Statistics and Computing

James E. Gentle

Numerical Linear Algebra for Applications in Statistics

Springer

Statistics and Computing

Springer Science+Business Media, LLC

Statistics and Computing

Gentle: Numerical Linear Algebra for Applications in Statistics.
Gentle: Random Number Generation and Monte Carlo Methods.
Härdle/Klinke/Turlach: XploRe: An Interactive Statistical Computing Environment.
Krause/Olson: The Basics of S and S-PLUS.
Ó Ruanaidh/Fitzgerald: Numerical Bayesian Methods Applied to Signal Processing.
Pannatier: VARIOWIN: Software for Spatial Data Analysis in 2D.
Venables/Ripley: Modern Applied Statistics with S-PLUS, 2nd edition.
Lange: Numerical Analysis for Statisticians.

James E. Gentle

Numerical Linear Algebra for Applications in Statistics

With 21 Illustrations

Springer

James E. Gentle
Institute for Computational
Sciences and Informatics
George Mason University
Fairfax, VA 22030-4444
USA

Series Editors:

Library of Congress Cataloging-in-Publication Data
Gentle, James E., 1943–
Numerical linear algebra for applications in statistics / James E.
Gentle.
p. cm. – (Statistics and computing)
Includes bibliographical references and indexes.
ISBN 978-1-4612-6842-0 ISBN 978-1-4612-0623-1 (eBook)
DOI 10.1007/978-1-4612-0623-1
1. Algebras, Linear. 2. Linear models (Statistics) I. Title.
II. Series.
QA184.G45 1998
512' .5--DC21 98-3959

Printed on acid-free paper.

Production managed by Lesley Poliner; manufacturing supervised by Joe Quatela.
Camera-ready copy prepared from the author's LaTeX files.

9 8 7 6 5 4 3 2 1

ISBN 978-1-4612-6842-0

To María

Preface

Linear algebra is one of the most important mathematical and computational tools in the sciences. While linear models and linear transformations are important in their own right, the basic role that linear algebra plays in nonlinear models, in optimization, and in other areas of statistics also makes an understanding of linear methods one of the most fundamental requirements for research in statistics or in application of statistical methods.

This book presents the aspects of numerical linear algebra that are important to statisticians. An understanding of numerical linear algebra requires both some basic understanding of computer arithmetic and some knowledge of the basic properties of vectors and matrices, so parts of the first two chapters are devoted to these fundamentals.

There are a number of books that cover linear algebra and matrix theory, with an emphasis on applications in statistics. In this book I state most of the propositions of linear algebra that are useful in data analysis, but in most cases I do not give the details of the proofs. Most of the proofs are rather straightforward and are available in the general references cited.

There are also a number of books on numerical linear algebra for numerical analysts. This book covers the computational aspects of vectors and matrices with an emphasis on statistical applications.

Books on statistical linear models also often address the computational issues; in this book the computations are central.

Throughout this book, the difference between an expression and a computing method is emphasized. For example, often we may write the solution to the linear system $Ax = b$ as $A^{-1}b$. Although this is the solution (so long as A is square and full rank), solving the linear system does not involve computing A^{-1}. We may write $A^{-1}b$, but we know we can compute the solution without inverting the matrix.

Chapter 1 provides some basic information on how data are stored and manipulated in a computer. Some of this material is rather tedious, but it is important to have a general understanding of computer arithmetic before considering computations for linear algebra. The impatient reader may skip or just skim Chapter 1, but the reader should be aware that the way the computer stores numbers and performs computations has far-reaching consequences. Computer arithmetic differs from ordinary arithmetic in many ways; for exam-

ple, computer arithmetic lacks associativity of addition and multiplication, and series often converge even when they are not supposed to. (On the computer, a straightforward evaluation of $\sum_{x=1}^{\infty} x$ converges!) Much of Chapter 1 is presented in the spirit of Forsythe (1970), "Pitfalls in computation, or why a math book isn't enough."

I emphasize the difference in the abstract number systems called the reals, $\mathbb{R}$, and the integers, $\mathbb{Z}$, from the computer number systems $\mathbb{F}$, the floating-point numbers, and $\mathbb{I}$, the fixed-point numbers. (Appendix A provides definitions for the notation.)

Chapter 1 also covers some of the fundamentals of algorithms, such as iterations, recursion, and convergence. It also discusses software development. Software issues are revisited in Chapter 5.

In Chapter 2, before considering numerical linear algebra, I begin with some basic properties of linear algebra. Except for some occasional asides, this material in the lengthy Section 2.1 is not in the area of *numerical* linear algebra. Knowledge of this material, however, is assumed in many places in the rest of the book. This section also includes topics, such as condition numbers, that would not be found in the usual books on "matrix algebra for statisticians". Section 2.1 can be considered as a mini-reference source on vectors and matrices for statisticians.

In Section 2.2, building on the material from Chapter 1, I discuss how vectors and matrices are represented and manipulated in a computer.

Chapters 3 and 4 cover the basic computations for decomposing matrices, solving linear systems, and extracting eigenvalues and eigenvectors. These are the fundamental operations of numerical linear algebra. The need to solve linear systems arises much more often in statistics than does the need for eigenanalysis, and consequently Chapter 3 is longer and more detailed than Chapter 4.

Chapter 5 provides a brief introduction to software available for computations with linear systems. Some specific systems mentioned include the IMSL Libraries for Fortran and C, Matlab, and S-Plus. All of these systems are easy to use, and the best way to learn them is to begin using them for simple problems.

Throughout the text, the methods particularly useful in statistical computations are emphasized; and in Chapter 6, a few applications in statistics are discussed specifically.

Appendix A collects the notation used in this book. It is generally "standard" notation, but one thing the reader must become accustomed to is the lack of notational distinction between a vector and a scalar.

All vectors are "column" vectors, although I may write them as horizontal lists of their elements. (Whether vectors are "row" vectors or "column" vectors is generally only relevant for how we write expressions involving vector/matrix multiplication or partitions of matrices.)

I write algorithms in various ways, sometimes in a form that looks similar to Fortran or C, and sometimes as a list of numbered steps. I believe all of the descriptions used are straightforward and unambiguous.

One of the most significant developments in recent years, along with the general growth of computing power, has been the growth of data. It is now common to search through massive datasets and compute summary statistics from various items that may indicate relationships that were not previously recognized. The individual items or the relationships among them may not have been of primary interest when the data were originally collected. This process of prowling through the data is sometimes called *data mining* or *knowledge discovery in databases* (KDD). The objective is to discover characteristics of the data that may not be expected based on the existing theory.

Data must be stored; it must be transported; it must be sorted, searched, and otherwise rearranged; and computations must be performed on it. The size of the dataset largely determines whether these actions are feasible. For processing such massive datasets, the order of computations is a key measure of feasibility. Massive datasets make seemingly trivial improvements in algorithms important. The speedup of Strassen's method of an $O(n^3)$ algorithm to an $O(n^{2.81})$ algorithm, for example, becomes relevant for very large datasets. (Strassen's method is described on page 83.)

We now are beginning to encounter datasets of size 10^{10} and larger. We can quickly determine that a process whose computations are $O(n^2)$ cannot be reasonably contemplated for such massive datasets. If computations can be performed at a rate of 10^{12} per second (teraflop), it would take roughly three years to complete the computations. (A rough order of magnitude for quick "year" computations is $\pi \times 10^7$ seconds equals approximately one year.)

Advances in computer hardware continue to expand what is computationally feasible. It is interesting to note, however, that the order of time required for computations are determined by the problem to be solved and the algorithm to be used, not by the capabilities of the hardware. Advances in algorithm design have reduced the order of computations for many standard problems, while advances in hardware have not changed the order of the computations. Hardware advances have changed only the constant in the order of time.

This book has been assembled from lecture notes I have used in various courses in the computational and statistical sciences over the past few years. I believe the topics addressed in this book constitute the most important material for an introductory course in statistical computing, and should be covered in every such course. There are a number of additional topics that could be covered in a course in scientific computing, such as random number generation, optimization, and quadrature and solution of differential equations. Most of these topics require an understanding of computer arithmetic and of numerical linear algebra as covered in this book, so this book could serve as a basic reference for courses on other topics in statistical computing.

This book could also be used as a supplementary reference text for a course in linear regression that emphasizes the computational aspects.

The prerequisites for this text are minimal. Obviously some background in mathematics is necessary. Some background in statistics or data analysis and some level of scientific computer literacy are also required.

I do not use any particular software system in the book; but I do assume the ability to program in either Fortran or C, and the availability of either S-Plus, Matlab, or Maple. For some exercises the required software can be obtained from either `statlib` or `netlib` (see the bibliography).

Some exercises are Monte Carlo studies. I do not discuss Monte Carlo methods in this text, so the reader lacking background in that area may need to consult another reference in order to work those exercises.

I believe examples are very important. When I have taught this material, my lectures have consisted in large part of working through examples. Some of those examples have become exercises in the present text. The exercises should be considered an integral part of the book.

Acknowledgments

Over the years, I have benefited from associations with top-notch statisticians, numerical analysts, and computer scientists. There are far too many to acknowledge individually, but four stand out. My first real mentor — who was a giant in each of these areas — was Hoh Hartley. My education in statistical computing continued with Bill Kennedy, as I began to find my place in the broad field of statistics. During my years of producing software used by people all over the world, my collaborations with Tom Aird helped me better to understand some of the central issues of mathematical software. Finally, during the past several years, my understanding of computational statistics has been honed through my association with Ed Wegman. I thank these four people especially.

