $\|\delta A\|_\infty \leq 3n^2\varepsilon\|A\|_\infty$.

2.7.2. Symmetric Indefinite Matrices

The question of whether we can still save half the time and half the space when solving a symmetric but indefinite (neither positive definite nor negative definite) linear system naturally arises. It turns out to be possible, but a more complicated pivoting scheme and factorization is required. If A is nonsingular, one can show that there exists a permutation P, a unit lower triangular matrix L, and a block diagonal matrix D with 1-by-1 and 2-by-2 blocks such that $PAP^T = LDL^T$. To see why 2-by-2 blocks are needed in D, consider the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$. This factorization can be computed stably, saving about half the work and space compared to standard Gaussian elimination. The name of the LAPACK subroutine which does this operation is `ssysv`. The algorithm is described in [44].

2.7.3. Band Matrices

A matrix A is called a *band matrix* with *lower bandwidth* b_L and *upper bandwidth* b_U if $a_{ij} = 0$ whenever $i > j + b_L$ or $i < j - b_U$:

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1,b_U+1} & & 0 \\ \vdots & & & \ddots & \\ a_{b_L+1,1} & & & & a_{n-b_U,n} \\ & a_{b_L+2,2} & & & \vdots \\ & & \ddots & & \\ 0 & & & a_{n,n-b_L} \cdots & a_{n,n} \end{bmatrix}.$$

Band matrices arise often in practice (we give an example later) and are useful to recognize because their L and U factors are also "essentially banded," making them cheaper to compute and store. We explain what we mean by "essentially banded" below. But first, we consider LU factorization without pivoting and show that L and U are banded in the usual sense, with the same bandwidths as A.

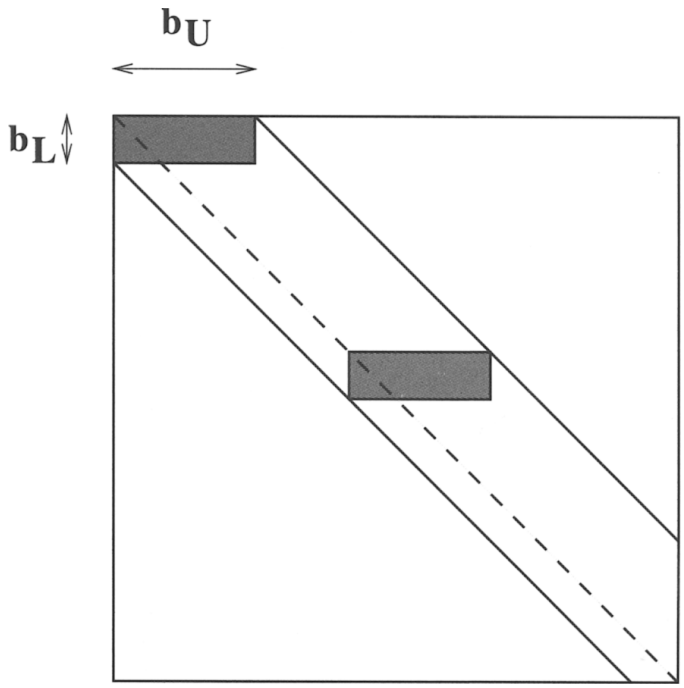


Fig. 2.7. *Band LU factorization without pivoting.*

PROPOSITION 2.3. *Let A be banded with lower bandwidth b_L and upper bandwidth b_U. Let $A = LU$ be computed without pivoting. Then L has lower bandwidth b_L and U has upper bandwidth b_U. L and U can be computed in about $2n \cdot b_U \cdot b_L$ arithmetic operations when b_U and b_L are small compared to n. The space needed is $(b_L + b_U + 1)$. The full cost of solving $Ax = b$ is $2nb_U \cdot b_L + 2nb_U + 2nb_L$.*

Sketch of Proof. It suffices to look at one step; see Figure 2.7. At step j of Gaussian elimination, the shaded region is modified by subtracting the product of the first column and first row of the shaded region; note that this does not enlarge the bandwidth. □

PROPOSITION 2.4. *Let A be banded with lower bandwidth b_L and upper bandwidth b_U. Then after Gaussian elimination with partial pivoting, U is banded with upper bandwidth at most $b_L + b_U$, and L is "essentially banded" with lower bandwidth b_L. This means that L has at most $b_L + 1$ nonzeros in each column and so can be stored in the same space as a band matrix with lower bandwidth b_L.*

Sketch of Proof. Again a picture of the region changed by one step of the algorithm illustrates the proof. As illustrated in Figure 2.8, pivoting can increase the upper bandwidth by at most b_L. Later permutations can reorder the entries of earlier columns so that entries of L may lie below subdiagonal b_L but no new nonzeros can be introduced, so the storage needed for L remains b_L per column. □

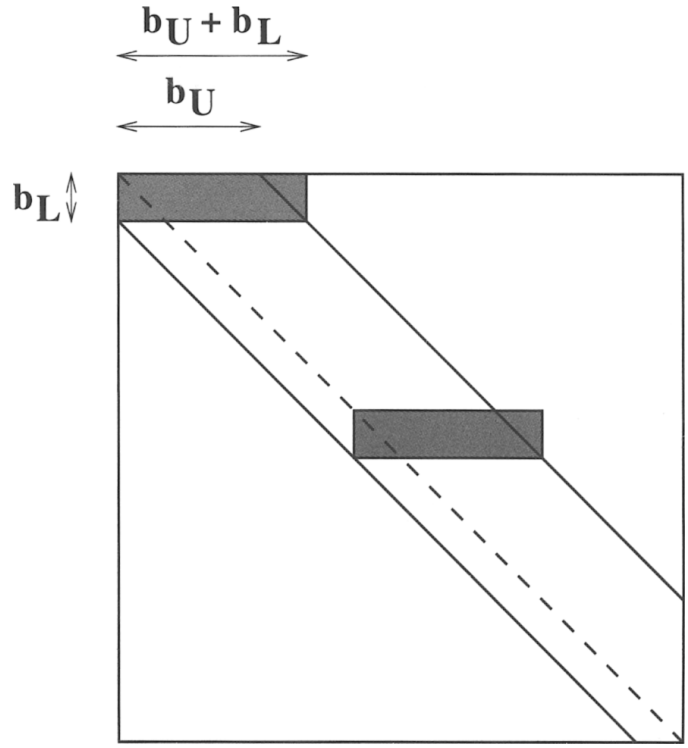


Fig. 2.8. *Band LU factorization with partial pivoting.*

Gaussian elimination and Cholesky for band matrices are available in LAPACK routines like `ssbsv` and `sspsv`.

Band matrices often arise from discretizing physical problems with nearest neighbor interactions on a mesh (provided the unknowns are ordered rowwise or columnwise; see also Example 2.9 and section 6.3).

EXAMPLE 2.8. Consider the ordinary differential equation (ODE) $y''(x) - p(x)y'(x) - q(x)y(x) = r(x)$ on the interval $[a, b]$ with boundary conditions $y(a) = \alpha$, $y(b) = \beta$. We also assume $q(x) \geq \underline{q} > 0$. This equation may be used to model the heat flow in a long, thin rod, for example. To solve the differential equation numerically, we *discretize* it by seeking its solution only at the evenly spaced mesh points $x_i = a + ih$, $i = 0, \ldots, N+1$, where $h = (b-a)/(N+1)$ is the mesh spacing. Define $p_i = p(x_i)$, $r_i = r(x_i)$, and $q_i = q(x_i)$. We need to derive equations to solve for our desired approximations $y_i \approx y(x_i)$, where $y_0 = \alpha$ and $y_{N+1} = \beta$. To derive these equations, we approximate the derivative $y'(x_i)$ by the following *finite difference approximation*:

$$y'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}.$$

(Note that as h gets smaller, the right-hand side approximates $y'(x_i)$ more and more accurately.) We can similarly approximate the second derivative by

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}.$$

(See section 6.3.1 in Chapter 6 for a more detailed derivation.)

Inserting these approximations into the differential equation yields

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} - p_i \frac{y_{i+1} - y_{i-1}}{2h} - q_i y_i = r_i, \quad 1 \le i \le N.$$

Rewriting this as a linear system we get $Ay = b$, where

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}, \quad b = \frac{-h^2}{2} \begin{bmatrix} r_1 \\ \vdots \\ r_N \end{bmatrix} + \begin{bmatrix} (\frac{1}{2} + \frac{h}{4}p_1)\alpha \\ 0 \\ \vdots \\ 0 \\ (\frac{1}{2} - \frac{h}{4}p_N)\beta \end{bmatrix},$$

and

$$A = \begin{bmatrix} a_1 & -c_1 & & \\ -b_2 & \ddots & \ddots & \\ & \ddots & \ddots & c_{N-1} \\ & & -b_N & a_N \end{bmatrix}, \quad \begin{aligned} a_i &= 1 + \tfrac{h^2}{2} q_i, \\ b_i &= \tfrac{1}{2}[1 + \tfrac{h}{2} p_i], \\ c_i &= \tfrac{1}{2}[1 - \tfrac{h}{2} p_i]. \end{aligned}$$

Note that $a_i > 0$, and also $b_i > 0$ and $c_i > 0$ if h is small enough.

This is a nonsymmetric *tridiagonal* system to solve for y. We will show how to change it to a symmetric positive definite tridiagonal system, so that we may use *band Cholesky* to solve it.

Choose $D = \mathrm{diag}(1, \sqrt{\frac{c_1}{b_2}}, \sqrt{\frac{c_1 c_2}{b_2 b_3}}, \ldots, \sqrt{\frac{c_1 c_2 \cdots c_{N-1}}{b_2 b_3 \cdots b_N}})$. Then we may change $Ay = b$ to $(DAD^{-1})(Dy) = Db$ or $\tilde{A}\tilde{y} = \tilde{b}$, where

$$\tilde{A} = \begin{bmatrix} a_1 & -\sqrt{c_1 b_2} & & \\ -\sqrt{c_1 b_2} & a_2 & -\sqrt{c_2 b_3} & \\ & -\sqrt{c_2 b_3} & \ddots & \\ & \ddots & \ddots & -\sqrt{c_{N-1} b_N} \\ & & -\sqrt{c_{N-1} b_N} & a_N \end{bmatrix}.$$

It is easy to see that $\tilde{A}$ is symmetric, and it has the same eigenvalues as A because A and $\tilde{A} = DAD^{-1}$ are *similar*. (See section 4.2 in Chapter 4 for details.) We will use the next theorem to show it is also positive definite.

THEOREM 2.9. Gershgorin. *Let B be an arbitrary matrix. Then the eigenvalues λ of B are located in the union of the n disks*

$$|\lambda - b_{kk}| \le \sum_{j \ne k} |b_{kj}|.$$

Proof. Given λ and $x \neq 0$ such that $Bx = \lambda x$, let $1 = \|x\|_\infty = x_k$ by scaling x if necessary. Then $\sum_{j=1}^{N} b_{kj}x_j = \lambda x_k = \lambda$, so $\lambda - b_{kk} = \sum_{\substack{j=1 \\ j\neq k}}^{N} b_{kj}x_j$, implying

$$|\lambda - b_{kk}| \leq \sum_{j\neq k} |b_{kj}x_j| \leq \sum_{j\neq k} |b_{kj}|. \quad \square$$

Now if h is so small that for all i, $|\frac{h}{2}p_i| < 1$, then

$$|b_i| + |c_i| = \frac{1}{2}\left(1 + \frac{h}{2}p_i\right) + \frac{1}{2}\left(1 - \frac{h}{2}p_i\right) = 1 < 1 + \frac{h^2}{2}\underline{q} \leq 1 + \frac{h^2}{2}q_i = a_i.$$

Therefore all eigenvalues of A lie inside the disks centered at $1 + h^2 q_i/2 \geq 1 + h^2\underline{q}/2$ with radius 1; in particular, they must all have positive real parts. Since $\tilde{A}$ is symmetric, its eigenvalues are real and hence positive, so $\tilde{A}$ is positive definite. Its smallest eigenvalue is bounded below by $\underline{q}h^2/2$. Thus, it can be solved by Cholesky. The LAPACK subroutine for solving a symmetric positive definite tridiagonal system is `sptsv`.

In section 4.3 we will again use Gershgorin's theorem to compute perturbation bounds for eigenvalues of matrices. $\diamond$

2.7.4. General Sparse Matrices

A sparse matrix is defined to be a matrix with a large number of zero entries. In practice, this means a matrix with enough zero entries that it is worth using an algorithm that avoids storing or operating on the zero entries. Chapter 6 is devoted to methods for solving sparse linear systems other than Gaussian elimination and its variants. There are a large number of sparse methods, and choosing the best one often requires substantial knowledge about the matrix [24]. In this section we will only sketch the basic issues in sparse Gaussian elimination and give pointers to the literature and available software.

To give a very simple example, consider the following matrix, which is ordered so that GEPP does not permute any rows:

$$A = \begin{bmatrix} 1 & & & & .1 \\ & 1 & & & .1 \\ & & 1 & & .1 \\ & & & 1 & .1 \\ .1 & .1 & .1 & .1 & 1 \end{bmatrix} = LU$$

$$= \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ .1 & .1 & .1 & .1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & & & & .1 \\ & 1 & & & .1 \\ & & 1 & & .1 \\ & & & 1 & .1 \\ & & & & .96 \end{bmatrix}.$$

A is called an *arrow matrix* because of the pattern of its nonzero entries. Note that none of the zero entries of A were *filled in* by GEPP so that L and U together can be stored in the same space as the nonzero entries of A. Also, if we count the number of essential arithmetic operations, i.e., not multiplication by zero or adding zero, there are only 12 of them (4 divisions to compute the last row of L and 8 multiplications and additions to update the (5,5) entry), instead of $\frac{2}{3}n^3 \approx 83$. More generally, if A were an n-by-n arrow matrix, it would take only $3n-2$ locations to store it instead of n^2, and $3n-3$ floating point operations to perform Gaussian elimination instead of $\frac{2}{3}n^3$. When n is large, both the space and operation count become tiny compared to a dense matrix.

Suppose that instead of A we were given A', which is A with the order of its rows and columns reversed. This amounts to reversing the order of the equations and of the unknowns in the linear system $Ax = b$. GEPP applied to A' again permutes no rows, and to two decimal places we get

$$A' = \begin{bmatrix} 1 & .1 & .1 & .1 & .1 \\ .1 & 1 & & & \\ .1 & & 1 & & \\ .1 & & & 1 & \\ .1 & & & & 1 \end{bmatrix} = L'U'$$

$$= \begin{bmatrix} 1 & & & & \\ .1 & 1 & & & \\ .1 & -.01 & 1 & & \\ .1 & -.01 & -.01 & 1 & \\ .1 & -.01 & -.01 & -.01 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & .1 & .1 & .1 & .1 \\ & .99 & -.01 & -.01 & -.01 \\ & & .99 & -.01 & -.01 \\ & & & .99 & -.01 \\ & & & & .99 \end{bmatrix}.$$

Now we see that L' and U' have filled in completely and require n^2 storage. Indeed, after the first step of the algorithm all the nonzeros of A' have filled in, so we must do the same work as dense Gaussian elimination, $\frac{2}{3}n^3$.

This illustrates that the order of the rows and columns is extremely important for saving storage and work. Even if we do not have to worry about pivoting for numerical stability (such as in Cholesky), choosing the optimal permutations of rows and columns to minimize storage or work is an extremely hard problem. In fact, it is NP-complete [111], which means that all known algorithms for finding the optimal permutation run in time which grows *exponentially* with n and so are vastly more expensive than even dense Gaussian elimination for large n. Thus we must settle for using heuristics, of which there are several successful candidates. We illustrate some of these below.

In addition to the complication of choosing a good row and column permutation, there are other reasons sparse Gaussian elimination or Cholesky are much more complicated than their dense counterparts. First, we need to design a data structure that holds only the nonzero entries of A; there are several in common use [93]. Next, we need a data structure to accommodate new entries

of L and U that fill in during elimination. This means that either the data structure must grow dynamically during the algorithm or we must cheaply precompute it without actually performing the elimination. Finally, we must use the data structure to perform only the minimum number of floating point operations and at most proportionately many integer and logical operations. In other words, we cannot afford to do $O(n^3)$ integer and logical operations to discover the few floating point operations that we want to do. A more complete discussion of these algorithms is beyond the scope of this book [114, 93], but we will indicate available software.

EXAMPLE 2.9. We illustrate sparse Cholesky on a more realistic example that arises from modeling the displacement of a mechanical structure subject to external forces. Figure 2.9 shows a simple mesh of a two-dimensional slice of a mechanical structure with two internal cavities. The mathematical problem is to compute the displacements of all the grid points of the mesh (which are internal to the structure) subject to some forces applied to the boundary of the structure. The mesh points are numbered from 1 to $n = 483$; more realistic problems would have much larger values of n. The equations relating displacements to forces leads to a system of linear equations $Ax = b$, with one row and column for each of the 483 mesh points and with $a_{ij} \neq 0$ if and only if mesh point i is connected by a line segment to mesh point j. This means that A is a symmetric matrix; it also turns out to be positive definite, so that we can use Cholesky to solve $Ax = b$. Note that A has only $nz = 3971$ nonzeros of a possible $483^2 = 233289$, so A is just $3971/233289 = 1.7\%$ filled. (See Examples 4.1 and 5.1 for similar mechanical modeling problems, where the matrix A is derived in detail.)

Figure 2.10 shows the same mesh (above) along with the nonzero pattern of the matrix A (below), where the 483 nodes are ordered in the "natural" way, with the logically rectangular substructures numbered rowwise, one substructure after the other. The edges in each such substructure have a common color, and these colors match the colors of the nonzeros in the matrix. Each substructure has a label "$(i : j)$" to indicate that it corresponds to rows and columns i through j of A. The corresponding submatrix $A(i : j, i : j)$ is a narrow band matrix. (Example 2.8 and section 6.3 describe other situations in which a mesh leads to a band matrix.) The edges connecting different substructures are red and correspond to the red entries of A, which are farthest from the diagonal of A.

The top pair of plots in Figure 2.11 again shows the sparsity structure of A in the natural order, along with the sparsity structure of its Cholesky factor L. Nonzero entries of L corresponding to nonzero entries of A are black; new nonzeros of L, called *fill-in*, are red. L has 11533 nonzero entries, over five times as many as the lower triangle of A. Computing L by Cholesky costs just 296923 flops, just .8% of the $\frac{1}{3}n^3 = 3.76 \cdot 10^7$ flops that dense Cholesky would have required.

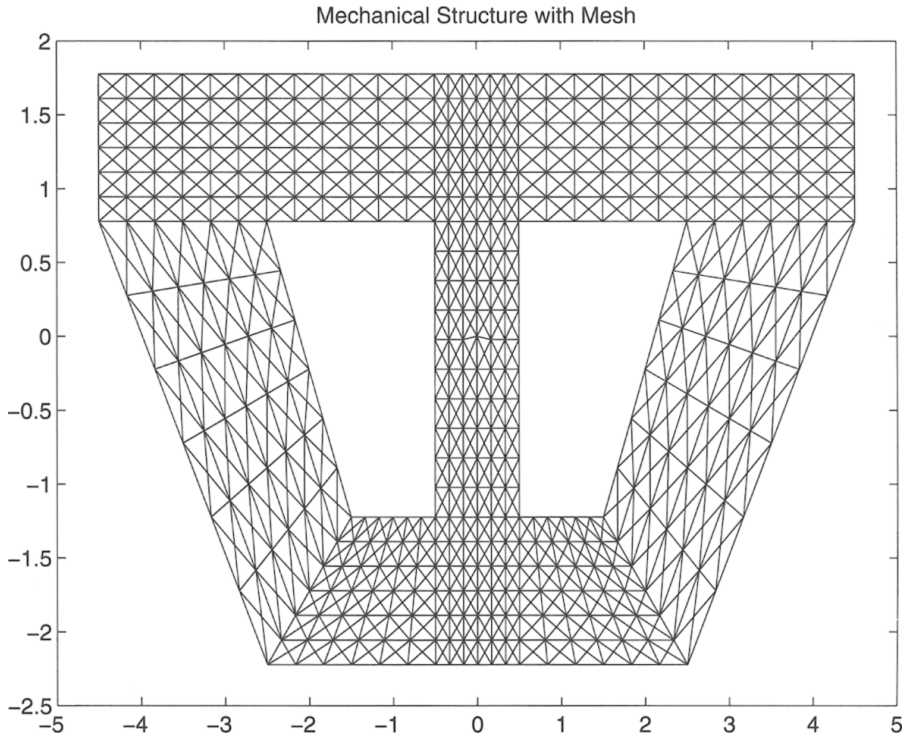


Fig. 2.9. *Mesh for a mechanical structure.*

The number of nonzeros in L and the number of flops required to compute L can be changed significantly by reordering the rows and columns of A. The middle pair of plots in Figure 2.11 shows the results of one such popular reordering, called *reverse Cuthill–McKee* [114, 93], which is designed to make A a narrow band matrix. As can be seen, it is quite successful at this, reducing the fill-in of L 21% (from 11533 to 9073) and reducing the flop count almost 39% (from 296923 to 181525).

Another popular ordering algorithm is called *minimum degree ordering* [114, 93], which is designed to create as little fill-in at each step of Cholesky as possible. The results are shown in the bottom pair of plots in Figure 2.11: the fill-in of L is reduced a further 7% (from 9073 to 8440) but the flop count is increased 9% (from 181525 to 198236). ◇

Many sparse matrix examples are available as built-in demos in Matlab, which also has many sparse matrix operations built into it (type "help sparfun" in Matlab for a list). To see the examples, type demo in Matlab, then click on "continue," then on "Matlab/Visit," and then on either "Matrices/Select a demo/Sparse" or "Matrices/Select a demo/Cmd line demos." For example, Figure 2.12 shows a Matlab example of a mesh around a wing, where the goal is to compute the airflow around the wing at the mesh points. The corresponding partial differential equations of airflow lead to a nonsymmetric linear system whose sparsity pattern is also shown.

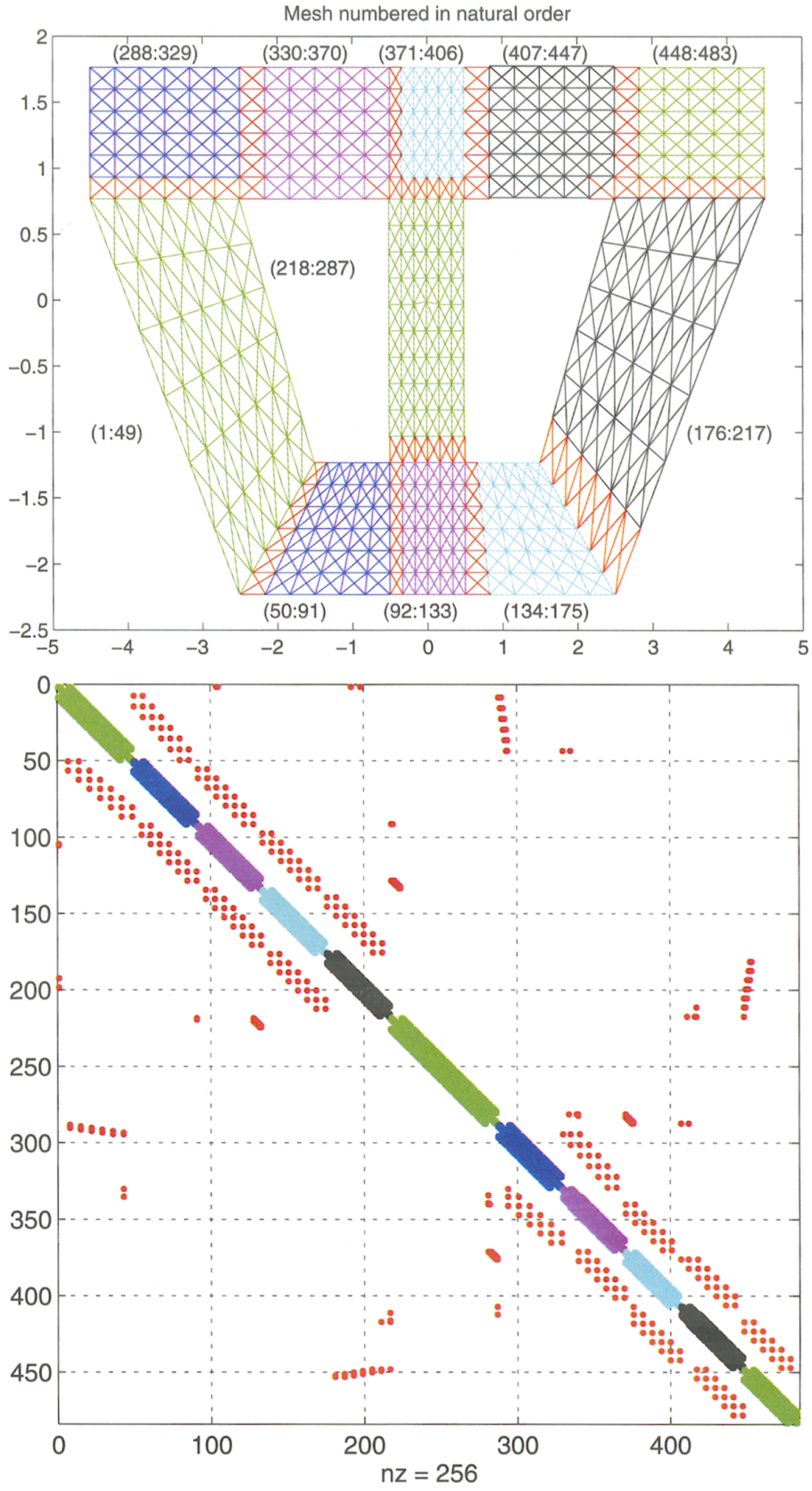


Fig. 2.10. *The edges in the mesh at the top are colored and numbered to match the sparse matrix A at the bottom. For example the first* 49 *nodes of the mesh (the leftmost green nodes) correspond to rows and columns* 1 *through* 49 *of A.*

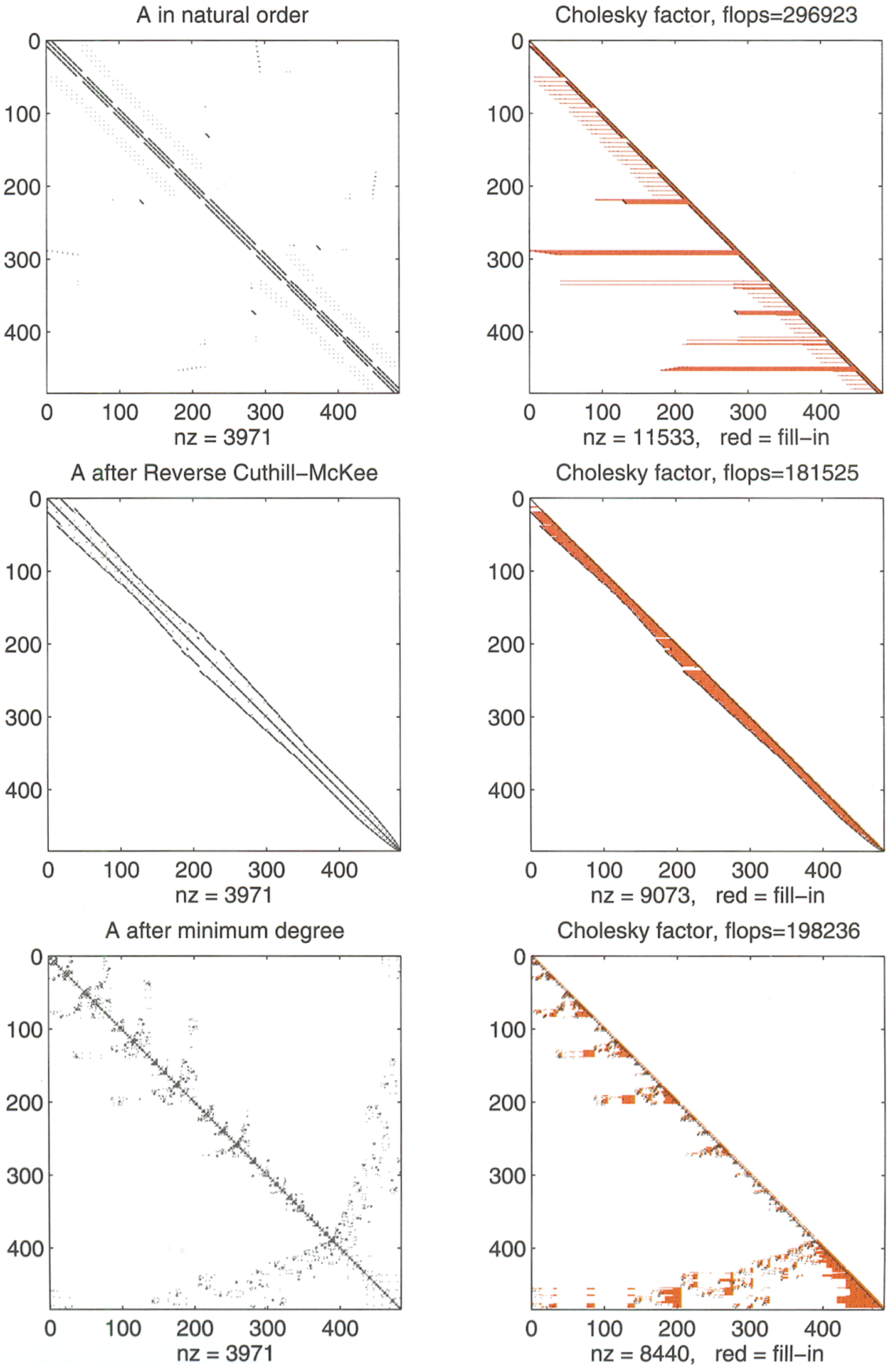


Fig. 2.11. *Sparsity and flop counts for A with various orderings.*

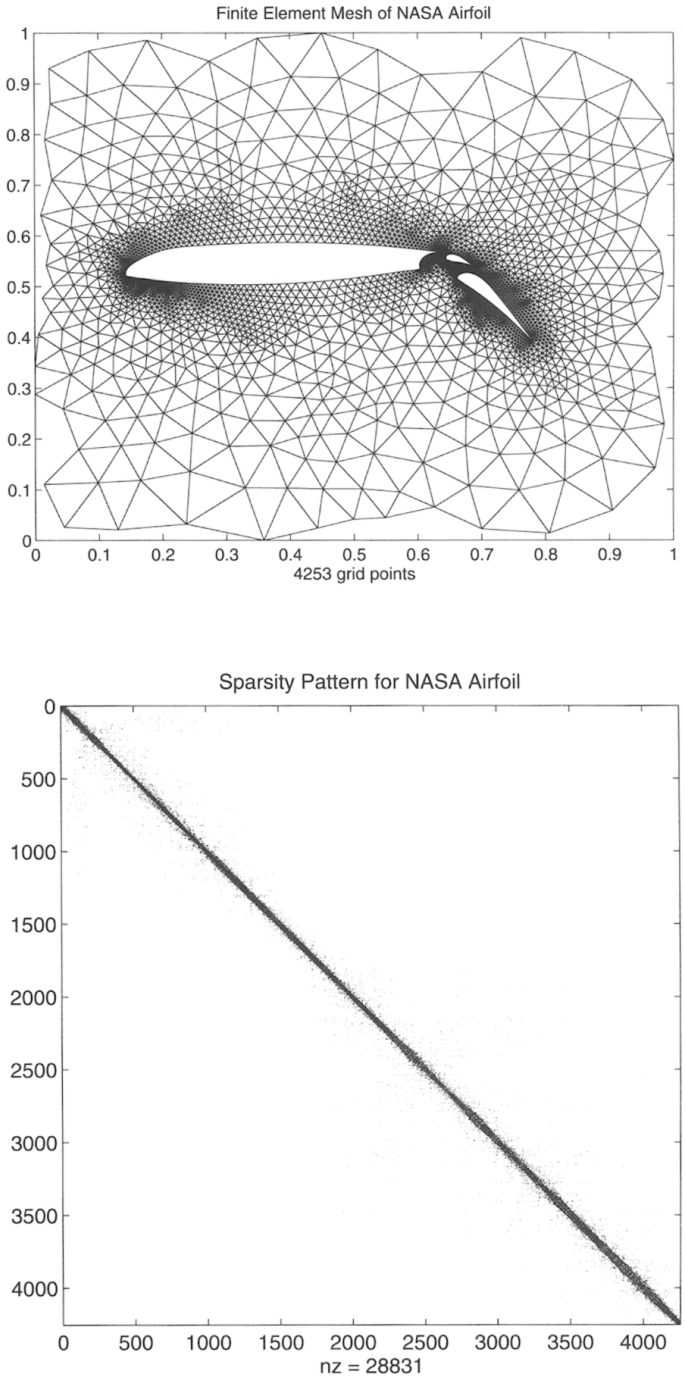


Fig. 2.12. *Mesh around the NASA airfoil.*

Sparse Matrix Software

Besides Matlab, there is a variety of public domain and commercial sparse matrix software available in Fortran or C. Since this is still an active research area (especially with regard to high-performance machines), it is impossible to recommend a single best algorithm. Table 2.2 [177] gives a list of available software, categorized in several ways. We restrict ourselves to supported codes (either public or commercial) or else research codes when no other software is available for that type of problem or machine. We refer to [177, 94] for more complete lists and explanations of the algorithms below.

Table 2.2 is organized as follows. The top group of routines, labeled *serial algorithms*, are designed for single-processor workstations and PCs. The *shared-memory algorithms* are for symmetric multiprocessors, such as the Sun SPARCcenter 2000 [238], SGI Power Challenge [223], DEC AlphaServer 8400 [103], and Cray C90/J90 [253, 254]. The *distributed-memory algorithms* are for machines such as the IBM SP-2 [256], Intel Paragon [257], Cray T3 series [255], and networks of workstations [9]. As you can see, most software has been written for serial machines, some for shared-memory machines, and very little (besides research software) for distributed memory.

The first column gives the *matrix type*. The possibilities include nonsymmetric, symmetric pattern (i.e., either $a_{ij} = a_{ji} \neq 0$, or both can be nonzero and unequal), symmetric (and possibly indefinite), and symmetric positive definite (s.p.d.). The second column gives the name of the routine or of the authors.

The third column gives some detail on the algorithm, indeed more than we have explained in detail in the text: LL (left looking), RL (right looking), frontal, MF (multifrontal), and LDL^T refer to different ways to organize the three nested loops defining Gaussian elimination. Partial, Markowitz, and threshold refer to different pivoting strategies. 2D-blocking refers to which parallel processors are responsible for which parts of the matrix. CAPSS assumes that the linear system is defined by a grid and requires the x, y, and z coordinates of the grid points in order to distribute the matrix among the processors.

The third column also describes the organization of the innermost loop, which could be BLAS1, BLAS2, BLAS3, or scalar. SD refers to the algorithm switching to dense Gaussian elimination after step k when the trailing $(n-k$-by-$(n-k)$ submatrix is dense enough.

The fifth column describes the status and availability of the software, including whether it is public or commercial and how to get it.

2.7.5. Dense Matrices Depending on Fewer Than $O(n^2)$ Parameters

This is a catch-all heading, which includes a large variety of matrices that arise in practice. We mention just a few cases.

Matrix type	Name	Algorithm	Status/ source
Serial algorithms			
nonsym.	SuperLU	LL, partial, BLAS-2.5	Pub/NETLIB
nonsym.	UMFPACK [62, 63]	MF, Markowitz, BLAS-3	Pub/NETLIB
	MA38 (same as UMFPACK)		Com/HSL
nonsym.	MA48 [96]	Anal: RL, Markowitz Fact: LL, partial, BLAS-1, SD	Com/HSL
nonsym.	SPARSE [167]	RL, Markowitz, scalar	Pub/NETLIB
sym-pattern	MUPS [5]	MF, threshold, BLAS-3	Com/HSL
sym-pattern	MA42 [98]	Frontal, BLAS-3	Com/HSL
sym.	MA27 [97]/MA47 [95]	MF, LDL^T, BLAS-1/BLAS-3	Com/HSL
s.p.d.	Ng & Peyton [191]	LL, BLAS-3	Pub/Author
Shared-memory algorithms			
nonsym.	SuperLU	LL, partial, BLAS-2.5	Pub/UCB
nonsym.	PARASPAR [270, 271]	RL, Markowitz, BLAS-1, SD	Res/Author
sym-pattern	MUPS [6]	MF, threshold, BLAS-3	Res/Author
nonsym.	George & Ng [115]	RL, partial, BLAS-1	Res/Author
s.p.d.	Gupta et al. [133]	LL, BLAS-3	Com/SGI Pub/Author
s.p.d.	SPLASH [155]	RL, 2-D block, BLAS-3	Pub/Stanford
Distributed-memory algorithms			
sym.	van der Stappen [245]	RL, Markowitz, scalar	Res/Author
sym-pattern	Lucas et al. [180]	MF, no pivoting, BLAS-1	Res/Author
s.p.d.	Rothberg & Schreiber [207]	RL, 2-D block, BLAS-3	Res/Author
s.p.d.	Gupta & Kumar [132]	MF, 2-D block, BLAS-3	Res/Author
s.p.d.	CAPSS [143]	MF, full parallel, BLAS-1 (require coordinates)	Pub/NETLIB

Table 2.2. *Software to solve sparse linear systems using direct methods.*
Abbreviations used in the table:
nonsym. = nonsymmetric.
sym-pattern = symmetric nonzero structure, nonsymmetric values.
sym. = symmetric and may be indefinite.
s.p.d. = symmetric and positive definite.
MF, LL, and RL = multifrontal, left-looking, and right-looking.
SD = switches to a dense code on a sufficiently dense trailing submatrix.
Pub = publicly available; authors may help use the code.
Res = published in literature but may not be available from the authors.
Com = commercial.
HSL = Harwell Subroutine Library:
http://www.rl.ac.uk/departments/ccd/numerical/hsl/hsl.html.
UCB = http://www.cs.berkeley.edu/~xiaoye/superlu.html.
Stanford = http://www-flash.stanford.edu/apps/SPLASH/.

Vandermonde matrices are of the form

$$V = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & & x_n \\ x_0^2 & x_1^2 & & x_n^2 \\ \vdots & \vdots & & \vdots \\ x_0^{n-1} & x_1^{n-1} & & x_n^{n-1} \end{bmatrix}.$$

Note that the matrix-vector multiplication

$$V^T \cdot [a_0, \ldots, a_n]^T = \left[\sum a_i x_0^i, \ldots, \sum a_i x_n^i\right]^T$$

is equivalent to polynomial evaluation; therefore, solving $V^T a = y$ is polynomial interpolation. Using Newton interpolation we can solve $V^T a = y$ in $\frac{5}{2}n^2$ instead of $\frac{2}{3}n^3$ flops. There is a similar trick to solve $Va = y$ in $\frac{5}{2}n^2$ flops too. See [121, p. 178].

Cauchy matrices C have entries

$$c_{ij} = \frac{\alpha_i \beta_j}{\xi_i - \eta_j},$$

where $\alpha = [\alpha_1, \ldots, \alpha_n]$, $\beta = [\beta_1, \ldots, \beta_n]$, $\xi = [\xi_1, \ldots, \xi_n]$, and $\eta = [\eta_1, \ldots, \eta_n]$ are given vectors. The best-known example is the notoriously ill-conditioned *Hilbert matrix* H, with $h_{ij} = 1/(i+j-1)$. These matrices arise in interpolating data by rational functions: Suppose that we want to find the coefficients x_j of the rational function with fixed poles η_j

$$f(z) = \sum_{j=1}^{n} \frac{x_j}{z - \eta_j}$$

such that $f(\xi_i) = y_i$ for $i = 1$ to n. Taken together these n equations $f(\xi_i) = y_i$ form an n-by-n linear system with a coefficient matrix that is Cauchy. The inverse of a Cauchy matrix turns out to be a Cauchy matrix, and there is a closed form expression for C^{-1}, based on its connection with interpolation:

$$(C^{-1})_{ij} = \beta_i^{-1} \alpha_j^{-1} (\xi_j - \eta_i) P_j(\eta_i) Q_i(-\xi_j),$$

where $P_j(\cdot)$ and $Q_i(\cdot)$ are the Lagrange interpolation polynomials

$$P_j(z) = \prod_{k \neq j} \frac{\xi_k - z}{\xi_k - \xi_j} \quad \text{and} \quad Q_i(z) = \prod_{k \neq i} \frac{-\eta_k - z}{-\eta_k + \eta_i}.$$

Toeplitz matrices look like

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_n \\ a_{-1} & \ddots & \ddots & \ddots & \vdots \\ a_{-2} & \ddots & \ddots & \ddots & a_2 \\ \vdots & \ddots & \ddots & \ddots & a_1 \\ a_{-n} & \cdots & a_{-2} & a_{-1} & a_0 \end{bmatrix};$$

i.e., they are constant along diagonals. They arise in problems of signal processing. There are algorithms for solving such systems that take only $O(n^2)$ operations.

All these methods generalize to many other similar matrices depending on only $O(n)$ parameters. See [121, p. 183] or [160] for a recent survey.

2.8. References and Other Topics for Chapter 2

Further details about linear equation solving in general may be found in chapters 3 and 4 of [121]. The reciprocal relationship between condition numbers and distance to the nearest ill-posed problem is further explored in [71]. An average case analysis of pivot growth is described in [242], and an example of bad pivot growth with complete pivoting is given in [122]. Condition estimators are described in [138, 146, 148]. Single precision iterative refinement is analyzed in [14, 225, 226]. A comprehensive discussion of error analysis for linear equation solvers, which covers most of these topics, can be found in [149].