I thank my wife María, to whom this book is dedicated, for everything.

I used TEX via LATEX to write the book. I did all of the typing, programming, etc., myself (mostly early in the morning or late at night), so all mistakes are mine.

Material relating to courses I teach in the computational sciences is available over the World Wide Web at the URL,
`http://www.science.gmu.edu/`

Notes on this book, including errata, are available at
`http://www.science.gmu.edu/~jgentle/linbk/`

Notes on a larger book in computational statistics are available at
`http://www.science.gmu.edu/~jgentle/cmpstbk/`

Fairfax County, Virginia

James E. Gentle
June 15, 1998

Contents

Chapter 1

Computer Storage and Manipulation of Data

The computer is a tool for a variety of applications. The statistical applications include storage, manipulation, and presentation of data. The data may be numbers, text, or images. For each type of data, there are several ways of coding that can be used to store the data, and specific ways the data may be manipulated.

How much a computer user needs to know about the way the computer works depends on the complexity of the use and the extent to which the necessary operations of the computer have been encapsulated in software that is oriented toward the specific application. This chapter covers many of the basics of how digital computers represent data and perform operations on the data. Although some of the specific details we discuss will not be important for the computational scientist or for someone doing statistical computing, the consequences of those details are important, and the serious computer user must be at least vaguely aware of the consequences. The fact that multiplying two positive numbers on the computer can yield a negative number should cause anyone who programs a computer to take care.

Data of whatever form is represented by groups of 0's and 1's, called *bits* from the words "binary" and "digits". (The word was coined by John Tukey.) For representing simple text, that is, strings of characters with no special representation, the bits are usually taken in groups of eight, called *bytes*, and associated with a specific character according to a fixed coding rule. Because of the common association of a byte with a character, those two words are often used synonymously.

The most widely used code for representing characters in bytes is "ASCII" (pronounced "askey", from American Standard Code for Information Interchange). Because the code is so widely used, the phrase "ASCII data" is sometimes used as a synonym for text or character data. The ASCII code for the character "A", for example, is 01000001; for "a" is 01100001; and for

"5" is 00110101. Strings of bits are read by humans more easily if grouped into strings of fours; a four-bit string is equivalent to a hexadecimal digit, 1, 2, ... , 9, A, B, ... , or F. Thus, the ASCII codes just shown could be written in hexadecimal notation as 41 ("A"), 61 ("a"), and 35 ("5").

Because the common character sets differ from one language to another (both natural languages and computer languages), there are several modifications of the basic ASCII code set. Also, when there is a need for more different characters than can be represented in a byte (2^8), codes to associate characters with larger groups of bits are necessary. For compatibility with the commonly used ASCII codes using groups of 8 bits, these codes usually are for groups of 16 bits. These codes for "16-bit characters" are useful for representing characters in some Oriental languages, for example. The Unicode Consortium (1990, 1992) has developed a 16-bit standard, called Unicode, that is becoming widely used for representing characters from a variety of languages. For any ASCII character, the Unicode representation uses eight leading 0's and then the same eight bits as the ASCII representation.

A standard scheme of representing data is very important when data are moved from one computer system to another. Except for some bits that indicate how other bits are to be formed into groups (such as an indicator of the end of a file, or a record within a file), a set of data in ASCII representation would be the same on different computer systems. The Java system uses Unicode for representing characters so as to insure that documents can be shared among widely disparate systems.

In addition to standard schemes for representing the individual data elements, there are some standard formats for organizing and storing sets of data. Although most of these formats are defined by commercial software vendors, two that are open and may become more commonly used are the Common Data Format (CDF), developed by the National Space Science Data Center, and the Hierarchical Data Format (HDF), developed by the National Center for Supercomputing Applications. Both standards allow a variety of types and structures of data; the standardization is in the descriptions that accompany the datasets. More information about these can be obtained from the Web sites. The top-level Web sites at both organizations are stable, and provide links to the Web pages describing the formats. The National Space Science Data Center is a part of NASA and can be reached from

```
http://www.nasa.gov
```

and the National Center for Supercomputing Applications can be reached from

```
http://www.ncsa.uiuc.edu
```

Types of Data

Bytes that correspond to characters are often concatenated to form *character string data* (or just "strings"). Strings represent text without regard to the appearance of the text if it were to be printed. Thus, a string representing "ABC" does not distinguish between "ABC", "*ABC* ", and "**ABC**". The appearance of the printed character must be indicated some other way, perhaps by additional bit strings designating a font.

The appearance of characters or of other visual entities such as graphs or pictures is often represented more directly as a "bitmap". Images on a display medium such as paper or a CRT screen consist of an arrangement of small dots, possibly of various colors. The dots must be coded into a sequence of bits, and there are various coding schemes in use, such as *gif* (graphical interchange file) or *wmf* (windows meta file). Image representations of "ABC", "*ABC* ", and "**ABC**" would all be different. In each case, the data would be represented as a set of dots located with respect to some coordinate system. More dots would be turned on to represent "**ABC**" than to represent "ABC". The location of the dots and the distance between the dots depend on the coordinate system; thus the image can be repositioned or rescaled.

Computers initially were used primarily to process numeric data, and numbers are still the most important type of data in statistical computing. There are important differences between the numerical quantities with which the computer works and the numerical quantities of everyday experience. The fact that numbers in the computer must have a finite representation has very important consequences.

1.1 Digital Representation of Numeric Data

For representing a number in a finite number of digits or bits, the two most relevant things are the magnitude of the number and the precision to which the number is to be represented. Whenever a set of numbers is to be used in the same context, we must find a method of representing the numbers that will accommodate their full range and will carry enough precision for all of the numbers in the set.

Another important aspect in the choice of a method to represent data is the way data are communicated within a computer and between the computer and peripheral components such as data storage units. Data are usually treated as a fixed-length sequence of bits. The basic grouping of bits in a computer is sometimes called a "word", or a "storage unit". The lengths of words or storage units commonly used in computers are 32 or 64 bits.

Unlike data represented in ASCII (in which the representation is actually of the characters, which in turn, represent the data themselves), the same numeric data will very often have different representations on different computer systems. It is also necessary to have different kinds of representations for different sets of numbers, even on the same computer. Like the ASCII standard for

characters, however, there are some standards for representation of, and operations on, numeric data. The Institute for Electrical and Electronics Engineers (IEEE) has been active in promulgating these standards, and the standards themselves are designated by an IEEE number.

The two mathematical models that are often used for numeric data are the ring of integers, $\mathbb{Z}$, and the field of reals, $\mathbb{R}$. We use two computer models, $\mathbb{I}$ and $\mathbb{F}$, to simulate these mathematical entities. (Unfortunately, neither $\mathbb{I}$ nor $\mathbb{F}$ is a simple mathematical construct such as a ring or field.)

Representation of Relatively Small Integers: Fixed-Point Representation

Because an important set of numbers is a finite set of reasonably sized integers, efficient schemes for representing these special numbers are available in most computing systems. The scheme is usually some form of a base 2 representation, and may use one storage unit (this is most common), two storage units, or one half of a storage unit. For example, if a storage unit consists of 32 bits and one storage unit is used to represent an integer, the integer 5 may be represented as in binary notation using the low-order bits, as shown in Figure 1.1.

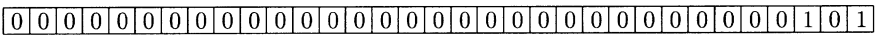


Figure 1.1: The Value 5 in a Binary Representation

The sequence of bits in Figure 1.1 represents the value 5; the ASCII code shown previously, 00110101 or 35 in hexadecimal, represents the character "5".

If the set of integers includes the negative numbers also, some way of indicating the sign must be available. The first bit in the bit sequence (usually one storage unit) representing an integer is usually used to indicate the sign; if it is 0, a positive number is represented; if it is 1, a negative number. In a common method for representing negative integers, called "twos-complement representation", the sign bit is set to 1, and the remaining bits are set to their opposite values (0 for 1; 1 for 0) and then 1 is added to the result. If the bits for 5 are ...00101, the bits for -5 would be ...11010 + 1, or ...11011. If there are k bits in a storage unit (and one storage unit is used to represent a single integer), the integers from 0 through $2^{k-1} - 1$ would be represented in ordinary binary notation using $k - 1$ bits. An integer i in the interval $[-2^{k-1}, -1]$ would be represented by the same bit pattern by which the nonnegative integer $2^{k-1} - i$ is represented, except the sign bit would be 1.

The sequence of bits in Figure 1.2 represents the value -5 using twos-complement notation in 32 bits, with the leftmost bit being the sign bit, and the rightmost bit being the least significant bit, that is, the 1's position. The ASCII code for "-5" consists of the codes for "$-$" and "5", that is,
00101101 00110101.

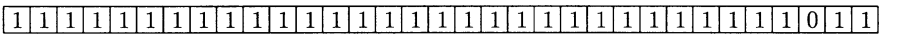


Figure 1.2: The Value -5 in a Twos-Complement Representation

The special representations for numeric data are usually chosen so as to facilitate manipulation of data. The twos-complement representation makes arithmetic operations particularly simple.