For symmetric indefinite factorization, see [44]. Sparse matrix algorithms are described in [114, 93] as well as the numerous references in Table 2.2. Implementations of many of the algorithms for dense and band matrices described in this chapter are available in LAPACK and CLAPACK [10], which includes a discussion of block algorithms suitable for high-performance computers. Parallel implementations are available in ScaLAPACK [34]. The BLAS are described in [87, 89, 169]. These and other routines are available electronically in NETLIB. An analysis of blocking strategies for matrix multiplication is given in [151]. Strassen's matrix multiplication algorithm is presented in [3], its performance in practice is described in [22], and its numerical stability is described in [77, 149]. A survey of parallel and other block algorithms is given in [76]. For a recent survey of algorithms for structured dense matrices depending only on $O(n)$ parameters, see [160]. For more material on sparse direct methods, see [93, 94, 114, 177].

2.9. Questions for Chapter 2

QUESTION 2.1. *(Easy)* Using your favorite World Wide Web browser, go to NETLIB (http://www.netlib.org), and answer the following questions.

1. You need a Fortran subroutine to compute the eigenvalues and eigenvectors of real symmetric matrices in double precision. Find one using the search facility in the NETLIB repository. Report the name and URL of the subroutine as well as how you found it.

2. Using the Performance Database Server, find out the current world speed record for solving 100-by-100 dense linear systems using Gaussian elimi-

nation. What is the speed in Mflops, and which machine attained it? Do the same for 1000-by-1000 dense linear systems and "big as you want" dense linear systems. Using the same database, find out how fast your workstation can solve 100-by-100 dense linear systems. Hint: Look at the LINPACK benchmark.

QUESTION 2.2. *(Easy)* Consider solving $AX = B$ for X, where A is n-by-n, and X and B are n-by-m. There are two obvious algorithms. The first algorithm factorizes $A = PLU$ using Gaussian elimination and then solves for each column of X by forward and back substitution. The second algorithm computes A^{-1} using Gaussian elimination and then multiplies $X = A^{-1}B$. Count the number of flops required by each algorithm, and show that the first one requires fewer flops.

QUESTION 2.3. *(Medium)* Let $\|\cdot\|$ be the two-norm. Given a nonsingular matrix A and a vector b, show that for sufficiently small $\|\delta A\|$, there are nonzero δA and δb such that inequality (2.2) is an equality. This justifies calling $\kappa(A) = \|A^{-1}\| \cdot \|A\|$ the condition number of A. Hint: Use the ideas in the proof of Theorem 2.1.

QUESTION 2.4. *(Hard)* Show that bounds (2.7) and (2.8) are attainable.

QUESTION 2.5. *(Medium)* Prove Theorem 2.3. Given the residual $r = A\hat{x} - b$, use Theorem 2.3 to show that bound (2.9) is no larger than bound (2.7). This explains why LAPACK computes a bound based on (2.9), as described in section 2.4.4.

QUESTION 2.6. *(Easy)* Prove Lemma 2.2.

QUESTION 2.7. *(Easy; Z. Bai)* If A is a nonsingular symmetric matrix and has the factorization $A = LDM^T$, where L and M are unit lower triangular matrices and D is a diagonal matrix, show that $L = M$.

QUESTION 2.8. *(Hard)* Consider the following two ways of solving a 2-by-2 linear system of equations:

$$Ax = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = b.$$

Algorithm 1. Gaussian elimination with partial pivoting (GEPP).

Algorithm 2. Cramer's rule:

$$\begin{aligned} \det &= a_{11} * a_{22} - a_{12} * a_{21}, \\ x_1 &= (a_{22} * b_1 - a_{12} * b_2)/\det, \\ x_2 &= (-a_{21} * b_1 + a_{11} * b_2)/\det. \end{aligned}$$

Show by means of a numerical example that Cramer's rule is not backward stable. Hint: Choose the matrix nearly singular and $[b_1\ b_2]^T \approx [a_{12}\ a_{22}]^T$. What does backward stability imply about the size of the residual? Your numerical example can be done by hand on paper (for example, with four-decimal-digit floating point), on a computer, or a hand calculator.

QUESTION 2.9. *(Medium)* Let B be an n-by-n upper bidiagonal matrix, i.e., nonzero only on the main diagonal and first superdiagonal. Derive an algorithm for computing $\kappa_\infty(B) \equiv \|B\|_\infty \|B^{-1}\|_\infty$ *exactly* (ignoring roundoff). In other words, you should not use an iterative algorithm such as Hager's estimator. Your algorithm should be as cheap as possible; it should be possible to do using no more than $2n-2$ additions, n multiplications, n divisions, $4n-2$ absolute values, and $2n-2$ comparisons. (Anything close to this is acceptable.)

QUESTION 2.10. *(Easy; Z. Bai)* Let A be n-by-m with $n \geq m$. Show that $\|A^TA\|_2 = \|A\|_2^2$ and $\kappa_2(A^TA) = \kappa_2(A)^2$.

Let M be n-by-n and positive definite and L be its Cholesky factor so that $M = LL^T$. Show that $\|M\|_2 = \|L\|_2^2$ and $\kappa_2(M) = \kappa_2(L)^2$.

QUESTION 2.11. *(Easy; Z. Bai)* Let A be symmetric and positive definite. Show that $|a_{ij}| < (a_{ii}a_{jj})^{1/2}$.

QUESTION 2.12. *(Easy; Z. Bai)* Show that if

$$Y = \begin{pmatrix} I & Z \\ 0 & I \end{pmatrix},$$

where I is an n-by-n identity matrix, then $\kappa_F(Y) = \|Y\|_F\|Y^{-1}\|_F = 2n + \|Z\|_F^2$.

QUESTION 2.13. *(Medium)* In this question we will ask how to solve $By = c$ given a fast way to solve $Ax = b$, where $A - B$ is "small" in some sense.

1. Prove the *Sherman–Morrison formula*: Let A be nonsingular, u and v be column vectors, and $A + uv^T$ be nonsingular. Then $(A + uv^T)^{-1} = A^{-1} - (A^{-1}uv^TA^{-1})/(1 + v^TA^{-1}u)$.

   More generally, prove the *Sherman–Morrison–Woodbury formula*: Let U and V be n-by-k rectangular matrices, where $k \leq n$ and A is n-by-n. Then $T = I + V^TA^{-1}U$ is nonsingular if and only if $A + UV^T$ is nonsingular, in which case $(A + UV^T)^{-1} = A^{-1} - A^{-1}UT^{-1}V^TA^{-1}$.

2. If you have a fast algorithm to solve $Ax = b$, show how to build a fast solver for $By = c$, where $B = A + uv^T$.

3. Suppose that $\|A-B\|$ is "small" and you have a fast algorithm for solving $Ax = b$. Describe an iterative scheme for solving $By = c$. How fast do you expect your algorithm to converge? Hint: Use iterative refinement.

QUESTION 2.14. *(Medium; Programming)* Use Netlib to obtain a subroutine to solve $Ax = b$ using Gaussian elimination with partial pivoting. You should get it from either LAPACK (in Fortran, NETLIB/lapack) or CLAPACK (in C, NETLIB/clapack); `sgesvx` is the main routine in both cases. (There is also a simpler routine `sgesv` that you might want to look at.) Modify `sgesvx` (and possibly other subroutines that it calls) to perform complete pivoting instead of partial pivoting; call this new routine `gecp`. It is probably simplest to modify `sgetf2` and use it in place of `sgetrf`. See HOMEPAGE/Matlab/gecp.m for a Matlab implementation. Test `sgesvx` and `gecp` on a number of randomly generated matrices of various sizes up to 30 or so. By choosing x and forming $b = Ax$, you can use examples for which you know the right answer. Check the accuracy of the computed answer $\hat{x}$ as follows. First, examine the error bounds `FERR` ("Forward ERRor") and `BERR` ("Backward ERRor") returned by the software; in your own words, say what these bounds mean. Using your knowledge of the exact answer, verify that `FERR` is correct. Second, compute the exact condition number by inverting the matrix explicitly, and compare this to the estimate `RCOND` returned by the software. (Actually, `RCOND` is an estimate of the reciprocal of the condition number.) Third, confirm that $\frac{\|\hat{x}-x\|}{\|\hat{x}\|}$ is bounded by a modest multiple of *macheps*/`RCOND`. Fourth, you should verify that the (scaled) backward error $R \equiv \|A\hat{x} - b\|/((\|A\| \cdot \|x\| + \|b\|) \cdot macheps)$ is of order unity in each case.

More specifically, your solution should consist of a well-documented program listing of `gecp`, an explanation of which random matrices you generated (see below), and a table with the following columns (or preferably graphs of each column of data, plotted against the first column):

- test matrix number (to identify it in your explanation of how it was generated);
- its dimension;
- from `sgesvx`:
  - —the pivot growth factor returned by the code (this should ideally not be much larger than 1),
  - —its estimated condition number (1/`RCOND`),
  - —the ratio of 1/`RCOND` to your explicitly computed condition number (this should ideally be close to 1),
  - —the error bound `FERR`,
  - —the ratio of `FERR` to the true error (this should ideally be at least 1 but not much larger unless you are "lucky" and the true error is zero),
  - —the ratio of the true error to ε/`RCOND` (this should ideally be at most 1 or a little less, unless you are "lucky" and the true error is zero),
  - —the scaled backward error R/ε (this should ideally be O(1) or perhaps O(n)),

—the backward error BERR/ε
(this should ideally be O(1) or perhaps O(n)),
—the run time in seconds;

- the same data for `gecp` as for `sgesvx`.

You need to print the data to only one decimal place, since we care only about approximate magnitudes. Do the error bounds really bound the errors? How do the speeds of `sgesvx` and `gecp` compare?

It is difficult to obtain accurate timings on many systems, since many timers have low resolution, so you should compute the run time as follows:

t_1 = time-so-far
for $i = 1$ to m
    set up problem
    solve the problem
endfor
t_2 = time-so-far
for $i = 1$ to m
    set up problem
endfor
t_3 = time-so-far
$t = ((t_2 - t_1) - (t_3 - t_2))/m$

m should be chosen large enough so that $t_2 - t_1$ is at least a few seconds. Then t should be a reliable estimate of the time to solve the problem.

You should test some well-conditioned problems as well as some that are ill-conditioned. To generate a well-conditioned matrix, let P be a permutation matrix, and add a small random number to each entry. To generate an ill-conditioned matrix, let L be a random lower triangular matrix with tiny diagonal entries and moderate subdiagonal entries. Let U be a similar upper triangular matrix, and let $A = LU$. (There is also an LAPACK subroutine `slatms` for generating random matrices with a given condition number, which you may use if you like.)

Also try both solvers on the following class of n-by-n matrices for $n = 1$ up to 30. (If you run in double precision, you may need to run up to $n = 60$.) Shown here is just the case $n = 5$; the others are similar:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}.$$

Explain the accuracy of the results in terms of the error analysis in section 2.4.

Your solution should *not* contain any tables of matrix entries or solution components.

In addition to teaching about error bounds, one purpose of this question is to show you what well-engineered numerical software looks like. In practice, one will often use or modify existing software instead of writing one's own from scratch.

QUESTION 2.15. *(Medium; Programming)* This problem depends on Question 2.14. Write another version of `sgesvx` called `sgesvxdouble` that computes the residual in double precision during iterative refinement. Modify the error bound `FERR` in `sgesvx` to reflect this improved accuracy. Explain your modification. (This may require you to explain how `sgesvx` computes its error bound in the first place.) On the same set of examples as in the last question, produce a similar table of data. When is `sgesvxdouble` more accurate than `sgesvx`?

QUESTION 2.16. *(Hard)* Show how to reorganize the Cholesky algorithm (Algorithm 2.11) to do most of its operations using Level 3 BLAS. Mimic Algorithm 2.10.

QUESTION 2.17. *(Easy)* Suppose that, in Matlab, you have an n-by-n matrix A and an n-by-1 matrix b. What do $A\backslash b$, b'/A, and A/b mean in Matlab? How does $A\backslash b$ differ from $\mathrm{inv}(A) * b$?

QUESTION 2.18. *(Medium)* Let

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where A_{11} is k-by-k and nonsingular. Then $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ is called the *Schur complement of* A_{11} *in* A, or just Schur complement for short.

1. Show that after k steps of Gaussian elimination without pivoting, A_{22} has been overwritten by S.
2. Suppose $A = A^T$, A_{11} is positive definite, and A_{22} is negative definite ($-A_{22}$ is positive definite). Show that A is nonsingular, that Gaussian elimination without pivoting will work in exact arithmetic, but (by means of a 2-by-2 example) that Gaussian elimination without pivoting may be numerically unstable.

QUESTION 2.19. *(Medium)* Matrix A is called *strictly column diagonally dominant*, or diagonally dominant for short, if

$$|a_{ii}| > \sum_{j=1,\, j\neq i}^{n} |a_{ji}|.$$

- Show that A is nonsingular. Hint: Use Gershgorin's theorem.

- Show that Gaussian elimination with partial pivoting does not actually permute any rows, i.e., that it is identical to Gaussian elimination without pivoting. Hint: Show that after one step of Gaussian elimination, the trailing $(n-1)$-by-$(n-1)$ submatrix, the *Schur complement of* a_{11} *in* A, is still diagonally dominant. (See Question 2.18 for more discussion of the Schur complement.)

QUESTION 2.20. *(Easy; Z. Bai)* Given an n-by-n nonsingular matrix A, how do you efficiently solve the following problems, using Gaussian elimination with partial pivoting?

(a) Solve the linear system $A^k x = b$, where k is a positive integer.

(b) Compute $\alpha = c^T A^{-1} b$.

(c) Solve the matrix equation $AX = B$, where B is n-by-m.

You should (1) describe your algorithms, (2) present them in pseudocode (using a Matlab-like language; you should not write down the algorithm for GEPP), and (3) give the required flops.

QUESTION 2.21. *(Medium)* Prove that Strassen's algorithm (Algorithm 2.8) correctly multiplies n-by-n matrices, where n is a power of 2.

3

Linear Least Squares Problems

3.1. Introduction

Given an m-by-n matrix A and an m-by-1 vector b, the *linear least squares problem* is to find an n-by-1 vector x minimizing $\|Ax - b\|_2$. If $m = n$ and A is nonsingular, the answer is simply $x = A^{-1}b$. But if $m > n$ so that we have more equations than unknowns, the problem is called *overdetermined*, and generally no x satisfies $Ax = b$ exactly. One occasionally encounters the *underdetermined* problem, where $m < n$, but we will concentrate on the more common overdetermined case.

This chapter is organized as follows. The rest of this introduction describes three applications of least squares problems, to *curve fitting*, to *statistical modeling* of noisy data, and to *geodetic modeling*. Section 3.2 discusses three standard ways to solve the least squares problem: the *normal equations*, the *QR decomposition*, and the *singular value decomposition (SVD)*. We will frequently use the SVD as a tool in later chapters, so we derive several of its properties (although algorithms for the SVD are left to Chapter 5). Section 3.3 discusses perturbation theory for least squares problems, and section 3.4 discusses the implementation details and roundoff error analysis of our main method, QR decomposition. The roundoff analysis applies to many algorithms using orthogonal matrices, including many algorithms for eigenvalues and the SVD in Chapters 4 and 5. Section 3.5 discusses the particularly ill-conditioned situation of rank-deficient least squares problem and how to solve them accurately. Section 3.7 and the questions at the end of the chapter give pointers to other kinds of least squares problems and to software for sparse problems.

EXAMPLE 3.1. A typical application of least squares is *curve fitting*. Suppose that we have m pairs of numbers $(y_1, b_1), \ldots, (y_m, b_m)$ and that we want to find the "best" cubic polynomial fit to b_i as a function of y_i. This means finding polynomial coefficients $x_1, \ldots, x_4$ so that the polynomial $p(y) = \sum_{j=1}^{4} x_j y^{j-1}$ minimizes the residual $r_i \equiv p(y_i) - b_i$ for $i = 1$ to m. We can also write this as

minimizing

$$
\begin{aligned}
r &\equiv \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{bmatrix} = \begin{bmatrix} p(y_1) \\ p(y_2) \\ \vdots \\ p(y_m) \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \\
&= \begin{bmatrix} 1 & y_1 & y_1^2 & y_1^3 \\ 1 & y_2 & y_2^2 & y_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & y_m & y_m^2 & y_m^3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \\
&\equiv A \cdot x - b,
\end{aligned}
$$

where r and b are m-by-1, A is m-by-4, and x is 4-by-1. To minimize r, we could choose any norm, such as $\|r\|_\infty$, $\|r\|_1$, or $\|r\|_2$. The last one, which corresponds to minimizing the sum of the squared residuals $\sum_{i=1}^m r_i^2$, is a *linear least squares problem.*

Figure 3.1 shows an example, where we fit polynomials of increasing degree to the smooth function $b = \sin(\pi y/5) + y/5$ at the 23 points $y = -5, -4.5, -4, \ldots, 5.5, 6$. The left side of Figure 3.1 plots the data points as circles, and four different approximating polynomials of degrees 1, 3, 6, and 19. The right side of Figure 3.1 plots the residual norm $\|r\|_2$ versus degree for degrees from 1 to 20. Note that as the degree increases from 1 to 17, the residual norm decreases. We expect this behavior, since increasing the polynomial degree should let us fit the data better.

But when we reach degree 18, the residual norm suddenly increases dramatically. We can see how erratic the plot of the degree 19 polynomial is on the left (the blue line). This is due to ill-conditioning, as we will later see. Typically, one does polynomial fitting only with relatively low degree polynomials, avoiding ill-conditioning [61]. Polynomial fitting is available as the function `polyfit` in Matlab.

Here is an alternative to polynomial fitting. More generally, one has a set of independent functions $f_1(y), \ldots, f_n(y)$ from $\mathbb{R}^k$ to $\mathbb{R}$ and a set of points $(y_1, b_1), \ldots, (y_m, b_m)$ with $y_i \in \mathbb{R}^k$ and $b_i \in \mathbb{R}$, and one wishes to find a best fit to these points of the form $b = \sum_{j=1}^n x_j f_j(y)$. In other words one wants to choose $x = [x_1, \ldots, x_n]^T$ to minimize the residuals $r_i \equiv \sum_{j=1}^n x_j f_j(y_i) - b_i$ for $1 \le i \le m$. Letting $a_{ij} = f_j(y_i)$, we can write this as $r = Ax - b$, where A is m-by-n, x is n-by-1, and b and r are m-by-1. A good choice of basis functions $f_i(y)$ can lead to better fits and less ill-conditioned systems than using polynomials [33, 84, 168]. $\diamond$

EXAMPLE 3.2. In statistical modeling, one often wishes to estimate certain parameters x_j based on some observations, where the observations are contaminated by noise. For example, suppose that one wishes to predict the college grade point average (GPA) (b) of freshman applicants based on their

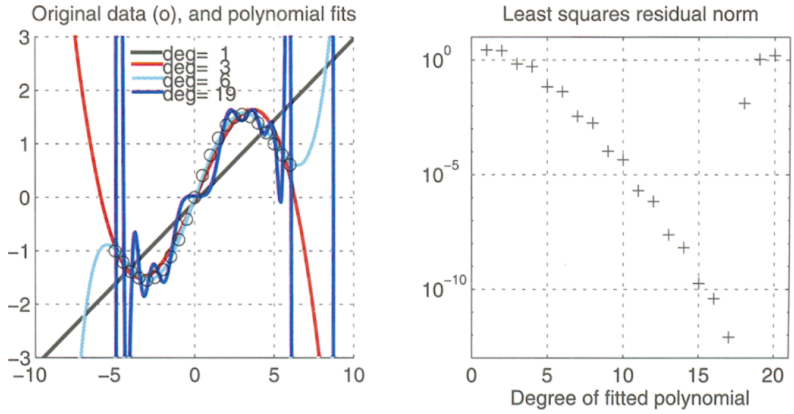


Fig. 3.1. *Polynomial fit to curve* $b = \sin(\pi y/5) + y/5$ *and residual norms.*

high school GPA (a_1) and two Scholastic Aptitude Test scores, verbal (a_2) and quantitative (a_3), as part of the college admissions process. Based on past data from admitted freshmen one can construct a *linear model* of the form $b = \sum_{j=1}^{3} a_j x_j$. The observations are a_{i1}, a_{i2}, a_{i3}, and b_i, one set for each of the m students in the database. Thus, one wants to minimize

$$r \equiv \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \equiv A \cdot x - b,$$

which we can do as a least squares problem.