It is easy to see that the largest integer that can be represented in the twos-complement form is $2^{k-1} - 1$, and the smallest integer is -2^{k-1}.

A representation scheme such as that described above is called *fixed-point* representation or *integer* representation, and the set of such numbers is denoted by $\mathbb{I}$. The notation $\mathbb{I}$ is also used to denote the system built on this set. This system is similar in some ways to a field instead of a ring, which is what the integers $\mathbb{Z}$ are.

There are several variations of the fixed-point representation. The number of bits used and the method of representing negative numbers are two aspects that generally vary from one computer to another. Even within a single computer system, the number of bits used in fixed-point representation may vary; it is typically one storage unit or a half of a storage unit.

Representation of Larger Numbers and Nonintegral Numbers: Floating-Point Representation

In a fixed-point representation all bits represent values greater than or equal to 1; the *base point* or *radix point* is at the far right, before the first bit. In a fixed-point representation scheme using k bits, the range of representable numbers is of the order of 2^k, usually from approximately -2^{k-1} to 2^{k-1}. Numbers outside of this range cannot be represented directly in the fixed-point scheme. Likewise, nonintegral numbers cannot be represented. Large numbers and fractional numbers are generally represented in a scheme similar to what is sometimes called "scientific notation", or in a type of logarithmic notation. Because within a fixed number of digits, the radix point is not fixed, this scheme is called *floating-point* representation, and the set of such numbers is denoted by $\mathbb{F}$. The notation $\mathbb{F}$ is also used to denote the system built on this set.

In a misplaced analogy to the real numbers, a floating-point number is also called "real". Both computer "integers", $\mathbb{I}$, and "reals", $\mathbb{F}$, represent useful subsets of the corresponding mathematical entities, $\mathbb{Z}$ and $\mathbb{R}$; but while the computer numbers called "integers" do constitute a fairly simple subset of the integers, the computer numbers called "real" do not correspond to the real numbers in a natural way. In particular, the floating-point numbers do not occur uniformly over the real number line.

Within the allowable range, a mathematical integer is exactly represented by a computer fixed-point number; but a given real number, even a rational,

of any size may or may not have an exact representation by a floating-point number. This is the familiar situation of fractions such as $\frac{1}{3}$ not having a finite representation in base 10. The simple rule, of course, is that the number must be a rational number whose denominator in reduced form factors into only primes that appear in the factorization of the base. In base 10, for example, only rational numbers whose factored denominators contain only 2's and 5's have an exact, finite representation; and in base 2, only rational numbers whose factored denominators contain only 2's have an exact, finite representation.

For a given real number x, we will occasionally use the notation

$$[x]_c$$

to indicate the floating-point number used to approximate x, and we will refer to the exact value of a floating-point number as a *computer number*. We will also use the phrase "computer number" to refer to the value of a computer fixed-point number. It is important to understand that computer numbers are members of proper, finite subsets, $\mathbb{I}$ and $\mathbb{F}$, of the corresponding sets $\mathbb{Z}$ and $\mathbb{R}$.

Our main purpose in using computers, of course, is not to evaluate functions of the set of computer floating-point numbers or of the set of computer integers; the main immediate purpose usually is to perform operations in the field of real (or complex) numbers, or occasionally in the ring of integers. Doing computations on the computer, then, involves use of the sets of computer numbers to simulate the sets of reals or integers.

The Parameters of the Floating-Point Representation

The parameters necessary to define a floating-point representation are the *base* or *radix*, the range of the *mantissa* or *significand*, and the range of the *exponent*. Because the number is to be represented in a fixed number of bits, such as one storage unit or word, the ranges of the significand and exponent must be chosen judiciously so as to fit within the number of bits available. If the radix is b, and the integer digits d_i are such that $0 \leq d_i < b$, and there are enough bits in the significand to represent p digits, then a real number is approximated by

$$\pm 0.d_1 d_2 \cdots d_p \times b^e, \tag{1.1}$$

where e is an integer. This is the standard model for the floating-point representation. (The d_i are called "digits" from the common use of base 10.)

The number of bits allocated to the exponent e must be sufficient to represent numbers within a reasonable range of magnitudes; that is, so that the smallest number in magnitude that may be of interest is approximately $b^{e_{\min}}$, and the largest number of interest is approximately $b^{e_{\max}}$, where $e_{\min}$ and $e_{\max}$ are, respectively, the smallest and the largest allowable values of the exponent. Because $e_{\min}$ is likely negative and $e_{\max}$ is positive, the exponent requires a sign. In practice, most computer systems handle the sign of the exponent by

defining $-e_{\min}$ to be a *bias*, and then subtracting the bias from the value of the exponent evaluated without regard to a sign.

The parameters b, p, and $e_{\min}$ and $e_{\max}$ are so fundamental to the operations of the computer that on most computers they are fixed, except for a choice of two or three values for p, and maybe two choices for the range of e.

In order to insure a unique representation for all numbers (except 0), most floating-point systems require that the leading digit in the significand be nonzero, unless the magnitude is less than $b^{e_{\min}}$. A number with a nonzero leading digit in the significand is said to be *normalized*.

The most common value of the base b is 2, although 16 and even 10 are sometimes used. If the base is 2, in a normalized representation, the first digit in the significand is always 1; therefore, it is not necessary to fill that bit position, and so we effectively have an extra bit in the significand. The leading bit, which is not represented, is called a "hidden bit". This requires a special representation for the number 0, however.

In a typical computer using a base of 2 and 64 bits to represent one floating-point number, 1 bit may be designated as the sign bit, 52 bits may be allocated to the significand, and 11 bits allocated to the exponent. The arrangement of these bits is somewhat arbitrary, and of course, the physical arrangement on some kind of storage medium would be different from the "logical" arrangement. A common logical arrangement would assign the first bit as the sign bit, the next 11 bits as the exponent, and the last 52 bits as the significand. (Computer engineers sometimes label these bits as $0, 1, \ldots$, and then get confused as to which is the i^{th} bit. When we say "first", we mean "first", whether an engineer calls it the "0^{th}" or the "1^{st}".) The range of exponents for the base of 2 in this typical computer would be 2,048. If this range is split evenly between positive and negative values, the range of orders of magnitude of representable numbers would be from -308 to 308. The bits allocated to the significand would provide roughly 16 decimal places of precision.

Figure 1.3 shows the bit pattern to represent the number 5, using $b = 2$, $p = 24$, $e_{\min} = -126$, and a bias of 127, in a word of 32 bits. The first bit on the left is the sign bit, the next 8 bits represent the exponent, 129, in ordinary base 2 with a bias, and the remaining 23 bits represent the significand beyond the leading bit, known to be 1. (The binary point is to the right of the leading bit that is not represented.) The value is therefore $+1.01 \times 2^2$ in binary notation.

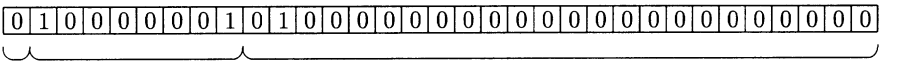


Figure 1.3: The Value 5 in a Floating-Point Representation

While in fixed-point twos-complement representations there are considerable differences between the representation of a given integer and the negative of that integer (see Figures 1.1 and 1.2), the only difference in the floating-point representation of a number and of its additive inverse is usually just in one bit.

In the example of Figure 1.3, only the first bit would be changed to represent the number -5.

As mentioned above, the set of floating-point numbers is not uniformly distributed over the ordered set of the reals. There are the same number of floating-point numbers in the interval $[b^i, b^{i+1}]$ as in the interval $[b^{i+1}, b^{i+2}]$, even though the second interval is b times as long as the first. Figures 1.4 through 1.6 illustrate this. The fixed-point numbers, on the other hand, are uniformly distributed over their range, as illustrated in Figure 1.7.

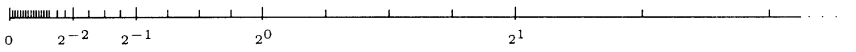


Figure 1.4: The Floating-Point Number Line, Nonnegative Half

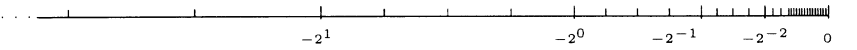


Figure 1.5: The Floating-Point Number Line, Nonpositive Half

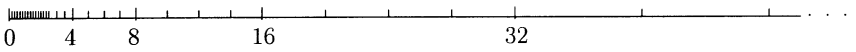


Figure 1.6: The Floating-Point Number Line, Nonnegative Half; Another View

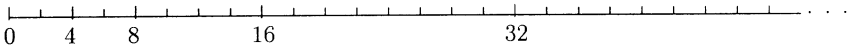


Figure 1.7: The Fixed-Point Number Line, Nonnegative Half

The density of the floating-point numbers is generally greater closer to zero. Notice that if floating-point numbers are all normalized, the spacing between 0 and $b^{e_{\min}}$ is $b^{e_{\min}}$ (that is, there is no floating-point number in that open interval), whereas the spacing between $b^{e_{\min}}$ and $b^{e_{\min}+1}$ is $b^{e_{\min}-p+1}$. Most systems do not require floating-point numbers less than $b^{e_{\min}}$ in magnitude to be normalized. This means that the spacing between 0 and $b^{e_{\min}}$ can be $b^{e_{\min}-p}$, which is more consistent with the spacing just above $b^{e_{\min}}$. When these nonnormalized numbers are the result of arithmetic operations, the result is called "graceful" or "gradual" underflow.

The spacing between floating-point numbers has some interesting (and, for the novice computer user, surprising!) consequences. For example, if 1 is repeatedly added to x, by the recursion

$$x^{(k+1)} = x^{(k)} + 1,$$

the resulting quantity does not continue to get larger. Obviously, it could not increase without bound, because of the finite representation. It does not even approach the largest number representable, however! (This is assuming that the parameters of the floating-point representation are reasonable ones.) In fact, if x is initially smaller in absolute value than $b^{e_{\max}-p}$ (approximately), the recursion

$$x^{(k+1)} = x^{(k)} + c$$

will converge to a stationary point for any value of c smaller in absolute value than $b^{e_{\max}-p}$.

The way the arithmetic is performed would determine these values precisely; as we shall see below, arithmetic operations may utilize more bits than are used in the representation of the individual operands.

The spacings of numbers just smaller than 1 and just larger than 1 are particularly interesting. This is because we can determine the *relative spacing* at any point by knowing the spacing around 1. These spacings at 1 are sometimes called the "machine epsilons", denoted $\epsilon_{\min}$ and $\epsilon_{\max}$ (not to be confused with $e_{\min}$ and $e_{\max}$). It is easy to see from the model for floating-point numbers on page 6 that

$$\epsilon_{\min} = b^{-p}$$

and

$$\epsilon_{\max} = b^{1-p}$$

The more conservative value, $\epsilon_{\max}$, sometimes called "the machine epsilon", ϵ or ϵ_{mach}, provides an upper bound on the rounding that occurs when a floating-point number is chosen to represent a real number. A floating-point number near 1 can be chosen within $\epsilon_{\max}/2$ of a real number that is near 1. This bound, $\frac{1}{2}b^{1-p}$, is called the *unit roundoff*.

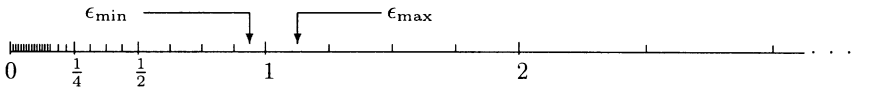


Figure 1.8: Relative Spacings at 1: "Machine Epsilons"

These machine epsilons are also called the "smallest relative spacing" and the "largest relative spacing" because they can be used to determine the relative spacing at the point x. If x is not zero, the relative spacing at x is approximately

$$\frac{x - (1 - \epsilon_{\min})x}{x}$$

or

$$\frac{(1 + \epsilon_{\max})x - x}{x}.$$

Notice we say "approximately". First of all, we do not even know that x is representable. Although $(1 - \epsilon_{\min})$ and $(1 + \epsilon_{\max})$ are members of the set of

floating-point numbers by definition, that does not guarantee that the product of either of these numbers and $[x]_c$ is also a member of the set of floating-point numbers. However, the quantities $[(1-\epsilon_{\min})[x]_c]_c$ and $[(1+\epsilon_{\max})[x]_c]_c$ are representable (by definition of $[\cdot]_c$ as a floating point number approximating the quantity within the brackets); and, in fact, they are respectively the next smallest number than $[x]_c$ (if $[x]_c$ is positive, or the next largest number otherwise), and the next largest number than $[x]_c$ (if $[x]_c$ is positive). The spacings at $[x]_c$ therefore are

$$[x]_c - [(1-\epsilon_{\min})[x]_c]_c$$

and

$$[(1+\epsilon_{\max})[x]_c - [x]_c]_c.$$

As an aside, note that this implies it is probable that

$$[(1-\epsilon_{\min})[x]_c]_c = [(1+\epsilon_{\min})[x]_c]_c.$$

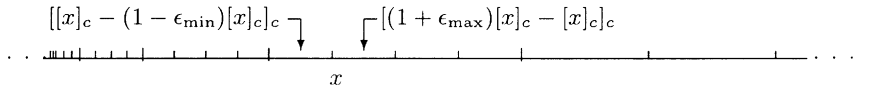


Figure 1.9: Relative Spacings

In practice, to compare two numbers x and y, we must compare $[x]_c$ and $[y]_c$. We consider x and y different if

$$[|y|]_c < [|x|]_c - [(1-\epsilon_{\min})[|x|]_c]_c,$$

or if

$$[|y|]_c > [|x|]_c + [(1+\epsilon_{\max})[|x|]_c]_c.$$

The relative spacing at any point obviously depends on the value represented by the least significant digit in the significand. This digit (or bit) is called the "unit in the last place", or "ulp". The magnitude of an ulp depends of course on the magnitude of the number being represented. Any real number within the range allowed by the exponent can be approximated within $\frac{1}{2}$ ulp by a floating-point number.

The subsets of numbers that we need in the computer depend on the kinds of numbers that are of interest for the problem at hand. Often, however, the kinds of numbers of interest change dramatically within a given problem. For example, we may begin with integer data in the range from 1 to 50. Most simple operations such as addition, squaring, and so on, with these data would allow a single paradigm for their representation. The fixed-point representation should work very nicely for such manipulations.

Something as simple as a factorial, however, immediately changes the paradigm. It is unlikely that the fixed-point representation would be able to handle

the resulting large numbers. When we significantly change the range of numbers that must be accommodated, another change that occurs is the ability to represent the numbers exactly. If the beginning data are integers between 1 and 50, and no divisions or operations leading to irrational numbers are performed, one storage unit would almost surely be sufficient to represent all values exactly. If factorials are evaluated, however, the results cannot be represented exactly in one storage unit and so must be approximated (even though the results are integers). When data are not integers, it is usually obvious that we must use approximations, but it may also be true for integer data.

As we have indicated, different computers represent numeric data in different ways. There has been some attempt to provide standards, at least in the range representable and in the precision for floating point quantities. There are two IEEE standards that specify characteristics of floating-point numbers (IEEE, 1985). The IEEE Standard 754 (sometimes called the "binary standard") specifies the exact layout of the bits for two different precisions, "single" and "double". In both cases, the standard requires that the radix be 2. For single precision, p must be 24, $e_{\max}$ must be 127, and $e_{\min}$ must be -126. For double precision, p must be 53, $e_{\max}$ must be 1023, and $e_{\min}$ must be -1022.

The IEEE Standard 754 also defines two additional precisions, "single extended" and "double extended". For each of the extended precisions, the standard sets bounds on the precision and exponent ranges, rather than specifying them exactly. The extended precisions have larger exponent ranges and greater precision than the corresponding precision that is not "extended".

The IEEE Standard 854 requires that the radix be either 2 or 10 and defines ranges for floating-point representations. Formerly, the most widely used computers (IBM System 360 and derivatives) used base 16 representation; and some computers still use this base. Additional information about the IEEE Standards for floating-point numbers can be found in Cody (1988a) and Goldberg (1991).

The environmental inquiry program MACHAR by Cody (1988b) can be used to determine the characteristics of a computer's floating-point representation and its arithmetic. The program, which is available in CALGO from `netlib` (see the bibliography), was written in Fortran 77, and has been translated into C.

W. J. Cody and W. Kahan were leaders in the effort to develop standards for computer arithmetic. A majority of the computers developed in the past few years comply with the standards, but it is up to the computer manufacturers to conform voluntarily to these standards. We would hope that the marketplace would penalize the manufacturers who do not conform.

Special Floating-Point Numbers

It is convenient to be able to represent certain special numeric entities, such as infinity or "indeterminate" (0/0), which do not have ordinary representations in any base-digit system. Although 8 bits are available for the exponent in the

single-precision IEEE binary standard, $e_{\max} = 127$ and $e_{\min} = -126$. This means there are two unused possible values for the exponent; likewise, for the double-precision standard there are two unused possible values for the exponent. These extra possible values for the exponent allow us to represent certain special floating-point numbers. An exponent of $e_{\min} - 1$ allows us to handle 0 and the numbers between 0 and $b^{e_{\min}}$ unambiguously even though there is a hidden bit (see the discussion above about normalization and gradual underflow). The special number 0 is represented with an exponent of $e_{\min} - 1$ and a significand of $00\ldots0$.

An exponent of $e_{\max} + 1$ allows us to represent $\pm\infty$ or the indeterminate value. A floating-point number with this exponent and a significand of 0 represents $\pm\infty$ (the sign bit determines the sign, as usual). A floating-point number with this exponent and a nonzero significand represents an indeterminate value such as $\frac{0}{0}$. This value is called "not-a-number", or NaN. In statistical data processing, a NaN is sometimes used to represent a missing value. Because a NaN is indeterminate, if a variable x has a value of NaN, $x \neq x$. Also, because a NaN can be represented in different ways, however, a programmer must be careful in testing for NaNs. Some software systems provide explicit functions for testing for a NaN. The IEEE binary standard recommended that a function `isnan` be provided to test for a NaN.

Language Constructs for Representing Numeric Data

Most general-purpose computer programming languages, such as Fortran and C, provide constructs for the user to specify the type of representation for numeric quantities. These specifications are made in declaration statements that are made at the beginning of some section of the program for which they apply.