Here is a statistical justification for least squares, which is called *linear regression* by statisticians: assume that the a_i are known exactly so that only b has noise in it, and that the noise in each b_i is independent and normally distributed with 0 mean and the same standard deviation σ. Let x be the solution of the least squares problem and x_T be the true value of the parameters. Then x is called a *maximum-likelihood estimate* of x_T, and the error $x - x_T$ is normally distributed, with zero mean in each component and *covariance matrix* $\sigma^2(A^T A)^{-1}$. We will see the matrix $(A^T A)^{-1}$ again below when we solve the least squares problem using the normal equations. For more details on the connection to statistics,[15] see, for example, [33, 259]. ◇

EXAMPLE 3.3. The least squares problem was first posed and formulated by Gauss to solve a practical problem for the German government. There are important economic and legal reasons to know exactly where the boundaries lie between plots of land owned by different people. Surveyors would go out and try to establish these boundaries, measuring certain angles and distances

[15] The standard notation in statistics differs from linear algebra: statisticians write $X\beta = y$ instead of $Ax = b$.

and then triangulating from known landmarks. As time passed, it became necessary to improve the accuracy to which the locations of the landmarks were known. So the surveyors of the day went out and remeasured many angles and distances between landmarks, and it fell to Gauss to figure out how to take these more accurate measurements and update the government database of locations. For this he invented least squares, as we will explain shortly [33].

The problem that Gauss solved did not go away and must be periodically revisited. In 1974 the US National Geodetic Survey undertook to update the US geodetic database, which consisted of about 700,000 points. The motivations had grown to include supplying accurate enough data for civil engineers and regional planners to plan construction projects and for geophysicists to study the motion of tectonic plates in the earth's crust (which can move up to 5 cm per year). The corresponding least squares problem was the largest ever solved at the time: about 2.5 million equations in 400,000 unknowns. It was also very sparse, which made it tractable on the computers available in 1978, when the computation was done [164].

Now we briefly discuss the formulation of this problem. It is actually nonlinear and is solved by approximating it by a sequence of linear ones, each of which is a linear least squares problem. The data base consists of a list of points (landmarks), each labeled by location: latitude, longitude, and possibly elevation. For simplicity of exposition, we assume that the earth is flat and suppose that each point i is labeled by linear coordinates $z_i = (x_i, y_i)^T$. For each point we wish to compute a correction $\delta z_i = (\delta x_i, \delta y_i)^T$ so that the corrected location $z_i' = (x_i', y_i')^T = z_i + \delta z_i$ more nearly matches the new, more accurate measurements. These measurements include both distances between selected pairs of points and angles between the line segment from point i to j and i to k (see Figure 3.2). To see how to turn these new measurements into constraints, consider the triangle in Figure 3.2. The corners are labeled by their (corrected) locations, and the angles θ and edge lengths L are also shown. From this data, it is easy to write down constraints based on simple trigonometric identities. For example, an accurate measurement of θ_i leads to the constraint

$$\cos^2\theta_i = \frac{[(z_j' - z_i')^T(z_k' - z_i')]^2}{(z_j' - z_i')^T(z_j' - z_i') \cdot (z_k' - z_i')^T(z_k' - z_i')},$$

where we have expressed $\cos\theta_i$ in terms of dot products of certain sides of the triangle. If we assume that δz_i is small compared to z_i, then we can linearize this constraint as follows: multiply through by the denominator of the fraction, multiply out all the terms to get a quartic polynomial in all the "δ-variables" (like δx_i), and throw away all terms containing more than one δ-variable as a factor. This yields an equation in which all δ-variables appear linearly. If we collect all these linear constraints from all the new angle and distance measurements together, we get an overdetermined linear system of

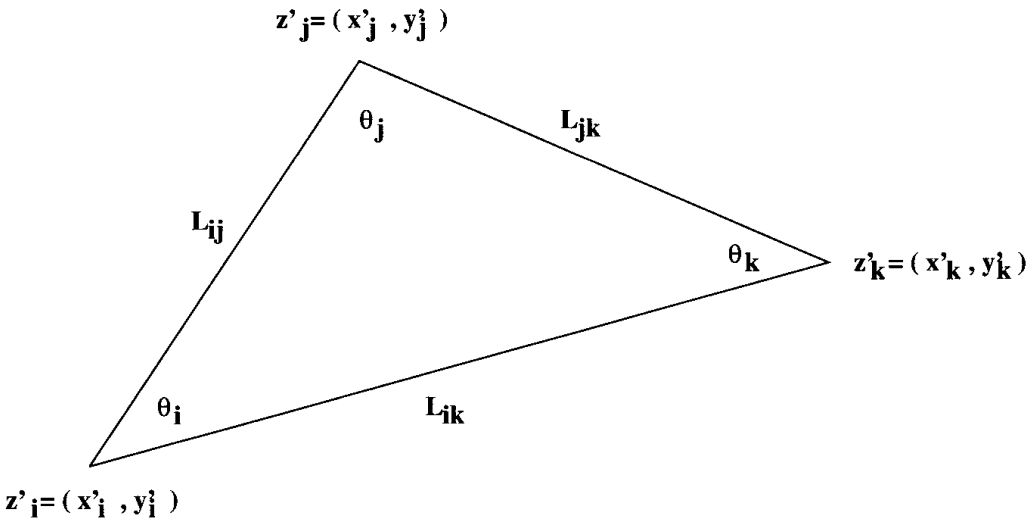


Fig. 3.2. *Constraints in updating a geodetic database.*

equations for all the δ-variables. We wish to find the smallest corrections, i.e., the smallest values of δx_i, etc., that most nearly satisfy these constraints. This is a least squares problem. $\diamond$

Later, after we introduce more machinery, we will also show how *image compression* can be interpreted as a least squares problem (see Example 3.4).

3.2. Matrix Factorizations That Solve the Linear Least Squares Problem

The linear least squares problem has several explicit solutions that we now discuss:

1. normal equations,
2. QR decomposition,
3. SVD,
4. transformation to a linear system (see Question 3.3).

The first method is the fastest but least accurate; it is adequate when the condition number is small. The second method is the standard one and costs up to twice as much as the first method. The third method is of most use on an ill-conditioned problem, i.e., when A is not of full rank; it is several times more expensive again. The last method lets us do iterative refinement to improve the solution when the problem is ill-conditioned. All methods but the third can be adapted to deal efficiently with sparse matrices [33]. We will discuss each solution in turn. We assume initially for methods 1 and 2 that A has full column rank n.

3.2.1. Normal Equations

To derive the *normal equations*, we look for the x where the gradient of $\|Ax - b\|_2^2 = (Ax - b)^T(Ax - b)$ vanishes. So we want

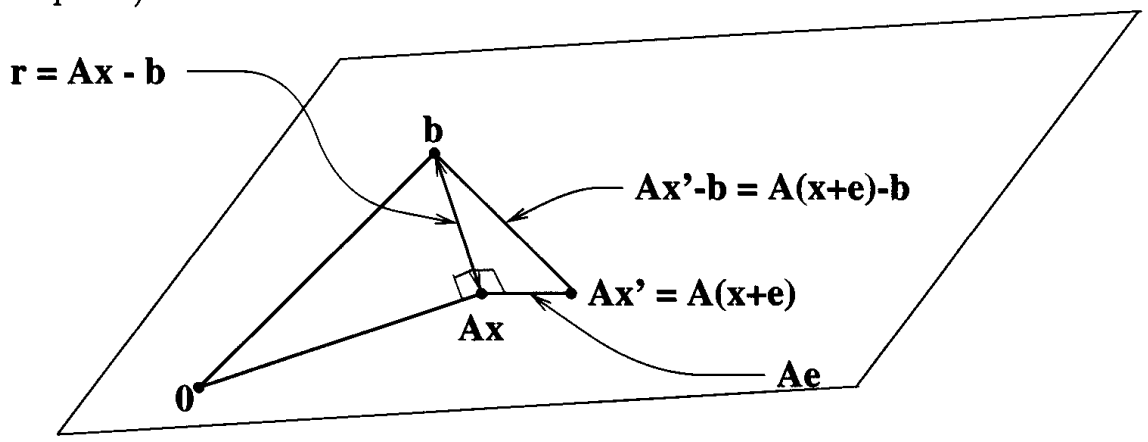
$$\begin{aligned} 0 &= \lim_{e \to 0} \frac{(A(x+e) - b)^T(A(x+e) - b) - (Ax - b)^T(Ax - b)}{\|e\|_2} \\ &= \lim_{e \to 0} \frac{2e^T(A^TAx - A^Tb) + e^TA^TAe}{\|e\|_2}. \end{aligned}$$

The second term $\frac{|e^TA^TAe|}{\|e\|_2} \leq \frac{\|A\|_2^2\|e\|_2^2}{\|e\|_2} = \|A\|_2^2\|e\|_2$ approaches 0 as e goes to 0, so the factor $A^TAx - A^Tb$ in the first term must also be zero, or $A^TAx = A^Tb$. This is a system of n linear equations in n unknowns, the normal equations.

Why is $x = (A^TA)^{-1}A^Tb$ the minimizer of $\|Ax - b\|_2^2$? We can note that the Hessian A^TA is positive definite, which means that the function is strictly convex and any critical point is a global minimum. Or we can complete the square by writing $x' = x + e$ and simplifying

$$\begin{aligned} (Ax' - b)^T(Ax' - b) &= (Ae + Ax - b)^T(Ae + Ax - b) \\ &= (Ae)^T(Ae) + (Ax - b)^T(Ax - b) \\ &\quad +2(Ae)^T(Ax - b) \\ &= \|Ae\|_2^2 + \|Ax - b\|_2^2 + 2e^T(A^TAx - A^Tb) \\ &= \|Ae\|_2^2 + \|Ax - b\|_2^2. \end{aligned}$$

This is clearly minimized by $e = 0$. This is just the Pythagorean theorem, since the residual $r = Ax - b$ is orthogonal to the space spanned by the columns of A, i.e., $0 = A^Tr = A^TAx - A^Tb$ as illustrated below (the plane shown is the span of the column vectors of A so that Ax, Ae, and $Ax' = A(x + e)$ all lie in the plane):



Since A^TA is symmetric and positive definite, we can use the Cholesky decomposition to solve the normal equations. The total cost of computing A^TA, A^Tb, and the Cholesky decomposition is $n^2m + \frac{1}{3}n^3 + O(n^2)$ flops. Since $m \geq n$, the n^2m cost of forming A^TA dominates the cost.

3.2.2. QR Decomposition

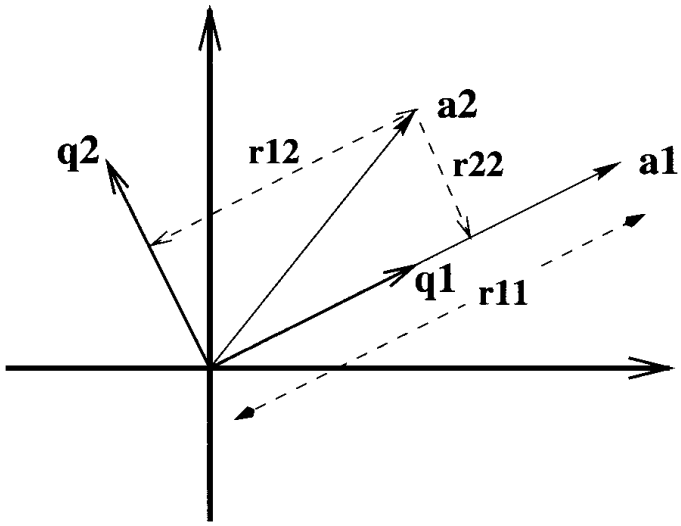
THEOREM 3.1. QR decomposition. *Let A be m-by-n with $m \geq n$. Suppose that A has full column rank. Then there exist a unique m-by-n orthogonal matrix Q ($Q^TQ = I_n$) and a unique n-by-n upper triangular matrix R with positive diagonals $r_{ii} > 0$ such that $A = QR$.*

Proof. We give two proofs of this theorem. First, this theorem is a restatement of the Gram–Schmidt orthogonalization process [139]. If we apply Gram–Schmidt to the columns a_i of $A = [a_1, a_2, \ldots, a_n]$ from left to right, we get a sequence of orthonormal vectors q_1 through q_n spanning the same space: these orthogonal vectors are the columns of Q. Gram–Schmidt also computes coefficients $r_{ji} = q_j^T a_i$ expressing each column a_i as a linear combination of q_1 through q_i: $a_i = \sum_{j=1}^{i} r_{ji} q_j$. The r_{ji} are just the entries of R.

ALGORITHM 3.1. *The classical Gram–Schmidt (CGS) and modified Gram–Schmidt (MGS) Algorithms for factoring $A = QR$:*

for $i = 1$ *to* n */* compute ith columns of Q and R */*
  $q_i = a_i$
  for $j = 1$ *to* $i - 1$ */* subtract component in q_j direction from a_i */*
    $\begin{cases} r_{ji} = q_j^T a_i & \text{CGS} \\ r_{ji} = q_j^T q_i & \text{MGS} \end{cases}$
    $q_i = q_i - r_{ji} q_j$
  end for
  $r_{ii} = \|q_i\|_2$
  if $r_{ii} = 0$ */* a_i is linearly dependent on $a_1, \ldots, a_{i-1}$ */*
    quit
  end if
  $q_i = q_i / r_{ii}$
end for

We leave it as an exercise to show that the two formulas for r_{ji} in the algorithm are mathematically equivalent (see Question 3.1). If A has full column rank, r_{ii} will not be zero. The following figure illustrates Gram–Schmidt when A is 2-by-2:



The second proof of this theorem will use Algorithm 3.2, which we present in section 3.4.1. □

Unfortunately, CGS is numerically unstable in floating point arithmetic when the columns of A are nearly linearly dependent. MGS is more stable and will be used in algorithms later in this book but may still result in Q being far from orthogonal ($\|Q^TQ - I\|$ being far larger than ε) when A is ill-conditioned [31, 32, 33, 149]. Algorithm 3.2 in section 3.4.1 is a stable alternative algorithm for factoring $A = QR$. See Question 3.2.

We will derive the formula for the x that minimizes $\|Ax - b\|_2$ using the decomposition $A = QR$ in three slightly different ways. First, we can always choose $m-n$ more orthonormal vectors $\tilde{Q}$ so that $[Q, \tilde{Q}]$ is a square orthogonal matrix (for example, we can choose any $m - n$ more independent vectors $\tilde{X}$ that we want and then apply Algorithm 3.1 to the n-by-n nonsingular matrix $[Q, \tilde{X}]$). Then

$$\begin{aligned}
\|Ax - b\|_2^2 &= \|[Q, \tilde{Q}]^T(Ax - b)\|_2^2 \quad \text{by part 4 of Lemma 1.7} \\
&= \left\| \begin{bmatrix} Q^T \\ \tilde{Q}^T \end{bmatrix} (QRx - b) \right\|_2^2 \\
&= \left\| \begin{bmatrix} I^{n\times n} \\ O^{(m-n)\times n} \end{bmatrix} Rx - \begin{bmatrix} Q^Tb \\ \tilde{Q}^Tb \end{bmatrix} \right\|_2^2 \\
&= \left\| \begin{bmatrix} Rx - Q^Tb \\ -\tilde{Q}^Tb \end{bmatrix} \right\|_2^2 \\
&= \|Rx - Q^Tb\|_2^2 + \|\tilde{Q}^Tb\|_2^2 \\
&\geq \|\tilde{Q}^Tb\|_2^2.
\end{aligned}$$

We can solve $Rx - Q^Tb = 0$ for x, since A and R have the same rank, n, and so R is nonsingular. Then $x = R^{-1}Q^Tb$, and the minimum value of $\|Ax - b\|_2$ is $\|\tilde{Q}^Tb\|_2$.

Here is a second, slightly different derivation that does not use the matrix

$\tilde{Q}$. Rewrite $Ax - b$ as

$$\begin{aligned} Ax - b &= QRx - b = QRx - (QQ^T + I - QQ^T)b \\ &= Q(Rx - Q^Tb) - (I - QQ^T)b. \end{aligned}$$

Note that the vectors $Q(Rx - Q^Tb)$ and $(I - QQ^T)b$ are orthogonal, because $(Q(Rx - Q^Tb))^T((I - QQ^T)b) = (Rx - Q^Tb)^T[Q^T(I - QQ^T)]b = (Rx - Q^Tb)^T[0]b = 0$. Therefore, by the Pythagorean theorem,

$$\begin{aligned} \|Ax - b\|_2^2 &= \|Q(Rx - Q^Tb)\|_2^2 + \|(I - QQ^T)b\|_2^2 \\ &= \|Rx - Q^Tb\|_2^2 + \|(I - QQ^T)b\|_2^2, \end{aligned}$$

where we have used part 4 of Lemma 1.7 in the form $\|Qy\|_2^2 = \|y\|_2^2$. This sum of squares is minimized when the first term is zero, i.e., $x = R^{-1}Q^Tb$.

Finally, here is a third derivation that starts from the normal equations solution:

$$\begin{aligned} x &= (A^TA)^{-1}A^Tb \\ &= (R^TQ^TQR)^{-1}R^TQ^Tb = (R^TR)^{-1}R^TQ^Tb \\ &= R^{-1}R^{-T}R^TQ^Tb = R^{-1}Q^Tb. \end{aligned}$$

Later we will show that the cost of this decomposition and subsequent least squares solution is $2n^2m - \frac{2}{3}n^3$, about twice the cost of the normal equations if $m \gg n$ and about the same if $m = n$.

3.2.3. Singular Value Decomposition

The SVD is a very important decomposition which is used for many purposes other than solving least squares problems.

THEOREM 3.2. SVD. *Let A be an arbitrary m-by-n matrix with $m \geq n$. Then we can write $A = U\Sigma V^T$, where U is m-by-n and satisfies $U^TU = I$, V is n-by-n and satisfies $V^TV = I$, and $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$, where $\sigma_1 \geq \cdots \geq \sigma_n \geq 0$. The columns $u_1, \ldots, u_n$ of U are called* left singular vectors. *The columns $v_1, \ldots, v_n$ of V are called* right singular vectors. *The σ_i are called* singular values. *(If $m < n$, the SVD is defined by considering A^T.)*

A geometric restatement of this theorem is as follows. Given any m-by-n matrix A, think of it as mapping a vector $x \in \mathbb{R}^n$ to a vector $y = Ax \in \mathbb{R}^m$. Then we can choose one orthogonal coordinate system for $\mathbb{R}^n$ (where the unit axes are the columns of V) and another orthogonal coordinate system for $\mathbb{R}^m$ (where the units axes are the columns of U) such that A is diagonal (Σ), i.e., maps a vector $x = \sum_{i=1}^n \beta_i v_i$ to $y = Ax = \sum_{i=1}^n \sigma_i \beta_i u_i$. In other words, any matrix is diagonal, provided that we pick appropriate orthogonal coordinate systems for its domain and range.

Proof of Theorem 3.2. We use induction on m and n: we assume that the SVD exists for $(m-1)$-by-$(n-1)$ matrices and prove it for m-by-n. We assume $A \neq 0$; otherwise we can take $\Sigma = 0$ and let U and V be arbitrary orthogonal matrices.

The basic step occurs when $n = 1$ (since $m \geq n$). We write $A = U\Sigma V^T$ with $U = A/\|A\|_2$, $\Sigma = \|A\|_2$, and $V = 1$.

For the induction step, choose v so $\|v\|_2 = 1$ and $\|A\|_2 = \|Av\|_2 > 0$. Such a v exists by the definition of $\|A\|_2 = \max_{\|v\|_2=1} \|Av\|_2$. Let $u = \frac{Av}{\|Av\|_2}$, which is a unit vector. Choose $\tilde{U}$ and $\tilde{V}$ so that $U = [u, \tilde{U}]$ is an m-by-m orthogonal matrix, and $V = [v, \tilde{V}]$ is an n-by-n orthogonal matrix. Now write

$$U^T A V = \begin{bmatrix} u^T \\ \tilde{U}^T \end{bmatrix} \cdot A \cdot \begin{bmatrix} v & \tilde{V} \end{bmatrix} = \begin{bmatrix} u^T Av & u^T A\tilde{V} \\ \tilde{U}^T Av & \tilde{U}^T A\tilde{V} \end{bmatrix}.$$

Then

$$u^T Av = \frac{(Av)^T(Av)}{\|Av\|_2} = \frac{\|Av\|_2^2}{\|Av\|_2} = \|Av\|_2 = \|A\|_2 \equiv \sigma$$

and $\tilde{U}^T Av = \tilde{U}^T u \|Av\|_2 = 0$. We claim $u^T A\tilde{V} = 0$ too because otherwise $\sigma = \|A\|_2 = \|U^T AV\|_2 \geq \|[1, 0, \ldots, 0]U^T AV\|_2 = \|[\sigma | u^T A\tilde{V}]\|_2 > \sigma$, a contradiction. (We have used part 7 of Lemma 1.7.)