The difference between fixed-point and floating-point representations has a conceptual basis that may correspond to the problem being addressed. The differences between other kinds of representations are often not because of conceptual differences; rather, they are the results of increasingly irrelevant limitations of the computer. The reasons there are "short" and "long", or "signed" and "unsigned" representations do not arise from the problem the user wishes to solve; the representations are to allow for more efficient use of computer resources. The wise software designer nowadays eschews the space-saving constructs that apply to only a relatively small proportion of the data. In some applications, however, the short representations of numeric data still have a place.

In C the types of all variables must be specified with a basic declarator, which may be qualified further. For variables containing numeric data, the possible types are shown in Table 1.1.

Exactly what these types mean is not specified by the language, but depends on the specific implementation, which associates each type with some natural type supported by the specific computer. A common storage for a fixed-point variable of type `short int` uses 16 bits, and for type `long int` uses 32 bits.

Basic type	Basic declarator	Fully qualified declarator
fixed-point	`int`	`signed short int`
		`unsigned short int`
		`signed long int`
		`unsigned long int`
floating-point	`float`	
	`double`	`double`
		`long double`

Table 1.1: Numeric Data Types in C

An `unsigned` quantity of either type specifies that no bit is to be used as a sign bit, which effectively doubles the largest representable number. Of course, this is essentially irrelevant for scientific computations, so `unsigned` integers are generally just a nuisance. If neither `short` nor `long` is specified, there is a default interpretation that is implementation-dependent. The default always favors `signed` over `unsigned`. There is a movement toward standardization of the meanings of these types. The American National Standards Institute (ANSI) and its international counterpart, the International Standards Organization (ISO) have specified standard definitions of several programming languages. ANSI (1989) is a specification of the C language. ANSI C requires that `short int` use at least 16 bits, that `long int` use at least 32 bits, and that `long int` is at least as long as `int`, which in turn is at least as long as `short int`. The `long double` type may or may not have more precision and a larger range than the `double` type.

C does not provide a complex data type. This deficiency can be overcome to some extent by means of a user-defined data type. The user must write functions for all the simple arithmetic operations on complex numbers, just as is done for the simple exponentiation for floats. The object-oriented hybrid language built on C, C++, provides the user the ability also to define operator functions, so that the four simple arithmetic operations can be implemented by the operators, "+", "−", "*", and "/". There is no good way of defining an exponentiation operator, however, because the user-defined operators are limited to extended versions of the operators already defined in the language.

In Fortran variables have a default numeric type that depends on the first letter in the name of the variable. The type can be explicitly declared also. The `signed` and `unsigned` qualifiers of C, which have very little use in scientific computing, are missing in Fortran. Fortran has a fixed-point type that corresponds to integers, and two floating-point types that correspond to reals and to complex numbers. For one standard version of Fortran, called Fortran 77, the possible types for variables containing numeric data are shown in Table 1.2.

Basic type	Basic declarator	Default variable name
fixed-point	`integer`	begin with i - n or I - N
floating-point	`real`	begin with a - h or o - z or with A - H or O - Z
	`double precision`	no default, although d or D is sometimes used
complex	`complex`	no default, although c or C is sometimes used

Table 1.2: Numeric Data Types in Fortran 77

Although the standards organizations have defined these constructs for the Fortran 77 language (ANSI, 1978), just as is the case with C, exactly what these types mean is not specified by the language, but depends on the specific implementation. Some extensions to the language allow the number of bytes to use for a type to be specified (e.g., `real*8`) and allow the type `double complex`.

The complex type is not so much a data type as a data structure composed of two floating-point numbers that has associated operations that simulate the operations defined on the field of complex numbers.

The Fortran 90 language supports the same types as Fortran 77, but also provides much more flexibility in selecting the number of bits to use in the representation of any of the basic types. A fundamental concept for the numeric types in Fortran 90 is called "*kind*". The kind is a qualifier for the basic type; thus a fixed-point number may be an `integer` of kind 1 or of kind 2, for example. The actual value of the qualifier kind may differ from one compiler to another, so the user defines a program parameter to be the kind that is appropriate to the range and precision required for a given variable. Fortran 90 provides the functions `selected_int_kind` and `selected_real_kind` to do this. Thus, to declare some fixed-point variables that have at least 3 decimal digits and some more fixed-point variables that have at least 8 decimal digits, the user may write the following statements

```
integer, parameter  :: little = selected_int_kind(3)
integer, parameter  :: big    = selected_int_kind(8)
integer (little)    :: ismall, jsmall
integer (big)       :: itotal_accounts, igain
```

The variables `little` and `big` would have integer values, chosen by the compiler designer, that could be used in the program to qualify integer types to insure that range of numbers could be handled. Thus, `ismall` and `jsmall` would be fixed-point numbers that could represent integers between -999 and 999, and

`itotal_accounts` and `igain` would be fixed-point numbers that could represent integers between $-99,999,999$ and $99,999,999$. Depending on the basic hardware, the compiler may assign two bytes as kind = `little`, meaning that integers between $-32,768$ and $32,767$ could probably be accommodated by any variable, such as `ismall`, that is declared as `integer (little)`. Likewise, it is probable that the range of variables declared as `integer (big)` could handle numbers in the range $-2,147,483,648$ and $2,147,483,647$. For declaring floating-point numbers, the user can specify a minimum range and precision with the function `selected_real_kind`, which takes two arguments, the number of decimal digits of precision, and the exponent of 10 for the range. Thus, the statements

```
integer, parameter  :: real4 = selected_real_kind(6,37)
integer, parameter  :: real8 = selected_real_kind(15,307)
```

would yield designators of floating-point types that would have either 6 decimals of precision and a range up to 10^{37} or 15 decimals of precision and a range up to 10^{307}. The statements

```
real (real4)    :: x, y
real (real8)    :: dx, dy
```

would declare `x` and `y` as variables corresponding roughly to `real` on most systems, and `dx` and `dy` as variables corresponding roughly to `double precision`.

If the system cannot provide types matching the requirements specified in `selected_int_kind` or `selected_real_kind`, these functions return -1. Because it is not possible to handle such an error situation in the declaration statements, the user should know in advance the available ranges. Fortran 90 provides a number of intrinsic functions, such as `epsilon`, `rrspacing`, and `huge`, to use in obtaining information about the fixed- and floating-point numbers provided by the system.

Fortran 90 also provides a number of intrinsic functions for dealing with bits. These functions are essentially those specified in the MIL-STD-1753 standard of the U.S. Department of Defense. These bit functions, which have been a part of many Fortran implementations for years, provide for shifting bits within a string, extracting bits, exclusive or inclusive oring of bits, and so on. (See ANSI, 1992; Kerrigan, 1993; or Metcalf and Reid, 1990, for more extensive discussions of the types and intrinsic function provided in Fortran 90.)

Many higher-level languages and application software packages do not give the user a choice of the way to represent numeric data. The software system may consistently use a type thought to be appropriate for the kinds of applications addressed. For example, many statistical analysis application packages choose to use a floating-point representation with about 64 bits for all numeric data. Making a choice such as this yields more comparable results across a range of computer platforms on which the software system may be implemented.

Whenever the user chooses the type and precision of variables it is a good idea to use some convention to name the variable in such a way as to indicate the type and precision. Books or courses on elementary programming suggest use of mnemonic names, such as "`time`" for a variable that holds the measure of time. If the variable takes fixed-point values, a better name might be "`itime`". It still has the mnemonic value of "time", but it also helps us to remember that, in the computer, `itime/length` may not be the same thing as `time/xlength`.

Even as we "humanize" computing, we must remember that there are details about the computer that matter. (The operator "/" is said to be "overloaded": in a general way, it means "divide", but it means different things depending on the contexts of the two expressions above.) Whether a quantity is a member of $\mathbb{I}$ or $\mathbb{F}$ may have major consequences for the computations, and a careful choice of notation can help to remind us of that.

Numerical analysts sometimes use the phrase "full precision" to refer to a precision of about 16 decimal digits, and the phrase "half precision" to refer to a precision of about 7 decimal digits. These terms are not defined precisely, but they do allow us to speak of the precision in roughly equivalent ways for different computer systems without specifying the precision exactly. Full precision is roughly equivalent to Fortran `double precision` on the common 32-bit workstations and to Fortran `real` on "supercomputer" machines such as Cray computers. Half precision corresponds roughly to Fortran `real` on the common 32-bit workstations. Full and half precision can be handled in a portable way in Fortran 90. The following statements would declare a variable `x` to be one with full precision:

```
integer, parameter  :: full = selected\_real\_kind(15,307)
real (full)         :: x
```

In a construct of this kind, the user can define "full" or "half" as appropriate.

Other Variations in the Representation of Data; Portability of Data

As we have indicated already, computer designers have a great deal of latitude in how they choose to represent data. The ASCII standards of ANSI and ISO have provided a common representation for individual characters. The IEEE standard 754 referred to previously (IEEE, 1985) has brought some standardization to the representation of floating-point data, but does not specify how the available bits are to be allocated among the sign, exponent, and significand.

Because the number of bits used as the basic storage unit has generally increased over time, some computer designers have arranged small groups of bits, such as bytes, together in strange ways to form words. There are two common schemes of organizing bits into bytes and bytes into words. In one scheme, called "big end" or "big endian", the bits are indexed from the "left", or most significant end of the byte; and bytes are indexed within words and words are indexed within groups of words in the same direction.