So $U^T AV = \begin{bmatrix} \sigma & 0 \\ 0 & \tilde{U}^T A\tilde{V} \end{bmatrix} = \begin{bmatrix} \sigma & 0 \\ 0 & \tilde{A} \end{bmatrix}$. We may now apply the induction hypothesis to $\tilde{A}$ to get $\tilde{A} = U_1 \Sigma_1 V_1^T$, where U_1 is $(m-1)$-by-$(n-1)$, Σ_1 is $(n-1)$-by-$(n-1)$, and V_1 is $(n-1)$-by-$(n-1)$. So

$$U^T AV = \begin{bmatrix} \sigma & 0 \\ 0 & U_1 \Sigma_1 V_1^T \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix} \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix}^T$$

or

$$A = \left(U \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix} \right) \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \left(V \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix} \right)^T,$$

which is our desired decomposition. □

The SVD has a large number of important algebraic and geometric properties, the most important of which we state here.

THEOREM 3.3. *Let $A = U\Sigma V^T$ be the SVD of the m-by-n matrix A, where $m \geq n$. (There are analogous results for $m < n$.)*

1. *Suppose that A is symmetric, with eigenvalues λ_i and orthonormal eigenvectors u_i. In other words $A = U\Lambda U^T$ is an eigendecomposition of A, with $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$, $U = [u_1, \ldots, u_n]$, and $UU^T = I$. Then an SVD of A is $A = U\Sigma V^T$, where $\sigma_i = |\lambda_i|$ and $v_i = \mathrm{sign}(\lambda_i) u_i$, where* $\mathrm{sign}(0) = 1$.

2. *The eigenvalues of the symmetric matrix* A^TA *are* σ_i^2. *The right singular vectors* v_i *are corresponding orthonormal eigenvectors.*

3. *The eigenvalues of the symmetric matrix* AA^T *are* σ_i^2 *and* $m-n$ *zeroes. The left singular vectors* u_i *are corresponding orthonormal eigenvectors for the eigenvalues* σ_i^2. *One can take any* $m-n$ *other orthogonal vectors as eigenvectors for the eigenvalue* 0.

4. *Let* $H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$, *where* A *is square and* $A = U\Sigma V^T$ *is the SVD of* A. *Let* $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$, $U = [u_1, \ldots, u_n]$, *and* $V = [v_1, \ldots, v_n]$. *Then the* $2n$ *eigenvalues of* H *are* $\pm\sigma_i$, *with corresponding unit eigenvectors* $\frac{1}{\sqrt{2}}\begin{bmatrix} v_i \\ \pm u_i \end{bmatrix}$.

5. *If* A *has full rank, the solution of* $\min_x \|Ax-b\|_2$ *is* $x = V\Sigma^{-1}U^Tb$.

6. $\|A\|_2 = \sigma_1$. *If* A *is square and nonsingular, then* $\|A^{-1}\|_2^{-1} = \sigma_n$ *and* $\|A\|_2 \cdot \|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n}$.

7. *Suppose* $\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0$. *Then the rank of* A *is* r. *The null space of* A, *i.e., the subspace of vectors* v *such that* $Av = 0$, *is the space spanned by columns* $r+1$ *through* n *of* V: $\mathrm{span}(v_{r+1}, \ldots, v_n)$. *The range space of* A, *the subspace of vectors of the form* Aw *for all* w, *is the space spanned by columns* 1 *through* r *of* U: $\mathrm{span}(u_1, \ldots, u_r)$.

8. *Let* S^{n-1} *be the unit sphere in* $\mathbb{R}^n$: $S^{n-1} = \{x \in \mathbb{R}^n : \|x\|_2 = 1\}$. *Let* $A \cdot S^{n-1}$ *be the image of* S^{n-1} *under* A: $A \cdot S^{n-1} = \{Ax : x \in \mathbb{R}^n \text{ and } \|x\|_2 = 1\}$. *Then* $A \cdot S^{n-1}$ *is an ellipsoid centered at the origin of* $\mathbb{R}^m$, *with principal axes* $\sigma_i u_i$.

9. *Write* $V = [v_1, v_2, \ldots, v_n]$ *and* $U = [u_1, u_2, \ldots, u_n]$, *so* $A = U\Sigma V^T = \sum_{i=1}^n \sigma_i u_i v_i^T$ *(a sum of rank-1 matrices). Then a matrix of rank* $k < n$ *closest to* A *(measured with* $\|\cdot\|_2$*) is* $A_k = \sum_{i=1}^k \sigma_i u_i v_i^T$, *and* $\|A - A_k\|_2 = \sigma_{k+1}$. *We may also write* $A_k = U\Sigma_k V^T$, *where* $\Sigma_k = \mathrm{diag}(\sigma_1, \ldots, \sigma_k, 0, \ldots, 0)$.

Proof.

1. This is true by the definition of the SVD.

2. $A^TA = V\Sigma U^TU\Sigma V^T = V\Sigma^2V^T$. This is an eigendecomposition of A^TA, with the columns of V the eigenvectors and the diagonal entries of Σ^2 the eigenvalues.

3. Choose an m-by-$(m-n)$ matrix $\tilde{U}$ so that $[U, \tilde{U}]$ is square and orthogonal. Then write

$$AA^T = U\Sigma V^TV\Sigma U^T = U\Sigma^2U^T = \left[U, \tilde{U}\right] \begin{bmatrix} \Sigma^2 & 0 \\ 0 & 0 \end{bmatrix} \left[U, \tilde{U}\right]^T.$$

This is an eigendecomposition of AA^T.

4. See Question 3.14.

5. $\|Ax - b\|_2^2 = \|U\Sigma V^T x - b\|_2^2$. Since A has full rank, so does Σ, and thus Σ is invertible. Now let $[U, \tilde{U}]$ be square and orthogonal as above so

$$\begin{aligned} \|U\Sigma V^T x - b\|_2^2 &= \left\| \begin{bmatrix} U^T \\ \tilde{U}^T \end{bmatrix} (U\Sigma V^T x - b) \right\|_2^2 \\ &= \left\| \begin{bmatrix} \Sigma V^T x - U^T b \\ -\tilde{U}^T b \end{bmatrix} \right\|_2^2 \\ &= \|\Sigma V^T x - U^T b\|_2^2 + \|\tilde{U}^T b\|_2^2. \end{aligned}$$

This is minimized by making the first term zero, i.e., $x = V\Sigma^{-1}U^T b$.

6. It is clear from its definition that the two-norm of a diagonal matrix is the largest absolute entry on its diagonal. Thus, by part 3 of Lemma 1.7, $\|A\|_2 = \|U^T AV\|_2 = \|\Sigma\|_2 = \sigma_1$ and $\|A^{-1}\|_2 = \|V^T A^{-1} U\|_2 = \|\Sigma^{-1}\|_2 = \sigma_n^{-1}$.

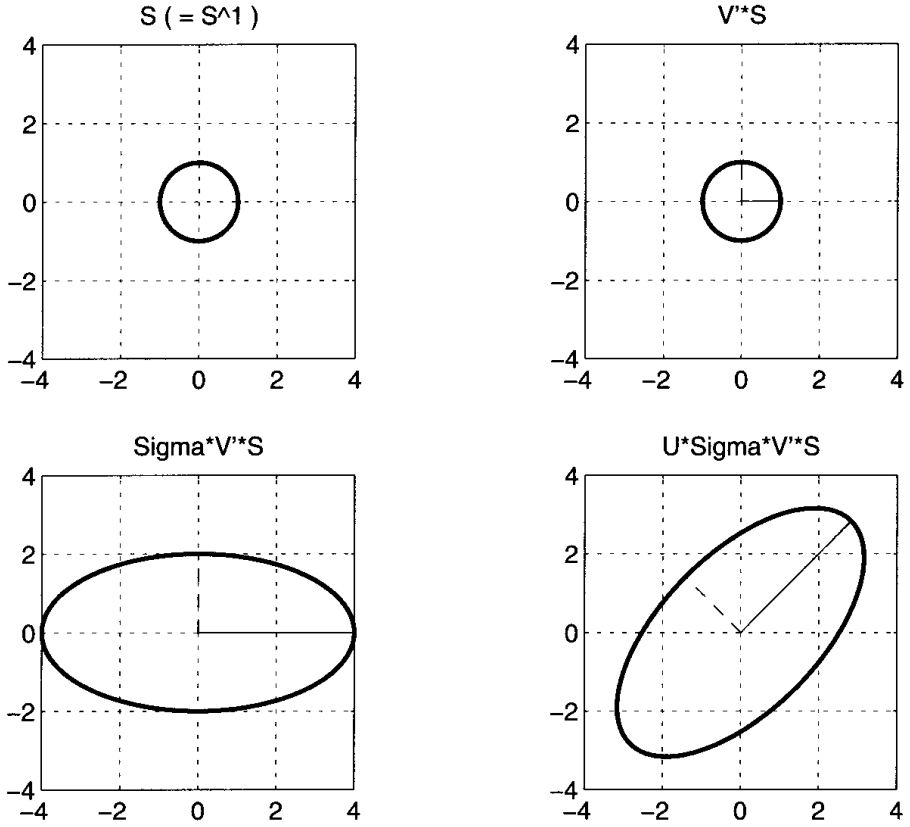
7. Again choose an m-by-$(m-n)$ matrix $\tilde{U}$ so that the m-by-m matrix $\hat{U} = [U, \tilde{U}]$ is orthogonal. Since $\hat{U}$ and V are nonsingular, A and $\hat{U}^T AV = \begin{bmatrix} \Sigma^{n\times n} \\ 0^{(m-n)\times n} \end{bmatrix} \equiv \hat{\Sigma}$ have the same rank—namely, r—by our assumption about Σ. Also, v is in the null space of A if and only if $V^T v$ is in the null space of $\hat{U}^T AV = \hat{\Sigma}$, since $Av = 0$ if and only if $\hat{U}^T AV(V^T v) = 0$. But the null space of $\hat{\Sigma}$ is clearly spanned by columns $r+1$ through n of the n-by-n identity matrix I_n, so the null space of A is spanned by V times these columns, i.e., v_{r+1} through v_n. A similar argument shows that the range space of A is the same as $\hat{U}$ times the range space of $\hat{U}^T AV = \hat{\Sigma}$, i.e., $\hat{U}$ times the first r columns of I_m, or u_1 through u_r.

8. We "build" the set $A \cdot S^{n-1}$ by multiplying by one factor of $A = U\Sigma V^T$ at a time. The figure below illustrates what happens when

$$\begin{aligned} A &= \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 2^{-1/2} & -2^{-1/2} \\ 2^{-1/2} & 2^{-1/2} \end{bmatrix} \cdot \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 2^{-1/2} & -2^{-1/2} \\ 2^{-1/2} & 2^{-1/2} \end{bmatrix}^T \\ &\equiv U\Sigma V^T. \end{aligned}$$

Assume for simplicity that A is square and nonsingular. Since V is orthogonal and so maps unit vectors to other unit vectors, $V^T \cdot S^{n-1} = S^{n-1}$. Next, since $v \in S^{n-1}$ if and only if $\|v\|_2 = 1$, $w \in \Sigma S^{n-1}$ if and only if $\|\Sigma^{-1} w\|_2 = 1$ or $\sum_{i=1}^n (w_i/\sigma_i)^2 = 1$. This defines an ellipsoid with

principal axes $\sigma_i e_i$, where e_i is the ith column of the identity matrix. Finally, multiplying each $w = \Sigma v$ by U just rotates the ellipse so that each e_i becomes u_i, the ith column of U.



9. A_k has rank k by construction and

$$\|A - A_k\|_2 = \left\| \sum_{i=k+1}^{n} \sigma_i u_i v_i^T \right\| = \left\| U \begin{bmatrix} 0 & & & \\ & \sigma_{k+1} & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix} V^T \right\|_2 = \sigma_{k+1}.$$

It remains to show that there is no closer rank k matrix to A. Let B be any rank k matrix, so its null space has dimension $n - k$. The space spanned by $\{v_1, \ldots, v_{k+1}\}$ has dimension $k + 1$. Since the sum of their dimensions is $(n - k) + (k + 1) > n$, these two spaces must overlap. Let h be a unit vector in their intersection. Then

$$\begin{aligned} \|A - B\|_2^2 &\geq \|(A - B)h\|_2^2 = \|Ah\|_2^2 = \|U\Sigma V^T h\|_2^2 \\ &= \|\Sigma(V^T h)\|_2^2 \\ &\geq \sigma_{k+1}^2 \|V^T h\|_2^2 \\ &= \sigma_{k+1}^2. \quad \square \end{aligned}$$

EXAMPLE 3.4. We illustrate the last part of Theorem 3.3 by using it for *image compression*. In particular, we will illustrate it with low-rank approximations

of a clown. An m-by-n image is just an m-by-n matrix, where entry (i,j) is interpreted as the brightness of pixel (i,j). In other words, matrix entries ranging from 0 to 1 (say) are interpreted as pixels ranging from black (=0) through various shades of gray to white (=1). (Colors also are possible.) Rather than storing or transmitting all $m \cdot n$ matrix entries to represent the image, we often prefer to *compress* the image by storing many fewer numbers, from which we can still approximately reconstruct the original image. We may use Part 9 of Theorem 3.3 to do this, as we now illustrate.

Consider the image in Figure 3.3(a). This 320-by-200 pixel image corresponds to a 320-by-200 matrix A. Let $A = U\Sigma V^T$ be the SVD of A. Part 9 of Theorem 3.3 tells us that $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ is the best rank-k approximation of A, in the sense of minimizing $\|A - A_k\|_2 = \sigma_{k+1}$. Note that it only takes $m \cdot k + n \cdot k = (m+n) \cdot k$ words to store u_1 through u_k and $\sigma_1 v_1$ through $\sigma_k v_k$, from which we can reconstruct A_k. In contrast, it takes $m \cdot n$ words to store A (or A_k explicitly), which is much larger when k is small. So we will use A_k as our compressed image, stored using $(m+n) \cdot k$ words. The other images in Figure 3.3 show these approximations for various values of k, along with the relative errors σ_{k+1}/σ_1 and compression ratios $(m+n) \cdot k/(m \cdot n) = 520 \cdot k/64000 \approx k/123$.

k	Relative error $= \sigma_{k+1}/\sigma_k$	Compression ratio $= 520k/64000$
3	.155	.024
10	.077	.081
20	.040	.163

These images were produced by the following commands (the clown and other images are available in Matlab among the visualization demonstration files; check your local installation for location):

```
load clown.mat; [U,S,V]=svd(X); colormap('gray');
image(U(:,1:k)*S(1:k,1:k)*V(:,1:k)')
```

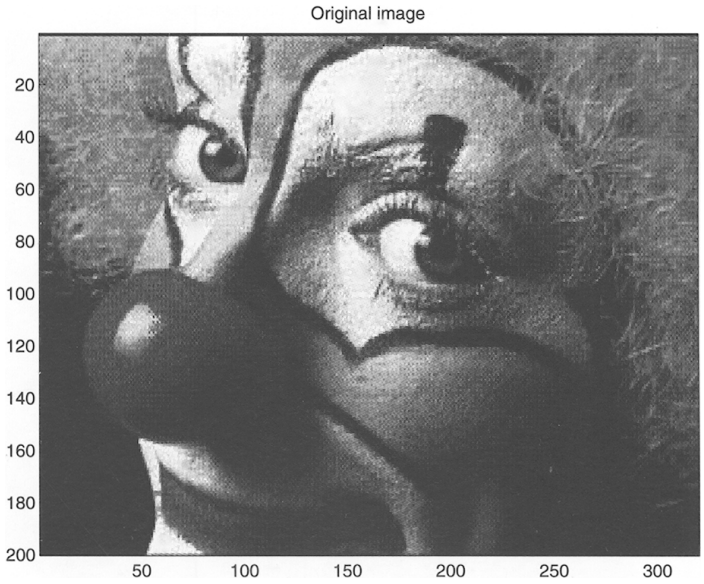
There are also many other, cheaper image-compression techniques available than the SVD [189, 152]. ◇

Later we will see that the cost of solving a least squares problem with the SVD is about the same as with QR when $m >> n$, and about $4n^2m - \frac{4}{3}n^3 + O(n^2)$ for smaller m. A precise comparison of the costs of QR and the SVD also depends on the machine being used. See section 3.6 for details.

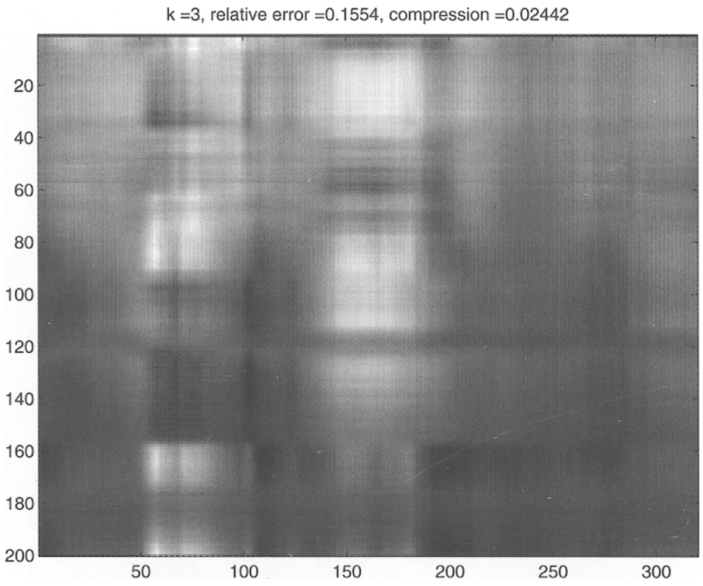
DEFINITION 3.1. *Suppose that A is m-by-n with $m \geq n$ and has full rank, with $A = QR = U\Sigma V^T$ being A's QR decomposition and SVD, respectively. Then*

$$A^+ \equiv (A^TA)^{-1}A^T = R^{-1}Q^T = V\Sigma^{-1}U^T$$

is called the (Moore–Penrose) pseudoinverse *of A. If $m < n$, then $A^+ \equiv A^T(AA^T)^{-1}$.*

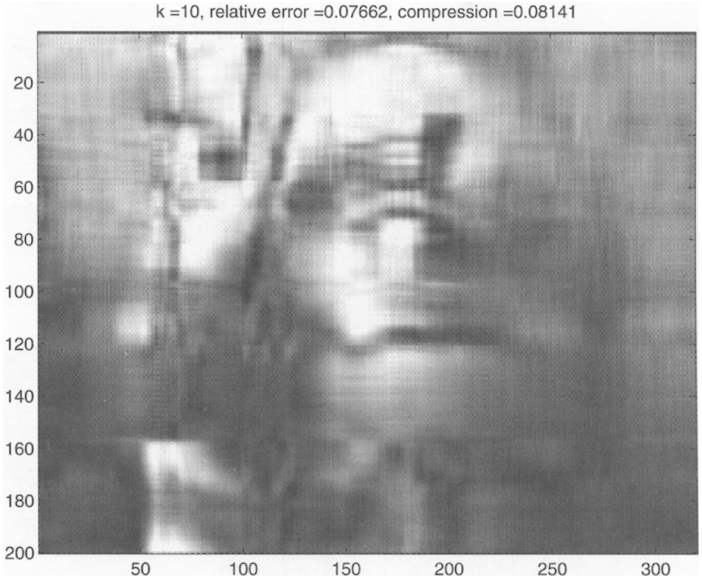


(a)

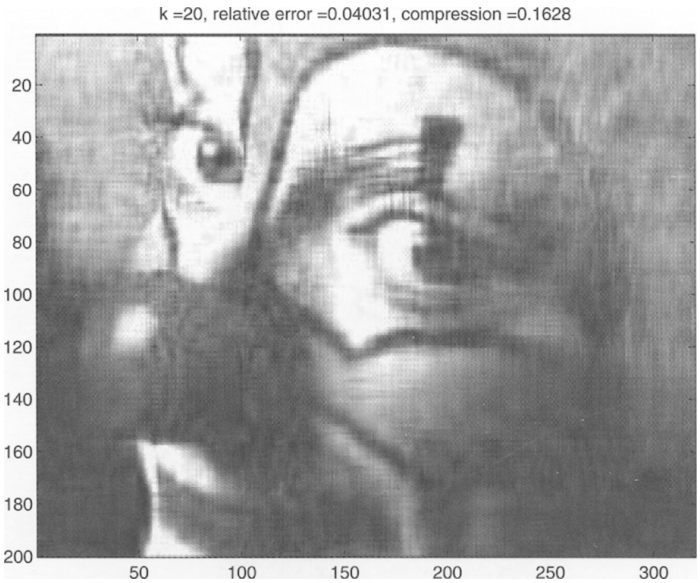


(b)

Fig. 3.3. *Image compression using the SVD.* (a) *Original image.* (b) *Rank* $k = 3$ *approximation.*



(c)



(d)

Fig. 3.3. *Continued.* (c) *Rank* $k = 10$ *approximation.* (d) *Rank* $k = 20$ *approximation.*

The pseudoinverse lets us write the solution of the full-rank, overdetermined least squares problem as simply $x = A^+ b$. If A is square and full rank, this formula reduces to $x = A^{-1}b$ as expected. The pseudoinverse of A is computed as `pinv(A)` in Matlab. When A is not full rank, the Moore–Penrose pseudoinverse is given by Definition 3.2 in section 3.5.

3.3. Perturbation Theory for the Least Squares Problem

When A is not square, we define its condition number with respect to the 2-norm to be $\kappa_2(A) \equiv \sigma_{\max}(A)/\sigma_{\min}(A)$. This reduces to the usual condition number when A is square. The next theorem justifies this definition.

THEOREM 3.4. *Suppose that A is m-by-n with $m \geq n$ and has full rank. Suppose that x minimizes $\|Ax-b\|_2$. Let $r = Ax-b$ be the residual. Let $\tilde{x}$ minimize $\|(A+\delta A)\tilde{x} - (b+\delta b)\|_2$. Assume $\epsilon \equiv \max(\frac{\|\delta A\|_2}{\|A\|_2}, \frac{\|\delta b\|_2}{\|b\|_2}) < \frac{1}{\kappa_2(A)} = \frac{\sigma_{\min}(A)}{\sigma_{\max}(A)}$. Then*

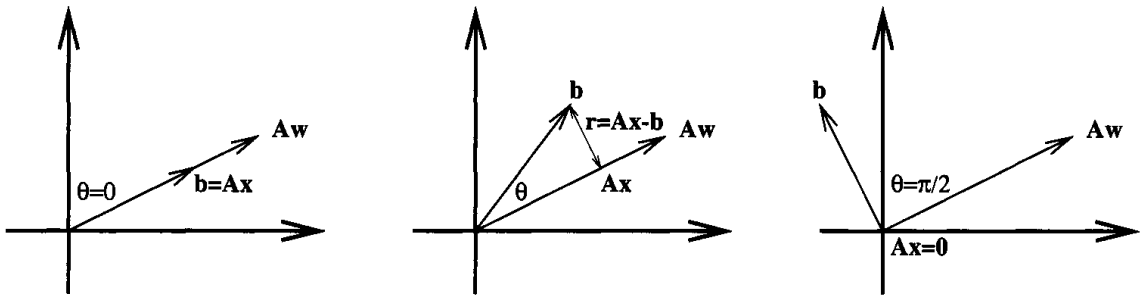
$$\frac{\|\tilde{x} - x\|_2}{\|x\|_2} \leq \epsilon \cdot \left\{ \frac{2 \cdot \kappa_2(A)}{\cos\theta} + \tan\theta \cdot \kappa_2^2(A) \right\} + O(\epsilon^2) \equiv \epsilon \cdot \kappa_{LS} + O(\epsilon^2),$$

where $\sin\theta = \frac{\|r\|_2}{\|b\|_2}$. In other words, θ is the angle between the vectors b and Ax and measures whether the residual norm $\|r\|_2$ is large (near $\|b\|$) or small (near 0). κ_{LS} is the condition number for the least squares problem.

Sketch of Proof. Expand $\tilde{x} = \left((A+\delta A)^T(A+\delta A)\right)^{-1}(A+\delta A)^T(b+\delta b)$ in powers of δA and δb, and throw away all but the linear terms in δA and δb. □

We have assumed that $\epsilon \cdot \kappa_2(A) < 1$ for the same reason as in the derivation of bound (2.4) for the perturbed solution of the square linear system $Ax = b$: it guarantees that $A + \delta A$ has full rank so that $\tilde{x}$ is uniquely determined.

We may interpret this bound as follows. If θ is 0 or very small, then the residual is small and the effective condition number is about $2\kappa_2(A)$, much like ordinary linear equation solving. If θ is not small but not close to $\pi/2$, the residual is moderately large, and then the effective condition number can be much larger: $\kappa_2^2(A)$. If θ is close to $\pi/2$, so the true solution is nearly zero, then the effective condition number becomes unbounded even if $\kappa_2(A)$ is small. These three cases are illustrated below. The right-hand picture makes it easy to see why the condition number is infinite when $\theta = \pi/2$: in this case the solution $x = 0$, and almost any arbitrarily small change in A or b will yield a nonzero solution x, an "infinitely" large relative change.



k_LS = 2k(A) k_LS = O(k(A)^2) k_LS = infinity

An alternative form for the bound in Theorem 3.4 that eliminates the $O(\epsilon^2)$ term is as follows [258, 149] (here $\tilde{r}$ is the perturbed residual $\tilde{r} = (A+\delta A)\tilde{x}) - (b+\delta b)$):

$$\begin{aligned} \frac{\|\tilde{x}-x\|_2}{\|x\|_2} &\leq \frac{\epsilon\kappa_2(A)}{1-\epsilon\kappa_2(A)}\left(2+(\kappa_2(A)+1)\frac{\|r\|_2}{\|A\|_2\|x\|_2}\right), \\ \frac{\|\tilde{r}-r\|_2}{\|r\|_2} &\leq (1+2\epsilon\kappa_2(A)). \end{aligned}$$

We will see that, properly implemented, both the QR decomposition and SVD are numerically stable; i.e., they yield a solution $\tilde{x}$ minimizing $\|(A+\delta A)\tilde{x}-(b+\delta b)\|_2$ with

$$\max\left(\frac{\|\delta A\|}{\|A\|}, \frac{\|\delta b\|}{\|b\|}\right) = O(\varepsilon).$$

We may combine this with the above perturbation bounds to get error bounds for the solution of the least squares problem, much as we did for linear equation solving.

The normal equations are not as accurate. Since they involve solving $(A^TA)x = A^Tb$, the accuracy depends on the condition number $\kappa_2(A^TA) = \kappa_2^2(A)$. Thus the error is always bounded by $\kappa_2^2(A)\varepsilon$, never just $\kappa_2(A)\varepsilon$. Therefore we expect that the normal equations can lose twice as many digits of accuracy as methods based on the QR decomposition and SVD.

Furthermore, solving the normal equations is *not* necessarily stable; i.e., the computed solution $\tilde{x}$ does not generally minimize $\|(A+\delta A)\tilde{x}-(b+\delta b)\|_2$ for small δA and δb. Still, when the condition number is small, we expect the normal equations to be about as accurate as the QR decomposition or SVD. Since the normal equations are the fastest way to solve the least squares problem, they are the method of choice when the matrix is well-conditioned.

We return to the problem of solving very ill-conditioned least squares problems in section 3.5.

3.4. Orthogonal Matrices

As we said in section 3.2.2, Gram–Schmidt orthogonalization (Algorithm 3.1) may not compute an orthogonal matrix Q when the vectors being orthogonal-

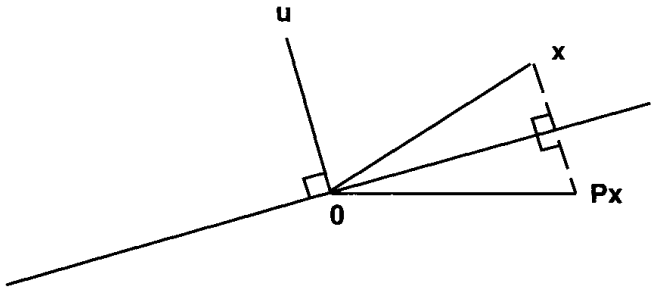
ized are nearly linearly dependent, so we cannot use it to compute the QR decomposition stably.

Instead, we base our algorithms on certain easily computable orthogonal matrices called *Householder reflections* and *Givens rotations*, which we can choose to introduce zeros into vectors that they multiply. Later we will show that any algorithm that uses these orthogonal matrices to introduce zeros is automatically stable. This error analysis will apply to our algorithms for the QR decomposition as well as many SVD and eigenvalue algorithms in Chapters 4 and 5.

Despite the possibility of nonorthogonal Q, the MGS algorithm has important uses in numerical linear algebra. (There is little use for its less stable version, CGS.) These uses include finding eigenvectors of symmetric tridiagonal matrices using bisection and inverse iteration (section 5.3.4) and the Arnoldi and Lanczos algorithms for reducing a matrix to certain "condensed" forms (sections 6.6.1, 6.6.6, and 7.4). Arnoldi and Lanczos algorithms are used as the basis of algorithms for solving sparse linear systems and finding eigenvalues of sparse matrices. MGS can also be modified to solve the least squares problem stably, but Q may still be far from orthogonal [33].

3.4.1. Householder Transformations

A Householder transformation (or reflection) is a matrix of the form $P = I - 2uu^T$ where $\|u\|_2 = 1$. It is easy to see that $P = P^T$ and $PP^T = (I - 2uu^T)(I - 2uu^T) = I - 4uu^T + 4uu^Tuu^T = I$, so P is a symmetric, orthogonal matrix. It is called a reflection because Px is reflection of x in the plane through 0 perpendicular to u.



Given a vector x, it is easy to find a Householder reflection $P = I - 2uu^T$ to zero out all but the first entry of x: $Px = [c, 0, \ldots, 0]^T = c \cdot e_1$. We do this as follows. Write $Px = x - 2u(u^Tx) = c \cdot e_1$ so that $u = \frac{1}{2(u^Tx)}(x - ce_1)$; i.e., u is a linear combination of x and e_1. Since $\|x\|_2 = \|Px\|_2 = |c|$, u must be parallel to the vector $\tilde{u} = x \pm \|x\|_2 e_1$, and so $u = \tilde{u}/\|\tilde{u}\|_2$. One can verify that either choice of sign yields a u satisfying $Px = ce_1$, as long as $\tilde{u} \neq 0$. We will use $\tilde{u} = x + \text{sign}(x_1)e_1$, since this means that there is no cancellation in

computing the first component of $\tilde{u}$. In summary, we get

$$\tilde{u} = \begin{bmatrix} x_1 + \text{sign}(x_1) \cdot \|x\|_2 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{with } u = \frac{\tilde{u}}{\|\tilde{u}\|_2}.$$

We write this as $u = \text{House}(x)$. (In practice, we can store $\tilde{u}$ instead of u to save the work of computing u, and use the formula $P = I - (2/\|\tilde{u}\|_2^2)\tilde{u}\tilde{u}^T$ instead of $P = I - 2uu^T$.)

EXAMPLE 3.5. We show how to compute the QR decomposition of a 5-by-4 matrix A using Householder transformations. This example will make the pattern for general m-by-n matrices evident. In the matrices below, P_i is a 5-by-5 orthogonal matrix, x denotes a generic nonzero entry, and o denotes a zero entry.

1. Choose P_1 so $$A_1 \equiv P_1 A = \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & x & x & x \\ o & x & x & x \\ o & x & x & x \end{bmatrix}.$$

2. Choose $P_2 = \left[\begin{array}{c|cc} 1 & 0 & \\ \hline 0 & & P_2' \end{array}\right]$ so $$A_2 \equiv P_2 A_1 = \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & x & x \\ o & o & x & x \end{bmatrix}.$$

3. Choose $P_3 = \left[\begin{array}{cc|c} 1 & & \\ & 1 & 0 \\ \hline & 0 & P_3' \end{array}\right]$ so $$A_3 \equiv P_3 A_2 = \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & o & x \\ o & o & o & x \end{bmatrix}.$$

4. Choose $P_4 = \left[\begin{array}{ccc|c} 1 & & & \\ & 1 & & 0 \\ & & 1 & \\ \hline & 0 & & P_4' \end{array}\right]$ so $$A_4 \equiv P_4 A_3 = \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & o & x \\ o & o & o & o \end{bmatrix}.$$

Here, we have chosen a Householder matrix P_i' to zero out the subdiagonal entries in column i; this does not disturb the zeros already introduced in previous columns.

Let us call the final 5-by-4 upper triangular matrix $\tilde{R} \equiv A_4$. Then $A = P_1^T P_2^T P_3^T P_4^T \tilde{R} = QR$, where Q is the first four columns of $P_1^T P_2^T P_3^T P_4^T = P_1 P_2 P_3 P_4$ (since all P_i are symmetric) and R is the first four rows of $\tilde{R}$. $\diamond$

Here is the general algorithm for QR decomposition using Householder transformations.

ALGORITHM 3.2. *QR factorization using Householder reflections:*

for $i = 1$ *to* $\min(m-1, n)$
  $u_i = \text{House}(A(i:m, i))$
  $P_i' = I - 2u_i u_i^T$
  $A(i:m, i:n) = P_i' A(i:m, i:n)$
end for

Here are some more implementation details. We never need to form P_i explicitly but just multiply

$$(I - 2u_i u_i^T)A(i:m, i:n) = A(i:m, i:n) - 2u_i(u_i^T A(i:m, i:n)),$$

which costs less. To store P_i, we need only u_i, or $\tilde{u}_i$ and $\|\tilde{u}_i\|$. These can be stored in column i of A; in fact it need not be changed! Thus QR can be "overwritten" on A, where Q is stored in factored form $P_1 \cdots P_{n-1}$, and P_i is stored as $\tilde{u}_i$ below the diagonal in column i of A. (We need an extra array of length n for the top entry of $\tilde{u}_i$, since the diagonal entry is occupied by R_{ii}.)

Recall that to solve the least squares problem $\min \|Ax - b\|_2$ using $A = QR$, we need to compute $Q^T b$. This is done as follows: $Q^T b = P_n P_{n-1} \cdots P_1 b$, so we need only keep multiplying b by $P_1, P_2, \ldots, P_n$:

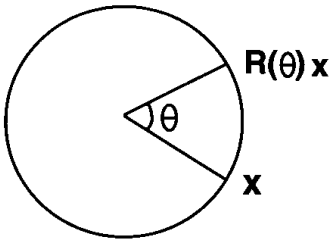
for $i = 1$ *to* n
  $\gamma = -2 \cdot u_i^T b(i:m)$
  $b(i:m) = b(i:m) + \gamma u_i$
end for

The cost is n dot products $\gamma = -2 \cdot u_i^T b$ and n "saxpys" $b + \gamma u_i$. The cost of computing $A = QR$ this way is $2n^2 m - \frac{2}{3}n^3$, and the subsequent cost of solving the least squares problem given QR is just an additional $O(mn)$.

The LAPACK routine for solving the least squares problem using QR is `sgels`. Just as Gaussian elimination can be reorganized to use matrix-matrix multiplication and other Level 3 BLAS (see section 2.6), the same can be done for the QR decomposition; see Question 3.17. In Matlab, if the m-by-n matrix A has more rows than columns and b is m by 1, `A\b` solves the least squares problem. The QR decomposition itself is also available via [Q,R]=qr(A).

3.4.2. Givens Rotations

A Givens rotation $R(\theta) \equiv \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ rotates any vector $x \in \mathbb{R}^2$ counterclockwise by θ:



We also need to define the Givens rotation by θ in coordinates i and j:

$$R(i,j,\theta) \equiv \begin{array}{c} \\ i \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccccc} & & & i & & j & & & \end{array} \\ \begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & \ddots & & & & & & \\ & & & \cos\theta & & -\sin\theta & & & \\ & & & & \ddots & & & & \\ & & & \sin\theta & & \cos\theta & & & \\ & & & & & & \ddots & & \\ & & & & & & & 1 & \\ & & & & & & & & 1 \end{bmatrix} \end{array}.$$

Given x, i, and j, we can zero out x_j by choosing $\cos\theta$ and $\sin\theta$ so that

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_i \\ x_j \end{bmatrix} = \begin{bmatrix} \sqrt{x_i^2 + x_j^2} \\ 0 \end{bmatrix}$$

or $\cos\theta = \frac{x_i}{\sqrt{x_i^2+x_j^2}}$ and $\sin\theta = \frac{-x_j}{\sqrt{x_i^2+x_j^2}}$.

The QR algorithm using Givens rotations is analogous to using Householder reflections, but when zeroing out column i, we zero it out one entry at a time (bottom to top, say).

EXAMPLE 3.6. We illustrate two intermediate steps in computing the QR decomposition of a 5-by-4 matrix using Givens rotations. To progress from

$$\begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & x & x \\ o & o & x & x \end{bmatrix} \quad \text{to} \quad \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & o & x \\ o & o & o & x \end{bmatrix}$$

we multiply

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & c & -s \\ & & & s & c \end{bmatrix} \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & x & x \\ o & o & x & x \end{bmatrix} = \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & x & x \\ o & o & o & x \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & c' & -s' & \\ & & s' & c' & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & x & x \\ o & o & o & x \end{bmatrix} = \begin{bmatrix} x & x & x & x \\ o & x & x & x \\ o & o & x & x \\ o & o & o & x \\ o & o & o & x \end{bmatrix}. \quad \diamond$$

The cost of the QR decomposition using Givens rotations is twice the cost of using Householder reflections. We will need Givens rotations for other applications later.

Here are some implementation details. Just as we overwrote A with Q and R when using Householder reflections, we can do the same with Givens rotations. We use the same trick, storing the information describing the transformation in the entries zeroed out. Since a Givens rotation zeros out just one entry, we must store the information about the rotation there. We do this as follows. Let $s = \sin\theta$ and $c = \cos\theta$. If $|s| < |c|$, store $s \cdot \text{sign}(c)$ and otherwise store $\frac{\text{sign}(s)}{c}$. To recover s and c from the stored value (call it p) we do the following: if $|p| < 1$, then $s = p$ and $c = \sqrt{1-s^2}$; otherwise $c = \frac{1}{p}$ and $s = \sqrt{1-c^2}$. The reason we do not just store s and compute $c = \sqrt{1-s^2}$ is that when s is close to 1, c would be inaccurately reconstructed. Note also that we may recover either s and c or $-s$ and $-c$; this is adequate in practice.

There is also a way to apply a sequence of Givens rotations while performing fewer floating point operations than described above. These are called *fast Givens rotations* [7, 8, 33]. Since they are still slower than Householder reflections for the purposes of computing the QR factorization, we will not consider them further.

3.4.3. Roundoff Error Analysis for Orthogonal Matrices

This analysis proves backward stability for the QR decomposition and for many of the algorithms for eigenvalues and singular values that we will discuss.

LEMMA 3.1. *Let P be an exact Householder (or Givens) transformation, and $\tilde{P}$ be its floating point approximation. Then*

$$\text{fl}(\tilde{P}A) = P(A+E) \quad \|E\|_2 = O(\varepsilon) \cdot \|A\|_2$$

and

$$\text{fl}(A\tilde{P}) = (A+F)P \quad \|F\|_2 = O(\varepsilon) \cdot \|A\|_2.$$

Sketch of Proof. Apply the usual formula $\text{fl}(a \odot b) = (a \odot b)(1+\varepsilon)$ to the formulas for computing and applying $\tilde{P}$. See Question 3.16. □

In words, this says that applying a single orthogonal matrix is backward stable.

THEOREM 3.5. *Consider applying a sequence of orthogonal transformations to A_0. Then the computed product is an exact orthogonal transformation of $A_0 + \delta A$, where $\|\delta A\|_2 = O(\varepsilon)\|A\|_2$. In other words, the entire computation is backward stable:*

$$\mathrm{fl}(\tilde{P}_j \tilde{P}_{j-1} \cdots \tilde{P}_1 A_0 \tilde{Q}_1 \tilde{Q}_2 \cdots \tilde{Q}_j) = P_j \cdots P_1 (A_0 + E) Q_1 \cdots Q_j$$

with $\|E\|_2 = j \cdot O(\varepsilon) \cdot \|A\|_2$. Here, as in Lemma 3.1, $\tilde{P}_i$ and $\tilde{Q}_i$ are floating point orthogonal matrices and P_i and Q_i are exact orthogonal matrices.