In another scheme, called "little end" or "little endian", the bytes are indexed within the word in the opposite direction. Figures 1.10 through 1.13 illustrate some of the differences.

```
character a
character*4 b
integer i, j
equivalence (b,i), (a,j)
print '(10x, a7 , a8)', ' Bits  ', '   Value'
a = 'a'
print '(1x, a10, z2, 7x, a1)',  'a:         ', a, a
print '(1x, a10, z8, 1x, i12)', 'j (=a):    ', j, j
b = 'abcd'
print '(1x, a10, z8, 1x, a4)',  'b:         ', b, b
print '(1x, a10, z8, 1x, i12)', 'i (=b):    ', i, i
end
```

Figure 1.10: A Fortran Program Illustrating Bit and Byte Organization

```
           Bits      Value
a:         61        a
j (=a):          61            97
b:         64636261  abcd
i (=b):    64636261    1684234849
```

Figure 1.11: Output from a Little Endian System (DEC VAX; Unix, VMS)

These differences are important only when accessing the individual bits and bytes, when making data type transformations directly, or when moving data from one machine to another without interpreting the data in the process ("binary transfer"). One lesson to be learned from observing such subtle differences in the way the same quantities are treated in different computer systems is that programs should rarely rely on the inner workings of the computer. A program that does will not be *portable*; that is, it will not give the same results on different computer systems. Programs that are not portable may work well on one system, and the developers of the programs may never intend for them to be used anywhere else. As time passes, however, systems change or users change systems. When that happens, the programs that were not portable may cost more than they ever saved by making use of computer-specific features.

```
          Bits     Value
a:        61       a
j (=a):   00000061          97
b:        61626364 abcd
i (=b):   64636261   1684234849
```

Figure 1.12: Output from a Little Endian System (Intel x86, Pentium, etc.; DOS, Windows 95/8)

```
          Bits     Value
a:        61       a
j (=a):   61000000   1627389952
b:        61626364 abcd
i (=b):   61626364   1633837924
```

Figure 1.13: Output from a Big Endian System (Sun SPARC, Silicon Graphics MIPS, etc.; Unix)

1.2 Computer Operations on Numeric Data

As we have emphasized above, the numerical quantities represented in the computer are used to simulate or approximate more interesting quantities, namely the real numbers or perhaps the integers. Obviously, because the sets (computer numbers and real numbers) are not the same, we could not define operations on the computer numbers that would yield the same field as the familiar field of the reals. In fact, because of the nonuniform spacing of floating-point numbers, we would suspect that some of the fundamental properties of a field may not hold. Depending on the magnitudes of the quantities involved, it is possible, for example, that if we compute ab and ac and then $ab + ac$, we may not get the same thing as if we compute $(b + c)$ and then $a(b + c)$. Just as we use the computer quantities to simulate real quantities, we define operations on the computer quantities to simulate the familiar operations on real quantities. Designers of computers attempt to define computer operations so as to correspond closely to operations on real numbers, but we must not lose sight of the fact that the computer uses a different arithmetic system.

The basic objective in numerical computing, of course, is that a computer operation, when applied to computer numbers, yields computer numbers that approximate the number that would be yielded by a certain mathematical operation applied to the numbers approximated by the original computer numbers. Just as we introduced the notation

$$[x]_c$$

on page 6 to denote the computer floating-point number approximation to the real number x, we occasionally use the notation

$$[\circ]_c$$

to refer to a computer operation that simulates the mathematical operation $\circ$. Thus,

$$[+]_c$$

represents an operation similar to addition, but which yields a result in a set of computer numbers. (We use this notation only where necessary for emphasis, however, because it is somewhat awkward to use it consistently.) The failure of the familiar laws of the field of the reals, such as distributive law cited above, can be anticipated by noting that

$$[[a]_c\ [+]_c\ [b]_c]_c \neq [a+b]_c,$$

or by considering the simple example in which all numbers are rounded to one decimal and so $\frac{1}{3} + \frac{1}{3} \neq \frac{2}{3}$ (that is, $.3 + .3 \neq .7$).

The three familiar laws of the field of the reals (commutativity of addition and multiplication, associativity of addition and multiplication, and distribution of multiplication over addition) result in the independence of the order in which operations are performed; the failure of these laws implies that the order of the operations may make a difference. When computer operations are performed sequentially, we can usually define and control the sequence fairly easily. If the computer performs operations in parallel, the resulting differences in the orders in which some operations may be performed can occasionally yield unexpected results.

The computer operations for the two different types of computer numbers are different, and we discuss them separately.

Because the operations are not closed, special notice may need to be taken when the operation would yield a number not in the set. Adding two numbers, for example, may yield a number too large to be represented well by a computer number, either fixed-point or floating-point. When an operation yields such an anomalous result, an *exception* is said to exist.

Fixed-Point Operations

The operations of addition, subtraction, and multiplication for fixed-point numbers are performed in an obvious way that corresponds to the similar operations on the ring of integers. Subtraction is addition of the additive inverse. (In the usual twos-complement representation we described earlier, all fixed-point numbers have additive inverses except -2^{k-1}.) Because there is no multiplicative inverse, however, division is not multiplication by the inverse. The result of division with fixed-point numbers is the result of division with the corresponding real numbers rounded toward zero. This is not considered an exception.

As we indicated above, the set of fixed-point numbers together with addition and multiplication is not the same as the ring of integers, if for no other reason than the set is finite. Under the ordinary definitions of addition and multiplication, the set is not closed under either operation. The computer operations of addition and multiplication, however, are defined so that the set is closed. These operations occur as if there were additional higher-order bits and the sign bit were interpreted as a regular numeric bit. The result is then whatever would be in the standard number of lower-order bits. If the higher-order bits would be necessary, the operation is said to *overflow*. If fixed-point overflow occurs, the result is not correct under the usual interpretation of the operation, so an error situation, or an exception, has occurred. Most computer systems allow this error condition to be detected, but most software systems do not take note of the exception. The result, of course, depends on the specific computer architecture. On many systems, aside from the interpretation of the sign bit, the result is essentially the same as would result from a modular reduction. There are some special-purpose algorithms that actually use this modified modular reduction, although such algorithms would not be portable across different computer systems.

Floating-Point Operations; Errors

As we have seen, real numbers within the allowable range may or may not have an exact floating-point operation, and the computer operations on the computer numbers may or may not yield numbers that represent exactly the real number that would result from mathematical operations on the numbers. If the true result is r, the best we could hope for would be $[r]_c$. As we have mentioned, however, the computer operation may not be exactly the same as the mathematical operation being simulated, and further, there may be several operations involved in arriving at the result. Hence, we expect some error in the result. If the computed value is $\tilde{r}$ (for the true value r), we speak of the *absolute error*,

$$|\tilde{r} - r|,$$

and the *relative error*,

$$\frac{|\tilde{r} - r|}{|r|}$$

(so long as $r \neq 0$). An important objective in numerical computation obviously is to insure that the error in the result is small.

Ideally, the result of an operation on two floating-point numbers would be the same as if the operation were performed exactly on the two operands (considering them to be exact also) and then the result were rounded. Attempting to do this would be very expensive in both computational time and complexity of the software. If care is not taken, however, the relative error can be very large. Consider, for example, a floating-point number system with $b = 2$ and $p = 4$. Suppose we want to add 8 and -7.5. In the floating-point system we

would be faced with the problem:

$$\begin{aligned} 8: &\quad 1.000 \times 2^3 \\ 7.5: &\quad 1.111 \times 2^2 \end{aligned}$$

To make the exponents the same, we have

$$\begin{aligned} 8: &\quad 1.000 \times 2^3 \\ 7.5: &\quad 0.111 \times 2^3 \end{aligned} \quad \text{or} \quad \begin{aligned} 8: &\quad 1.000 \times 2^3 \\ 7.5: &\quad 1.000 \times 2^3 \end{aligned}$$

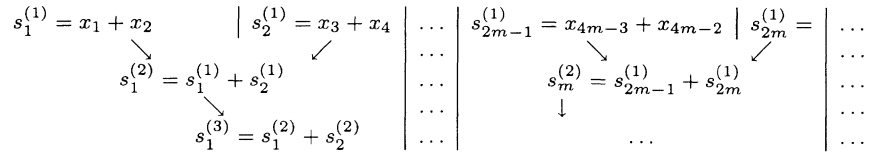
The subtraction will yield either 0.000_2 or $1.000_2 \times 2^0$, whereas the correct value is $1.000_2 \times 2^{-1}$. Either way, the absolute error is 0.5_{10}, and the relative error is 1. Every bit in the significand is wrong. The magnitude of the error is the same as the magnitude of the result. This is not acceptable. (More generally, we could show that the relative error in a similar computation could be as large as $b - 1$, for any base b.) The solution to this problem is to use one or more *guard digits*. A guard digit is an extra digit in the significand that participates in the arithmetic operation. If one guard digit is used (and this is the most common situation), the operands each have $p + 1$ digits in the significand. In the example above, we would have

$$\begin{aligned} 8: &\quad 1.0000 \times 2^3 \\ 7.5: &\quad 0.1111 \times 2^3 \end{aligned}$$

and the result is exact. In general, one guard digit can insure that the relative error is less than $2\epsilon_{\max}$. Use of guard digits requires that the operands be stored in special storage units. Whenever more than one operation is to be performed together, the operands and intermediate results can all be kept in the special registers to take advantage of the guard digits or even longer storage units. This is called chaining of operations.

When several numbers x_i are to be summed, it is likely that as the operations proceed serially, the magnitudes of the partial sum and the next summand will be quite different. In such a case, the full precision of the next summand is lost. This is especially true if the numbers are of the same sign. As we mentioned earlier, a computer program to implement serially the algorithm implied by $\sum_{i=1}^{\infty} i$ will converge to some number much smaller than the largest floating-point number.

If the numbers to be summed are not all the same constant (and if they are constant, just use multiplication!), the accuracy of the summation can be increased by first sorting the numbers and summing them in order of increasing magnitude. If the numbers are all of the same sign and have roughly the same magnitude, a pairwise "fan-in" method may yield good accuracy. In the fan-in method the n numbers to be summed are added two at a time to yield $\lceil n/2 \rceil$ partial sums. The partial sums are then added two at a time, and so on, until all sums are completed. The name "fan-in" comes from the tree diagram of the separate steps of the computations:



It is likely that the numbers to be added will be of roughly the same magnitude at each stage. Remember we are assuming they have the same sign initially; this would be the case, for example, if the summands are squares.

Another way that is even better is due to W. Kahan (see Goldberg, 1991):

$$
\begin{aligned}
&s = x_1 \\
&a = 0 \\
&\text{for } i = 2, \ldots, n \\
&\{ \\
&\quad y = x_i - a \\
&\quad t = s + y \\
&\quad a = (t - s) - y \\
&\quad s = t \\
&\}
\end{aligned}
\tag{1.2}
$$

Another kind of error that can result because of the finite precision used for floating-point numbers is *catastrophic cancellation.* This can occur when two rounded values of approximately equal magnitude and opposite signs are added. (If the values are exact, cancellation can also occur, but it is *benign.*) After catastrophic cancellation, the digits left are just the digits that represented the rounding. Suppose $x \approx y$, and that $[x]_c = [y]_c$. The computed result will be zero, whereas the correct (rounded) result is $[x-y]_c$. The relative error is 100%. This error is caused by rounding, but it is different from the "rounding error" discussed above. Although the loss of information arising from the rounding error is the culprit, the rounding would be of little consequence were it not for the cancellation.

To avoid catastrophic cancellation watch for possible additions of quantities of approximately equal magnitude and opposite signs, and consider rearranging the computations. Consider the problem of computing the roots of a quadratic polynomial, $ax^2 + bx + c$ (see Rice, 1983). In the quadratic formula,

$$
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{1.3}
$$

the square root of the discriminant, $(b^2 - 4ac)$, may be approximately equal to b in magnitude, meaning that one of the roots is close to zero, and, in fact, may be computed as zero. The solution is to compute only one of the roots, x_1, by the formula (the "$-$" root if b is positive, and the "$+$" root if b is negative), and then compute the other root, x_2 by the relationship $x_1 x_2 = c/a$.

The IEEE Binary Standard 754 (IEEE, 1985) applies not only to the representation of floating-point numbers, but also to certain operations on those

numbers. The standard requires correct rounded results for addition, subtraction, multiplication, division, remaindering, and extraction of the square root. It also requires that conversion between fixed-point numbers and floating-point numbers yields correct rounded results.

The standard also defines how exceptions should be handled. The exceptions are divided into five types: overflow, division by zero, underflow, invalid operation, and inexact operation.

If an operation on floating-point numbers would result in a number beyond the range of representable floating-point numbers, the exception, called *overflow*, is generally very serious. (It is serious in fixed-point operations, also, if it is unplanned. Because we have the alternative of using floating-point numbers if the magnitude of the numbers is likely to exceed what is representable in fixed-point, the user is expected to use this alternative. If the magnitude exceeds what is representable in floating-point, however, the user must resort to some indirect means, such as scaling, to solve the problem.)

Division by zero does not cause overflow; it results in a special number if the dividend is nonzero. The result is either ∞ or $-\infty$, which have special representations, as we have seen.

Underflow occurs whenever the result is too small to be represented as a normalized floating-point number. As we have seen, a nonnormalized representation can be used to allow a gradual underflow.

An invalid operation is one for which the result is not defined because of the value of an operand. The invalid operations are addition of ∞ to $-\infty$, multiplication of $\pm\infty$ and 0, 0 divided by 0 or by $\pm\infty$, $\pm\infty$ divided by 0 or by $\pm\infty$, extraction of the square root of a negative number (some systems, such as Fortran, have a special type for complex numbers and deal correctly with them), and remaindering any quantity with 0 or remaindering $\pm\infty$ with any quantity. An invalid operation results in a NaN. Any operation with a NaN also results in a NaN. Some systems distinguish two types of NaN, a "quiet NaN" and a "signaling NaN".

An inexact operation is one for which the result must be rounded. For example, if all p bits of the significand are required to represent both the multiplier and multiplicand, approximately $2p$ bits would be required to represent the product. Because only p are available, however, the result must be rounded.

Exact Computations; Rational Fractions

If the input data can be represented exactly as rational fractions, it may be possible to preserve exact values of the results of computations. Use of rational fractions allows avoidance of reciprocation, which is the operation that most commonly yields a nonrepresentable value from one that is representable. Of course, any addition or multiplication that increases the magnitude of an integer in a rational fraction beyond a value that can be represented exactly (that is, beyond approximately 2^{23}, 2^{31}, or 2^{53}, depending on the computing system),

may break the error-free chain of operations. Exact computations with integers can be carried out using *residue arithmetic*, in which each quantity is as a vector of residues, all from a vector of relatively prime moduli. (See Szabó and Tanaka, 1967, for discussion of the use of residue arithmetic in numerical computations; and see Stallings and Boullion, 1972, and Keller-McNulty and Kennedy, 1986, for applications of this technology in matrix computations.)

Computations with rational fractions are sometimes performed using a fixed-point representation. Gregory and Krishnamurthy (1984) discuss in detail these and other methods for performing error-free computations.

Language Constructs for Operations on Numeric Data

Most general-purpose computer programming languages, such as Fortran and C, provide constructs for operations that correspond to the common operations on scalar numeric data, such as "`+`", "`-`", "`*`" (multiplication), and "`/`". These operators *simulate* the corresponding mathematical operations. As we mentioned on page 19, we will occasionally use a notation such as

$$[+]_c$$

to indicate the computer operator. The operators have slightly different meanings depending on the operand objects; that is, the operations are "*overloaded*". Most of these operators are *binary infix* operators, meaning that the operator is written between the two operands.

Some languages provide operations beyond the four basic scalar arithmetic operations. C provides some specialized operations, such as the unary postfix increment "`++`" and decrement "`--`" operators, for trivial common operations; but does not provide an operator for exponentiation. (Exponentiation is handled by a function provided in a standard supplemental library in C, `<math.h>`.) C also overloads the basic multiplication operator so that it can indicate a change of the meaning of a variable, in addition to indicating the multiplication of two scalar numbers. A standard library in C (`<signal.h>`) allows for easy handling of arithmetic exceptions. With this facility, for example, the user can distinguish a quiet NaN from a signaling NaN.

The C language does not directly provide for operations on special data structures. For operations on complex data, for example, the user must define the type and its operations in a header file (or else, of course, just do the operations as if they were operations on an array of length 2).

Fortran provides the four basic scalar numeric operators, plus an exponentiation operator ("`**`"). (Exactly what this operator means may be slightly different in different versions of Fortran. Some versions interpret the operator always to mean

1. take log
2. multiply by power
3. exponentiate

if the base and the power are both floating-point types. This, of course, would

not work if the base is negative, even if the power is an integer. Most versions of Fortran will determine at run time if the power is an integer, and use repeated multiplication if it is.)

Fortran also provides the usual five operators for complex data (the basic four, plus exponentiation). Fortran 90 provides the same set of scalar numeric operators, plus a basic set of array and vector/matrix operators. The usual vector/matrix operators are implemented as functions, or prefix operators, in Fortran 90.

In addition to the basic arithmetic operators, both Fortran and C, as well as other general programming languages, provide several other types of operators, including relational operators and operators for manipulating structures of data.

Software packages have been built on Fortran and C to extend their accuracy. Two ways in which this is done are by use of *multiple precision* (see Brent, 1978, Smith, 1991, and Bailey, 1993, for example) and by use of *interval arithmetic* (see Yohe, 1979; Kulisch, 1983; and Kulisch and Miranker, 1981 and 1983, for example). Multiple precision operations are performed in the software by combining more than one computer storage unit to represent a single number. Multiple precision is different from "extended precision" discussed earlier; extended precision is implemented at the hardware level or at the microcode level. A multiple precision package may allow the user to specify the number of digits to use in representing data and performing computations. The software packages for symbolic computations, such as Maple, generally provide multiple precision capabilities.

Interval arithmetic maintains intervals in which the exact data and solution are known to lie. Instead of working with single-point approximations, for which we used notation such as

$$[x]_c$$

on page 6 for the value of floating-point approximation to the real number x, and

$$[\circ]_c$$

on page 19 for the simulated operation $\circ$, we can approach the problem by identifying a closed interval in which x lies and a closed interval in which the result of the operation $\circ$ lies. We denote the interval operation as

$$[\circ]_I.$$

For the real number x, we identify two floating-point numbers, x_l and x_u, such that $x_l \leq x \leq x_u$. (This relationship also implies $x_l \leq [x]_c \leq x_u$.) The real number x is then considered to be the interval $[x_l, x_u]$. For this approach to be useful, of course, we seek tight bounds. If $x = [x]_c$, the best interval is degenerate. In other cases either x_l or x_c is $[x]_c$ and the length of the interval is the floating-point spacing from $[x]_c$ in the appropriate direction.