Proof. Let $\bar{P}_j \equiv P_j \cdots P_1$ and $\bar{Q}_j \equiv Q_1 \cdots Q_j$. We wish to show that $A_j \equiv \mathrm{fl}(\tilde{P}_j A_{j-1} \tilde{Q}_j) = \bar{P}_j (A + E_j) \bar{Q}_j$ for some $\|E_j\|_2 = jO(\varepsilon)\|A\|_2$. We use Lemma 3.1 recursively. The result is vacuously true for $j = 0$. Now assume that the result is true for $j - 1$. Then we compute

$$\begin{aligned} B &= \mathrm{fl}(\tilde{P}_j A_{j-1}) \\ &= P_j (A_{j-1} + E') \text{ by Lemma 3.1} \\ &= P_j (\bar{P}_{j-1}(A + E_{j-1})\bar{Q}_{j-1} + E') \text{ by induction} \\ &= \bar{P}_j (A + E_{j-1} + \bar{P}_{j-1}^T E' \bar{Q}_{j-1}^T) \bar{Q}_{j-1} \\ &\equiv \bar{P}_j (A + E'') \bar{Q}_{j-1}, \end{aligned}$$

where

$$\begin{aligned} \|E''\|_2 &= \|E_{j-1} + \bar{P}_{j-1}^T E' \bar{Q}_{j-1}^T\|_2 \le \|E_{j-1}\|_2 + \|\bar{P}_{j-1}^T E' \bar{Q}_{j-1}^T\|_2 \\ &= \|E_{j-1}\|_2 + \|E'\|_2 \\ &= jO(\varepsilon)\|A\|_2 \end{aligned}$$

since $\|E_{j-1}\|_2 = (j-1)O(\varepsilon)\|A\|_2$ and $\|E'\|_2 = O(\varepsilon)\|A\|_2$. Postmultiplication by $\tilde{Q}_j$ is handled in the same way. □

3.4.4. Why Orthogonal Matrices?

Let us consider how the error would grow if we were to multiply by a sequence of *nonorthogonal* matrices in Theorem 3.5 instead of orthogonal matrices. Let X be the exact nonorthogonal transformation and $\tilde{X}$ be its floating point approximation. Then the usual floating point error analysis of matrix multiplication tells us that

$$\mathrm{fl}(\tilde{X}A) = XA + E = X(A + X^{-1}E) \equiv X(A + F),$$

where $\|E\|_2 \le O(\varepsilon)\|X\|_2 \cdot \|A\|_2$ and so $\|F\|_2 \le \|X^{-1}\|_2 \cdot \|E\|_2 \le O(\varepsilon) \cdot \kappa_2(X) \cdot \|A\|_2$.

So the error $\|E\|_2$ is magnified by the condition number $\kappa_2(X) \ge 1$. In a larger product $\tilde{X}_k \cdots \tilde{X}_1 A \tilde{Y}_1 \cdots \tilde{Y}_k$ the error would be magnified by $\prod_i \kappa_2(X_i) \cdot \kappa_2(Y_i)$. This factor is minimized if and only if all X_i and Y_i are orthogonal (or scalar multiples of orthogonal matrices), in which case the factor is one.

3.5. Rank-Deficient Least Squares Problems

So far we have assumed that A has full rank when minimizing $\|Ax - b\|_2$. What happens when A is rank deficient or "close" to rank deficient? Such problems arise in practice in many ways, such as extracting signals from noisy data, solution of some integral equations, digital image restoration, computing inverse Laplace transforms, and so on [141, 142]. These problems are very ill-conditioned, so we will need to impose extra conditions on their solutions to make them well-conditioned. Making an ill-conditioned problem well-conditioned by imposing extra conditions on the solution is called *regularization* and is also done in other fields of numerical analysis when ill-conditioned problems arise.

For example, the next proposition shows that if A is exactly rank deficient, then the least squares solution is not even unique.

PROPOSITION 3.1. *Let A be m-by-n with $m \geq n$ and rank $A = r < n$. Then there is an $n - r$ dimensional set of vectors x that minimize $\|Ax - b\|_2$.*

Proof. Let $Az = 0$. Then if x minimizes $\|Ax - b\|_2$, so does $x + z$. □

Because of roundoff in the entries of A, or roundoff during the computation, it is most often the case that A will have one or more very small computed singular values, rather than some exactly zero singular values. The next proposition shows that in this case, the unique solution is likely to be very large and is certainly very sensitive to error in the right-hand side b (see also Theorem 3.4).

PROPOSITION 3.2. *Let $\sigma_{\min} = \sigma_{\min}(A)$, the smallest singular value of A. Assume $\sigma_{\min} > 0$. Then*

1. *if x minimizes $\|Ax - b\|_2$, then $\|x\|_2 \geq |u_n^T b|/\sigma_{\min}$, where u_n is the last column of U in $A = U\Sigma V^T$.*

2. *changing b to $b + \delta b$ can change x to $x + \delta x$, where $\|\delta x\|_2$ is as large as $\|\delta b\|_2/\sigma_{\min}$.*

In other words, if A is nearly rank deficient ($\sigma_{\min}$ is small), then the solution x is ill-conditioned and possibly very large.

Proof. For part 1, $x = A^+ b = V\Sigma^{-1}U^T b$, so $\|x\|_2 = \|\Sigma^{-1}U^T b\|_2 \geq |(\Sigma^{-1}U^T b)_n| = |u_n^T b|/\sigma_{\min}$. For part 2, choose δb parallel to u_n. □

We begin our discussion of regularization by showing how to regularize an *exactly* rank-deficient least squares problem: Suppose A is m-by-n with rank $r < n$. Within the $(n - r)$-dimensional solution space, we will look for the unique solution of smallest norm. This solution is characterized by the following proposition.

PROPOSITION 3.3. *When A is exactly singular, the x that minimize $\|Ax-b\|_2$ can be characterized as follows. Let $A = U\Sigma V^T$ have rank $r < n$, and write the SVD of A as*

$$A = [U_1, U_2] \begin{bmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{bmatrix} [V_1, V_2]^T = U_1 \Sigma_1 V_1^T, \tag{3.1}$$

where Σ_1 is $r \times r$ and nonsingular and U_1 and V_1 have r columns. Let $\sigma = \sigma_{\min}(\Sigma_1)$, the smallest nonzero *singular value of A. Then*

1. *all solutions x can be written $x = V_1\Sigma_1^{-1}U_1^T b + V_2 z$, z an arbitrary vector.*
2. *the solution x has minimal norm $\|x\|_2$ precisely when $z = 0$, in which case $x = V_1\Sigma_1^{-1}U_1^T b$ and $\|x\|_2 \le \|b\|_2/\sigma$.*
3. *changing b to $b+\delta b$ can change the minimal norm solution x by at most $\|\delta b\|_2/\sigma$.*

In other words, the norm and condition number of the unique minimal norm solution x depend on the smallest nonzero *singular value of A.*

Proof. Choose $\tilde{U}$ so $[U, \tilde{U}] = [U_1, U_2, \tilde{U}]$ is an $m \times m$ orthogonal matrix. Then

$$\begin{aligned} \|Ax-b\|_2^2 &= \|[U,\tilde{U}]^T(Ax-b)\|_2^2 \\ &= \left\| \begin{bmatrix} U_1^T \\ U_2^T \\ \tilde{U}^T \end{bmatrix} (U_1\Sigma_1 V_1^T x - b) \right\|_2^2 \\ &= \left\| \begin{bmatrix} \Sigma_1 V_1^T x - U_1^T b \\ U_2^T b \\ \tilde{U}^T b \end{bmatrix} \right\|_2^2 \\ &= \|\Sigma_1 V_1^T x - U_1^T b\|_2^2 + \|U_2^T b\|_2^2 + \|\tilde{U}^T b\|_2^2. \end{aligned}$$

1. $\|Ax-b\|_2$ is minimized when $\Sigma_1 V_1^T x = U_1^T b$, or $x = V_1\Sigma_1^{-1}U_1^T b + V_2 z$ since $V_1^T V_2 z = 0$ for all z.
2. Since the columns of V_1 and V_2 are mutually orthogonal, the Pythagorean theorem implies that $\|x\|_2^2 = \|V_1\Sigma_1^{-1}U_1^T b\|_2^2 + \|V_2 z\|_2^2$, and this is minimized by $z = 0$.
3. Changing b by δb changes x by at most $\|V_1\Sigma_1^{-1}U_1^T\delta b\|_2 \le \|\Sigma_1^{-1}\|_2\|\delta b\|_2 = \|\delta b\|_2/\sigma$. □

Proposition 3.3 tells us that the minimum norm solution x is unique and may be well-conditioned if the smallest *nonzero* singular value is not too small. This is key to a practical algorithm, discussed in the next section.

EXAMPLE 3.7. Suppose that we are doing medical research on the effect of a certain drug on blood sugar level. We collect data from each patient (numbered from $i = 1$ to m) by recording his or her initial blood sugar level ($a_{i,1}$), final blood sugar level (b_i), the amount of drug administered ($a_{i,2}$), and other medical quantities, including body weights on each day of a week-long treatment ($a_{i,3}$ through $a_{i,9}$). In total, there are $n < m$ medical quantities measured for each patient. Our goal is to predict b_i given $a_{i,1}$ through $a_{i,n}$, and we formulate this as the least squares problem $\min_x \|Ax - b\|_2$. We plan to use x to predict the final blood sugar level b_j of future patient j by computing the dot product $\sum_{k=1}^{n} a_{jk}x_k$.

Since people's weight generally does not change significantly from day to day, it is likely that columns 3 through 9 of matrix A, which contain the weights, are very similar. For the sake of argument, suppose that columns 3 and 4 are *identical* (which may be the case if the weights are rounded to the nearest pound). This means that matrix A is rank deficient and that $x_0 = [0, 0, 1, -1, 0, \ldots, 0]^T$ is a right null vector of A. So if x is a (minimum norm) solution of the least squares problem $\min_x \|Ax - b\|_2$, then $x + \beta x_0$ is also a (nonminimum norm) solution for *any* scalar β, including, say, $\beta = 0$ and $\beta = 10^6$. Is there any reason to prefer one value of β over another? The value 10^6 is clearly not a good one, since future patient j, who gains one pound between days 1 and 2, will have that difference of one pound multiplied by 10^6 in the predictor $\sum_{k=1}^{n} a_{jk}x_k$ of final blood sugar level. It is much more reasonable to choose $\beta = 0$, corresponding to the minimum norm solution x. ◇

For further justification of using the minimum norm solution for rank-deficient problems, see [141, 142].

When A is square and nonsingular, the unique solution of $Ax = b$ is of course $b = A^{-1}x$. If A has more rows than columns and is possibly rank-deficient, the unique minimum-norm least squares solution may be similarly written $b = A^+b$, where the *Moore–Penrose pseudoinverse* A^+ is defined as follows.

DEFINITION 3.2. *(Moore–Penrose pseudoinverse A^+ for possibly rank-deficient A)*

Let $A = U\Sigma V^T = U_1\Sigma_1 V_1^T$ as in equation (3.1). Then $A^+ \equiv V_1\Sigma_1^{-1}U_1^T$. This is also written $A^+ = V^T\Sigma^+U$, where $\Sigma^+ = \begin{bmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{bmatrix}^+ = \begin{bmatrix} \Sigma_1^{-1} & 0 \\ 0 & 0 \end{bmatrix}$.

So the solution of the least squares problem is always $x = A^+b$, and when A is rank deficient, x has minimum norm.

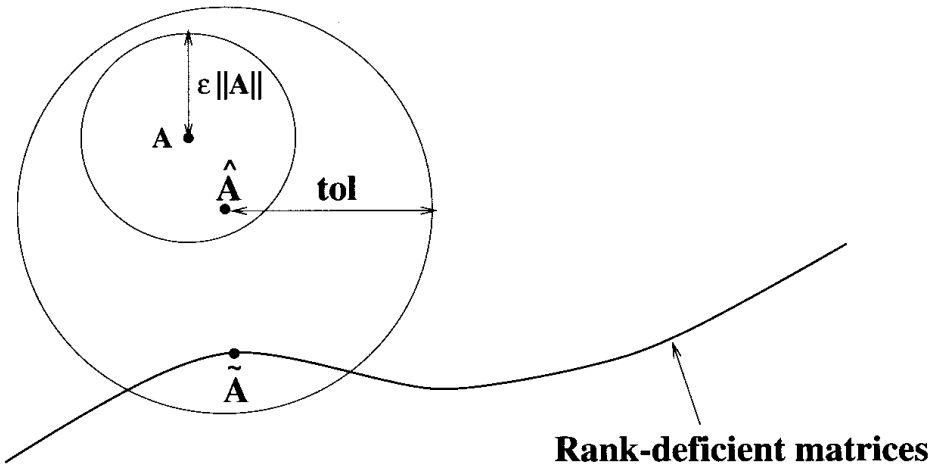
3.5.1. Solving Rank-Deficient Least Squares Problems Using the SVD

Our goal is to compute the minimum norm solution x, despite roundoff. In the last section, we saw that the minimal norm solution was unique and had a condition number depending on the smallest nonzero singular value. Therefore, computing the minimum norm solution requires knowing the smallest nonzero singular value and hence also the rank of A. The main difficulty is that the rank of a matrix changes discontinuously as a function of the matrix.

For example, the 2-by-2 matrix $A = \mathrm{diag}(1, 0)$ is exactly singular, and its smallest nonzero singular value is $\sigma = 1$. As described in Proposition 3.3, the minimum norm least squares solution to $\min_x \|Ax - b\|_2$ with $b = [1, 1]^T$ is $x = [1, 0]^T$, with condition number $1/\sigma = 1$. But if we make an arbitrarily tiny perturbation to get $\hat{A} = \mathrm{diag}(1, \epsilon)$, then σ drops to ϵ and $x = [1, 1/\epsilon]^T$ becomes enormous, as does its condition number $1/\epsilon$. In general, roundoff will make such tiny perturbations, of magnitude $O(\varepsilon)\|A\|_2$. As we just saw, this can increase the condition number from $1/\sigma$ to $1/\varepsilon$.

We deal with this discontinuity algorithmically as follows. In general each computed singular value $\hat{\sigma}_i$ satisfies $|\hat{\sigma}_i - \sigma_i| \leq O(\varepsilon)\|A\|_2$. This is a consequence of backward stability: the computed SVD will be the exact SVD of a slightly different matrix: $\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^T = A + \delta A$, with $\|\delta A\| = O(\varepsilon) \cdot \|A\|$. (This is discussed in detail in Chapter 5.) This means that any $\hat{\sigma}_i \leq O(\varepsilon)\|A\|_2$ can be treated as zero, because roundoff makes it indistinguishable from 0. In the above 2-by-2 example, this means we would set the ϵ in $\hat{A}$ to zero before solving the least squares problem. This would raise the smallest nonzero singular value from ϵ to 1 and correspondingly decrease the condition number from $1/\epsilon$ to $1/\sigma = 1$.

More generally, let tol be a user-supplied measure of uncertainty in the data A. Roundoff implies that tol $\geq \varepsilon \cdot \|A\|$, but it may be larger, depending on the source of the data in A. Now set $\tilde{\sigma}_i = \hat{\sigma}_i$ if $\hat{\sigma}_i >$ tol, and $\tilde{\sigma}_i = 0$ otherwise. Let $\tilde{\Sigma} = \mathrm{diag}(\tilde{\sigma}_i)$. We call $\hat{U}\tilde{\Sigma}\hat{V}^T$ the *truncated SVD* of A, because we have set singular values smaller than tol to zero. Now we solve the least squares problem using the truncated SVD instead of the original SVD. This is justified since $\|\hat{U}\tilde{\Sigma}\hat{V}^T - \hat{U}\hat{\Sigma}\hat{V}^T\|_2 = \|\hat{U}(\tilde{\Sigma} - \hat{\Sigma})\hat{V}^T\|_2 <$ tol; i.e., the change in A caused by changing each $\hat{\sigma}_i$ to $\tilde{\sigma}_i$ is less than the user's inherent uncertainty in the data. The motivation for using $\tilde{\Sigma}$ instead of $\hat{\Sigma}$ is that of all matrices within distance tol of $\hat{\Sigma}$, $\tilde{\Sigma}$ maximizes the smallest nonzero singular value σ. In other words, it minimizes both the norm of the minimum norm least squares solution x and its condition number. The picture below illustrates the geometric relationships among the input matrix A, $\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^T$, and $\tilde{A} = \hat{U}\tilde{\Sigma}\hat{V}^T$, where we we think of each matrix as a point in Euclidean space $\mathbb{R}^{m \cdot n}$. In this space, the rank-deficient matrices form a surface, as shown below:



EXAMPLE 3.8. We illustrate the above procedure on two 20-by-10 rank-deficient matrices A_1 (of rank $r_1 = 5$) and A_2 (of rank $r_2 = 7$). We write the SVDs of either A_1 or A_2 as $A_i = U_i \Sigma_i V_i^T$, where the common dimension of U_i, Σ_i, and V_i is the rank r_i of A_i; this is the same notation as in Proposition 3.3. The r_i nonzero singular values of A_i (singular values of Σ_i) are shown as red x's in Figure 3.4 (for A_1) and Figure 3.5 (for A_2). Note that A_1 in Figure 3.4 has five large nonzero singular values (all slightly exceeding 1 and so plotted on top of one another, on the right edge the graph), whereas the seven nonzero singular values of A_2 in Figure 3.5 range down to $1.2 \cdot 10^{-9} \approx$ tol.

We then choose an r_i-dimensional vector x_i', and let $x_i = V_i x_i'$ and $b_i = A_i x_i = U_i \Sigma_i x_i'$, so x_i is the exact minimum norm solution minimizing $\|A_i x_i - b_i\|_2$. Then we consider a sequence of perturbed problems $A_i + \delta A$, where the perturbation δA is chosen randomly to have a range of norms, and solve the least squares problems $\|(A_i + \delta A)y_i - b_i\|_2$ using the truncated least squares procedure with tol $= 10^{-9}$. The blue lines in Figures 3.4 and 3.5 plot the computed rank of $A_i + \delta A$ (number of computed singular values exceeding tol $= 10^{-9}$) versus $\|\delta A\|_2$ (in the top graphs), and the error $\|y_i - x_i\|_2 / \|x_i\|_2$ (in the bottom graphs). The Matlab code for producing these figures is in HOMEPAGE/Matlab/RankDeficient.m.

The simplest case is in Figure 3.4, so we consider it first. $A_1 + \delta A$ will have five singular values near or slightly exceeding 1 and the other five equal to $\|\delta A\|_2$ or less. For $\|\delta A\|_2 <$ tol, the computed rank of $A_1 + \delta A$ stays the same as that of A_1, namely, 5. The error also increases slowly from near machine epsilon ($\approx 10^{-16}$) to about 10^{-10} near $\|\delta A\|_2 =$ tol, and then both the rank and the error jump, to 10 and 1, respectively, for larger $\|\delta A\|_2$. This is consistent with our analysis in Proposition 3.3, which says that the condition number is the reciprocal of the smallest nonzero singular value, i.e., the smallest singular value exceeding tol. For $\|\delta A\|_2 <$ tol, this smallest nonzero singular value is near to, or slightly exceeds, 1. Therefore Proposition 3.3 predicts an error of $\|\delta A\|_2 / O(1) = \|\delta A\|_2$. This well-conditioned situation is confirmed by the small error plotted to the left of $\|\delta A\|_2 =$ tol in the bottom graph of Figure 3.4. On the other hand, when $\|\delta A\|_2 >$ tol, then the smallest nonzero

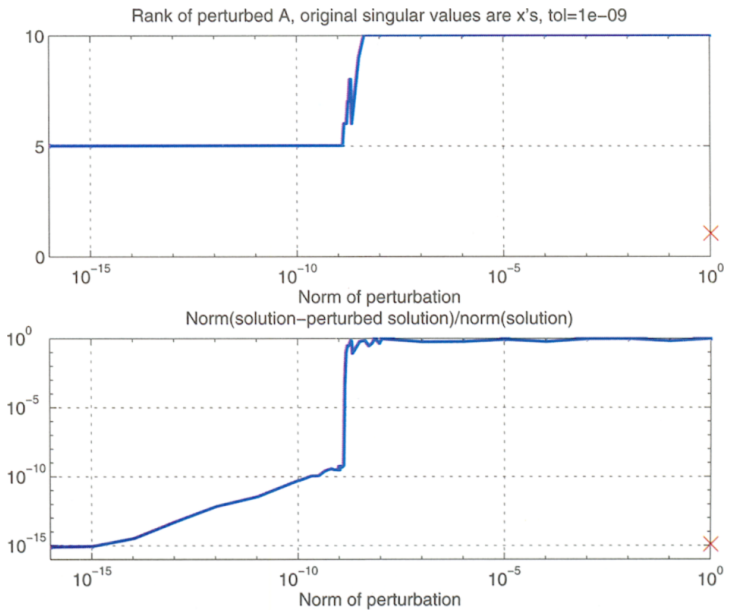


Fig. 3.4. *Graph of truncated least squares solution of* $\min_{y_1} \|(A_1 + \delta A)y_1 - b_1\|_2$, *using* tol $= 10^{-9}$. *The singular values of* A_1 *are shown as red x's. The norm* $\|\delta A\|_2$ *is the horizontal axis. The top graph plots the rank of* $A_1 + \delta A$, *i.e., the numbers of singular values exceeding* tol. *The bottom graph plots* $\|y_1 - x_1\|_2/\|x_1\|_2$, *where* x_1 *is the solution with* $\delta A = 0$.

singular value is $O(\|\delta A\|_2)$, which is quite small, causing the error to jump to $\|\delta A\|_2/O(\|\delta A\|_2) = O(1)$, as shown to the right of $\|\delta A\|_2 =$ tol in the bottom graph of Figure 3.4.

In Figure 3.5, the nonzero singular values of A_2 are also shown as red x's; the smallest one, $1.2 \cdot 10^{-9}$, is just larger than tol. So the predicted error when $\|\delta A\|_2 <$ tol is $\|\delta A\|_2/10^{-9}$, which grows to $O(1)$ when $\|\delta A\|_2 =$ tol. This is confirmed by the bottom graph in Figure 3.5. ◇

3.5.2. Solving Rank-Deficient Least Squares Problems Using QR with Pivoting

A cheaper but sometimes less accurate alternative to the SVD is *QR with pivoting*. In exact arithmetic, if A had rank $r < n$ and its first r columns were independent, then its QR decomposition would look like

$$A = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \\ 0 & 0 \end{bmatrix},$$

where R_{11} is r-by-r and nonsingular and R_{12} is r-by-$(n-r)$. With roundoff, we might hope to compute

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \\ 0 & 0 \end{bmatrix}$$

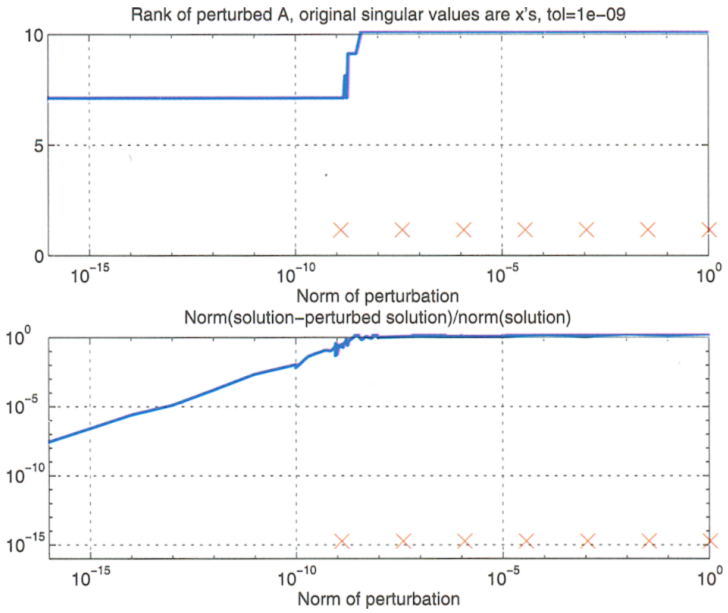


Fig. 3.5. *Graph of truncated least squares solution of* $\min_{y_2} \|(A_2 + \delta A)y_2 - b_2\|_2$, *using* tol $= 10^{-9}$. *The singular values of* A_2 *are shown as red* x*'s. The norm* $\|\delta A\|_2$ *is the horizontal axis. The top graph plots the rank of* $A_2 + \delta A$, *i.e., the numbers of singular values exceeding* tol. *The bottom graph plots* $\|y_2 - x_2\|_2/\|x_2\|_2$, *where* x_2 *is the solution with* $\delta A = 0$.

with $\|R_{22}\|_2$ very small, on the order of $\varepsilon\|A\|_2$. In this case we could just set $R_{22} = 0$ and minimize $\|Ax-b\|_2$ as follows: let $[Q, \tilde{Q}]$ be square and orthogonal so that

$$\begin{aligned} \|Ax - b\|_2^2 &= \left\| \begin{bmatrix} Q^T \\ \tilde{Q}^T \end{bmatrix} (Ax - b) \right\|_2^2 = \left\| \begin{bmatrix} Rx - Q^T b \\ -\tilde{Q}^T b \end{bmatrix} \right\|_2^2 \\ &= \|Rx - Q^T b\|_2^2 + \|\tilde{Q}^T b\|_2^2. \end{aligned}$$

Write $Q = [Q_1, Q_2]$ and $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ conformally with $R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$ so that

$$\|Ax - b\|_2^2 = \|R_{11}x_1 + R_{12}x_2 - Q_1^T b\|_2^2 + \|Q_2^T b\|_2^2 + \|\tilde{Q}^T b\|_2^2$$

is minimized by choosing $x = \begin{bmatrix} R_{11}^{-1}(Q_1^T b - R_{12}x_2) \\ x_2 \end{bmatrix}$ for any x_2. Note that the choice $x_2 = 0$ does not necessarily minimize $\|x\|_2$, but it is a reasonable choice, especially if R_{11} is well-conditioned and $R_{11}^{-1}R_{12}$ is small.

Unfortunately, this method is not reliable since R may be nearly rank deficient even if no R_{22} is small. For example, the n-by-n bidiagonal matrix

$$A = \begin{bmatrix} \frac{1}{2} & 1 & & \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ & & & \frac{1}{2} \end{bmatrix}$$

has $\sigma_{\min}(A) \approx 2^{-n}$, but $A = Q \cdot R$ with $Q = I$ and $R = A$, and no R_{22} is small.

To deal with this failure to recognize rank deficiency, we may do *QR with column pivoting.* This means that we factorize $AP = QR$, P being a permutation matrix. This idea is that at step i (which ranges from 1 to n, the number of columns) we select from the unfinished part of A (columns i to n and rows i to m) the column of largest norm and exchange it with the ith column. We then proceed to compute the usual Householder transformation to zero out column i in entries $i+1$ to m. This pivoting strategy attempts to keep R_{11} as well-conditioned as possible and R_{22} as small as possible.

EXAMPLE 3.9. If we compute the QR decomposition with column pivoting to the last example (.5 on the diagonal and 1 on the superdiagonal) with $n = 11$, we get $R_{11,11} = 4.23 \cdot 10^{-4}$, a reasonable approximation to $\sigma_{\min}(A) = 3.66 \cdot 10^{-4}$. Note that $R_{nn} \geq \sigma_{\min}(A)$ since $\sigma_{\min}(A)$ is the norm of the smallest perturbation that can lower the rank, and setting R_{nn} to 0 lowers rank. $\diamond$

One can show only $\frac{R_{nn}}{\sigma_{\min}(A)} \lesssim 2^n$, but usually R_{nn} is a reasonable approximation to $\sigma_{\min}(A)$. The worst case, however, is as bad as worst-case pivot growth in GEPP.

More sophisticated pivoting schemes than QR with column pivoting, called *rank-revealing* QR algorithms, have been a subject of much recent study. Rank-revealing QR algorithms that detect rank more reliably and sometimes also faster than QR with column pivoting have been developed [28, 30, 48, 50, 109, 126, 128, 150, 196, 236]. We discuss them further in the next section.

QR decomposition with column pivoting is available as subroutine `sgeqpf` in LAPACK. LAPACK also has several similar factorizations available: RQ (`sgerqf`), LQ (`sgelqf`), and QL (`sgeqlf`). Future LAPACK releases will contain improved versions of QR.

3.6. Performance Comparison of Methods for Solving Least Squares Problems

What is the fastest algorithm for solving dense least squares problems? As discussed in section 3.2, solving the normal equations is fastest, followed by QR and the SVD. If A is quite well-conditioned, then the normal equations are about as accurate as the other methods, so even though the normal equations are not numerically stable, they may be used as well. When A is not well-conditioned but far from rank deficient, we should use QR.

Since the design of fast algorithms for rank-deficient least squares problems is a current research area, it is difficult to recommend a single algorithm to use. We summarize a recent study [206] that compared the performance of several algorithms, comparing them to the fastest stable algorithm for the non–rank-deficient case: QR without pivoting, implemented using Householder transformations as described in section 3.4.1, with memory hierarchy optimizations

described in Question 3.17. These comparisons were made in double precision arithmetic on an IBM RS6000/590. Included in the comparison were the rank-revealing QR algorithms mentioned in section 3.5.2 and various implementations of the SVD (see section 5.4). Matrices of various sizes and with various singular value distributions were tested. We present results for two singular value distributions:

Type 1: random matrices, where each entry is uniformly distributed from -1 to 1;

Type 2: matrices with singular values distributed geometrically from 1 to ε (in other words, the ith singular value is γ^i, where γ is chosen so that $\gamma^n = \varepsilon$).

Type 1 matrices are generally well-conditioned, and Type 2 matrices are rank-deficient. We tested small square matrices ($n = m = 20$) and large square matrices ($m = n = 1600$). We tested square matrices because if m is sufficiently greater than n in the m-by-n matrix A, it is cheaper to do a QR decomposition as a "preprocessing step" and then perform rank-revealing QR or the SVD on R. (This is done in LAPACK.) If $m \gg n$, then the initial QR decomposition dominates the the cost of the subsequent operations on the n-by-n matrix R, and all the algorithms cost about the same.

The fastest version of rank-revealing QR was that of [30, 196]. On Type 1 matrices, this algorithm ranged from 3.2 times slower than QR without pivoting for $n = m = 20$ to just 1.1 times slower for $n = m = 1600$. On Type 2 matrices, it ranged from 2.3 times slower (for $n = m = 20$) to 1.2 times slower (for $n = m = 1600$). In contrast, the current LAPACK algorithm, `dgeqpf`, was 2 times to 2.5 times slower for both matrix types.

The fastest version of the SVD was the one in [58], although one based on divide-and-conquer (see section 5.3.3) was about equally fast for $n = m = 1600$. (The one based on divide-and-conquer also used much less memory.) For Type 1 matrices, the SVD algorithm was 7.8 times slower (for $n = m = 20$) to 3.3 times slower (for $n = m = 1600$). For Type 2 matrices, the SVD algorithm was 3.5 times slower (for $n = m = 20$) to 3.0 times slower (for $n = m = 1600$). In contrast, the current LAPACK algorithm, `dgelss`, ranged from 4 times slower (for Type 2 matrices with $n = m = 20$) to 97 times slower (for Type 1 matrices with $n = m = 1600$). This enormous slowdown is apparently due to memory hierarchy effects.

Thus, we see that there is a tradeoff between reliability and speed in solving rank-deficient least squares problems: QR without pivoting is fastest but least reliable, the SVD is slowest but most reliable, and rank-revealing QR is in-between. If $m \gg n$, all algorithms cost about the same. The choice of algorithm depends on the relative importance of speed and reliability to the user.

Future LAPACK releases will contain improved versions of both rank-revealing QR and SVD algorithms for the least squares problem.

3.7. References and Other Topics for Chapter 3

The best recent reference on least squares problems is [33], which also discusses variations on the basic problem discussed here (such as constrained, weighted, and updating least squares), different ways to regularize rank-deficient problems, and software for sparse least squares problems. See also chapter 5 of [121] and [168]. Perturbation theory and error bounds for the least squares solution are discussed in detail in [149]. Rank-revealing QR decompositions are discussed in [28, 30, 48, 50, 126, 150, 196, 206, 236]. In particular, these papers examine the tradeoff between cost and accuracy in rank determination, and in [206] there is a comprehensive performance comparison of the available methods for rank-deficient least squares problems.

3.8. Questions for Chapter 3

QUESTION 3.1. *(Easy)* Show that the two variations of Algorithm 3.1, CGS and MGS, are mathematically equivalent by showing that the two formulas for r_{ji} yield the same results in exact arithmetic.

QUESTION 3.2. *(Easy)* This question will illustrate the difference in numerical stability among three algorithms for computing the QR factorization of a matrix: Householder QR (Algorithm 3.2), CGS (Algorithm 3.1), and MGS (Algorithm 3.1). Obtain the Matlab program QRStability.m from HOMEPAGE/Matlab/QRStability.m. This program generates random matrices with user-specified dimensions `m` and `n` and condition number `cnd`, computes their QR decomposition using the three algorithms, and measures the accuracy of the results. It does this with the *residual* $\|A - Q \cdot R\|/\|A\|$, which should be around machine epsilon ε for a stable algorithm, and the *orthogonality of* Q $\|Q^T \cdot Q - I\|$, which should also be around ε. Run this program for small matrix dimensions (such as `m`= 6 and `n`= 4), modest numbers of random matrices (`samples`= 20), and condition numbers ranging from `cnd`= 1 up to `cnd`= 10^{15}. Describe what you see. Which algorithms are more stable than others? See if you can describe how large $\|Q^T \cdot Q - I\|$ can be as a function of choice of algorithm, `cnd` and ε.

QUESTION 3.3. *(Medium; Hard)* Let A be m-by-n, $m \geq n$, and have full rank.

1. *(Medium)* Show that $\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \cdot \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$ has a solution where x minimizes $\|Ax - b\|_2$. One reason for this formulation is that we can apply iterative refinement to this linear system if we want a more accurate answer (see section 2.5).

2. *(Medium)* What is the condition number of the coefficient matrix in terms of the singular values of A? Hint: Use the SVD of A.

3. *(Medium)* Give an explicit expression for the inverse of the coefficient matrix, as a block 2-by-2 matrix. Hint: Use 2-by-2 block Gaussian elimination. Where have we previously seen the (2,1) block entry?

4. *(Hard)* Show how to use the QR decomposition of A to implement an iterative refinement algorithm to improve the accuracy of x.

QUESTION 3.4. *(Medium) Weighted least squares:* If some components of $Ax - b$ are more important than others, we can weight them with a scale factor d_i and solve the weighted least squares problem $\min \|D(Ax-b)\|_2$ instead, where D has diagonal entries d_i. More generally, recall that if C is symmetric positive definite, then $\|x\|_C \equiv (x^T C x)^{1/2}$ is a norm, and we can consider minimizing $\|Ax - b\|_C$. Derive the normal equations for this problem, as well as the formulation corresponding to the previous question.

QUESTION 3.5. *(Medium; Z. Bai)* Let $A \in \mathbb{R}^{n\times n}$ be positive definite. Two vectors u_1 and u_2 are called A-orthogonal if $u_1^T A u_2 = 0$. If $U \in \mathbb{R}^{n\times r}$ and $U^T A U = I$, then the columns of U are said to be A-orthonormal. Show that every subspace has an A-orthonormal basis.

QUESTION 3.6. *(Easy; Z. Bai)* Let A have the form

$$A = \begin{bmatrix} R \\ S \end{bmatrix},$$

where R is n-by-n and upper triangular, and S is m-by-n and dense. Describe an algorithm using Householder transformations for reducing A to upper triangular form. Your algorithm should not "fill in" the zeros in R and thus require fewer operations than would Algorithm 3.2 applied to A.

QUESTION 3.7. *(Medium; Z. Bai)* If $A = R + uv^T$, where R is an upper triangular matrix, and u and v are column vectors, describe an efficient algorithm to compute the QR decomposition of A. Hint: Using Givens rotations, your algorithm should take $O(n^2)$ operations. In contrast, Algorithm 3.2 would take $O(n^3)$ operations.

QUESTION 3.8. *(Medium; Z. Bai)* Let $x \in \mathbb{R}^n$ and let P be a Householder matrix such that $Px = \pm\|x\|_2 e_1$. Let $G_{1,2}, \ldots, G_{n-1,n}$ be Givens rotations, and let $Q = G_{1,2} \cdots G_{n-1,n}$. Suppose $Qx = \pm\|x\|_2 e_1$. Must P equal Q? (You need to give a proof or a counterexample.)

QUESTION 3.9. *(Easy; Z. Bai)* Let A be m-by-n, with SVD $A = U\Sigma V^T$. Compute the SVDs of the following matrices in terms of U, Σ, and V:

1. $(A^T A)^{-1}$,

2. $(A^T A)^{-1} A^T$,

3. $A(A^TA)^{-1}$,

4. $A(A^TA)^{-1}A^T$.

QUESTION 3.10. *(Medium; R. Schreiber)* Let A_k be a best rank-k approximation of the matrix A, as defined in Part 9 of Theorem 3.3. Let σ_i be the ith singular value of A. Show that A_k is unique if $\sigma_k > \sigma_{k+1}$.

QUESTION 3.11. *(Easy; Z. Bai)* Let A be m-by-n. Show that $X = A^+$ (the Moore–Penrose pseudoinverse) minimizes $\|AX - I\|_F$ over all n-by-m matrices X. What is the value of this minimum?

QUESTION 3.12. *(Medium; Z. Bai)* Let A, B, and C be matrices with dimensions such that the product A^TCB^T is well defined. Let $\mathcal{X}$ be the set of matrices X minimizing $\|AXB - C\|_F$, and let X_0 be the unique member of $\mathcal{X}$ minimizing $\|X\|_F$. Show that $X_0 = A^+CB^+$. Hint: Use the SVDs of A and B.

QUESTION 3.13. *(Medium; Z. Bai)* Show that the Moore–Penrose pseudoinverse of A satisfies the following identities:

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ A^+A &= (A^+A)^T, \\ AA^+ &= (AA^+)^T. \end{aligned}$$

QUESTION 3.14. *(Medium)* Prove part 4 of Theorem 3.3: Let $H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$, where A is square and $A = U\Sigma V^T$ is its SVD. Let $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$, $U = [u_1, \ldots, u_n]$, and $V = [v_1, \ldots, v_n]$. Prove that the $2n$ eigenvalues of H are $\pm\sigma_i$, with corresponding unit eigenvectors $\frac{1}{\sqrt{2}}\begin{bmatrix} v_i \\ \pm u_i \end{bmatrix}$. Extend to the case of rectangular A.

QUESTION 3.15. *(Medium)* Let A be m-by-n, $m < n$, and of full rank. Then $\min \|Ax - b\|_2$ is called an *underdetermined least squares problem.* Show that the solution is an $(n - m)$-dimensional set. Show how to compute the unique minimum norm solution using appropriately modified normal equations, QR decomposition, and SVD.

QUESTION 3.16. *(Medium)* Prove Lemma 3.1.

QUESTION 3.17. *(Hard)* In section 2.6.3, we showed how to reorganize Gaussian elimination to perform Level 2 BLAS and Level 3 BLAS at each step in order to exploit the higher speed of these operations. In this problem, we will show how to apply a sequence of Householder transformations using Level 2 and Level 3 BLAS.

1. Let $u_1, \ldots, u_b$ be a sequence of vectors of dimension n, where $\|u_i\|_2 = 1$ and the first $i-1$ components of u_i are zero. Let $P = P_b \cdot P_{b-1} \cdots P_1$, where $P_i = I - 2u_i u_i^T$ is a Householder transformation. Show that there is a b-by-b lower triangular matrix T such that $P = I - UTU^T$, where $U = [u_1, \ldots, u_b]$. In particular, provide an algorithm for computing the entries of T. This identity shows that we can replace multiplication by b Householder transformations P_1 through P_b by three matrix multiplications by U, T, and U^T (plus the cost of computing T).

2. Let House(x) be a function of the vector x which returns a unit vector u such that $(I - 2uu^T)x = \|x\|_2 e_1$; we showed how to implement House(x) in section 3.4. Then Algorithm 3.2 for computing the QR decomposition of the m-by-n matrix A may be written as

   for $i = 1 : m$
   $u_i = \text{House}(A(i : m, i))$
   $P_i = I - 2u_i u_i^T$
   $A(i : m, i : n) = P_i A(i : m, i : n)$
   endfor

   Show how to implement this in terms of the Level 2 BLAS in an efficient way (in particular, matrix-vector multiplications and rank-1 updates). What is the floating point operation count? (Just the high-order terms in n and m are enough.) It is sufficient to write a short program in the same notation as above (although trying it in Matlab and comparing with Matlab's own QR factorization are a good way to make sure that you are right!).

3. Using the results of step (1), show how to implement QR decomposition in terms of Level 3 BLAS. What is the operation count? This technique is used to accelerate the QR decomposition, just as we accelerated Gaussian elimination in section 2.6. It is used in the LAPACK routine `sgeqrf`.

QUESTION 3.18. *(Medium)* It is often of interest to solve *constrained least squares problems*, where the solution x must satisfy a linear or nonlinear constraint in addition to minimizing $\|Ax - b\|_2$. We consider one such problem here. Suppose that we want to choose x to minimize $\|Ax - b\|_2$ subject to the linear constraint $Cx = d$. Suppose also that A is m-by-n, C is p-by-n, and C has full rank. We also assume that $p \leq n$ (so $Cx = d$ is guaranteed to

be consistent) and $n \leq m + p$ (so the system is not underdetermined). Show that there is a unique solution under the assumption that $\begin{bmatrix} A \\ C \end{bmatrix}$ has full column rank. Show how to compute x using two QR decompositions and some matrix-vector multiplications and solving some triangular systems of equations. Hint: Look at LAPACK routine `sgglse` and its description in the LAPACK manual [10] (NETLIB/lapack/lug/lapack_lug.html).

QUESTION 3.19. *(Hard; Programming)* Write a program (in Matlab or any other language) to update a geodetic database using least squares, as described in Example 3.3. Take as input a set of "landmarks," their approximate coordinates (x_i, y_i), and a set of new angle measurements θ_j and distance measurements L_{ij}. The output should be corrections $(\delta x_i, \delta y_i)$ for each landmark, an error bound for the corrections, and a picture (triangulation) of the old and new landmarks.

QUESTION 3.20. *(Hard)* Prove Theorem 3.4.

QUESTION 3.21. *(Medium)* Redo Example 3.1, using a rank-deficient least squares technique from section 3.5.1. Does this improve the accuracy of the high-degree approximating polynomials?