Addition and multiplication in interval arithmetic yields intervals:

$$x\ [+]_I\ y = [x_l + y_l,\ x_u + y_u]$$

and

$$x\ [*]_I\ y = [\min(x_l y_l, x_l y_u, x_u y_l, x_u y_u),\ \max(x_l y_l, x_l y_u, x_u y_l, x_u y_u)].$$

Change of sign results in $[-x_u,\ -x_l]$ and if $0 \notin [x_l,\ x_u]$, reciprocation results in $[1/x_u,\ 1/x_l]$. See Moore (1979) or Alefeld and Herzberger (1983) for an extensive treatment of interval arithmetic. The journal *Reliable Computing* is devoted to interval computations.

The `ACRITH` package of IBM (see Jansen and Weidner, 1986) is a library of Fortran subroutines that perform computations in interval arithmetic and also in extended precision. Kearfott et al. (1994) have produced a portable Fortran library of basic arithmetic operations and elementary functions in interval arithmetic, and Kearfott (1996) gives a Fortran 90 module defining an interval data type.

1.3 Numerical Algorithms and Analysis

The two most important aspects of a computer algorithm are its accuracy and its efficiency. Although each of these concepts appears rather simple on the surface, each is actually fairly complicated, as we shall see.

Error in Numerical Computations

An "accurate" algorithm is one that gets the "right" answer. Knowing that the right answer may not be representable, and rounding within a set of operations may result in variations in the answer, we often must settle for an answer that is "close". As we have discussed previously, we measure error, or closeness, either as the absolute error or the relative error of a computation.

Another way of considering the concept of "closeness" is by looking backward from the computed answer, and asking what perturbation of the original problem would yield the computed answer exactly. This approach, developed by Wilkinson (1963) is called *backward error analysis*. The backward analysis is followed by an assessment of the effect of the perturbation on the solution.

There are other complications in assessing errors. Suppose the answer is a vector, such as a solution to a linear system. What norm do we use to compare closeness of vectors? Another, more complicated, situation for which assessing correctness may be difficult is random number generation. It would be difficult to assign a meaning to "accuracy" for such a problem.

The basic source of error in numerical computations is the inability to work with the reals. The field of reals is simulated with a finite set. This has several consequences. A real number is rounded to a floating-point number; the result of an operation on two floating-point numbers is rounded to another floating-point number; and passage to the limit, which is a fundamental concept in the field of reals, is not possible in the computer.

Rounding errors that occur just because the result of an operation is not representable in the computer's set of floating-point numbers are usually not

too bad. Of course, if they accumulate through the course of many operations, the final result may have an unacceptably large accumulated rounding error.

A natural approach to studying errors in floating-point computations is to define random variables for the rounding at all stages, from the initial representation of the operands through any intermediate computations to the final result. Given a probability model for the rounding error in representation of the input data, a statistical analysis of rounding errors can be performed. Wilkinson (1963) introduced a uniform probability model for rounding of input, and derived distributions for computed results based on that model. Linnainmaa (1975) discusses the effects of accumulated error in floating-point computations based on a more general model of the rounding for the input. This approach leads to a forward error analysis that provides a probability distribution for the error in the final result. (See Bareiss and Barlow, 1980, for an analysis of error in fixed-point computations, which present altogether different problems.)

The obvious probability model for floating-point representations is that the reals within an interval between any two floating-point numbers have a uniform distribution (see Figure 1.4, page 8, and Calvetti, 1991). A probability model for the real line can be built up as a mixture of the uniform distributions (see Exercise 1.9, page 44). The density is obviously 0 in the tails. See Chaitin-Chatelin and Frayssé (1996) for further discussion of probability models for rounding errors. Dempster and Rubin (1983) discuss the application of statistical methods for dealing with grouped data to the data resulting from rounding in floating-point computations.

Another, more pernicious effect of rounding can occur in a single operation, resulting in catastrophic cancellation, as we have discussed previously.

Measures of Error and Bounds for Errors

We have discussed errors in the representation of numbers that are due to the finite precision number system. For the simple case of representing the real number r by an approximation $\tilde{r}$, we defined absolute error, $|\tilde{r} - r|$, and relative error, $|\tilde{r} - r|/|r|$ (so long as $r \neq 0$). These same types of measures are used to express the errors in numerical computations. As we indicated above, however, the result may not be a simple real number; it may consist of several real numbers. For example, in statistical data analysis, the numerical result, $\tilde{r}$, may consist of estimates of several regression coefficients, various sums of squares and their ratio, and several other quantities. We may then be interested in some more general measure of the difference of $\tilde{r}$ and r,

$$\Delta(\tilde{r}, r),$$

where $\Delta(\cdot, \cdot)$ is a nonnegative, real-valued function. This is the absolute error, and the relative error is the ratio of the absolute error to $\Delta(r, r_0)$, where r_0 is a baseline value, such as 0. When r, instead of just being a single number, consists of several components, we must measure error differently. If r is a

vector, the measure may be some norm, such as we will discuss in Chapter 2. In that case, $\Delta(\tilde{r}, r)$ may be denoted by $\|(\tilde{r} - r)\|$. A norm tends to become larger as the number of elements increases, so instead of using a raw norm, it may be appropriate to scale the norm to reflect the number of elements being computed.

However the error is measured, for a given algorithm we would like to have some knowledge of the amount of error to expect or at least some bound on the error. Unfortunately, almost any measure contains terms that depend on the quantity being evaluated. Given this limitation, however, often we can develop an upper bound on the error. In other cases, we can develop an estimate of an "average error", based on some assumed probability distribution of the data comprising the problem. In a Monte Carlo method we estimate the solution based on a "random" sample, so just as in ordinary statistical estimation, we are concerned about the variance of the estimate. We can usually derive expressions for the variance of the estimator in terms of the quantity being evaluated, and of course we can estimate the variance of the estimator using the realized random sample. The standard deviation of the estimator provides an indication of the distance around the computed quantity within which we may have some confidence that the true value lies. The standard deviation is sometimes called the "standard error", and nonstatisticians speak of it as a "probabilistic error bound".

It is often useful to identify the "order of the error", whether we are concerned about error bounds, average expected error, or the standard deviation of an estimator. In general, we speak of the order of one function in terms of another function, as the argument of the functions approach a given value. A function $f(t)$ is said to be of order $g(t)$ at t_0, written $O(g(t))$ ("big O of $g(t)$"), if there exists a constant M such that

$$|f(t)| \leq M|g(t)| \quad \text{as } t \to t_0.$$

This is the *order of convergence* of one function to another at a given point.

If our objective is to compute $f(t)$ and we use an approximation $\tilde{f}(t)$, the order of the error *due to the approximation* is the order of the convergence. In this case, the argument of the order of the error may be some variable that defines the approximation. For example, if $\tilde{f}(t)$ is a finite series approximation to $f(t)$ using, say, n terms, we may express the error as $O(h(n))$, for some function $h(n)$. Typical orders of errors due to the approximation may be $O(1/n)$, $O(1/n^2)$, or $O(1/n!)$. An approximation with order of error $O(1/n!)$ is to be preferred over one order of error $O(1/n)$ because the error is decreasing more rapidly. The order of error due to the approximation is only one aspect to consider; roundoff error in the representation of any intermediate quantities must also be considered.

We will discuss the order of error in iterative algorithms further in the section beginning on page 37. We will discuss order also in measuring the speed of an algorithm in the section beginning on page 33.

The special case of convergence to the constant zero is often of interest. A function $f(t)$ is said to be "little o of $g(t)$" at t_0, written $o(g(t))$, if

$$f(t)/g(t) \to 0 \quad \text{as } t \to t_0.$$

If the function $f(t)$ approaches 0 at t_0, $g(t)$ can be taken as a constant and $f(t)$ is said to be $o(1)$.

Usually the limit on t in order expressions is either 0 or ∞, and because it is obvious from the context, mention of it is omitted. The order of the error in numerical computations usually provides a measure in terms of something that can be controlled in the algorithm, such as the point at which an infinite series is truncated in the computations. The measure of the error usually also contains expressions that depend on the quantity being evaluated, however.

Sources of Error in Numerical Computations

Some algorithms are exact, such as an algorithm to multiply two matrices that just uses the definition of matrix multiplication. Other algorithms are approximate because the result to be computed does not have a finite closed-form expression. An example is the evaluation of the normal cumulative distribution function. One way of evaluating this is by use of a rational polynomial approximation to the distribution function. Such an expression may be evaluated with very little rounding error, but the expression has an *error of approximation.*

We need to have some knowledge of the magnitude of the error. For algorithms that use approximations, it is often useful to express the order of the error in terms of some quantity used in the algorithm or in terms of some aspect of the problem itself.

When solving a differential equation on the computer, the differential equation is often approximated by a difference equation. Even though the differences used may not be constant, they are finite and the passage to the limit can never be effected. This kind of approximation leads to a discretization error. The amount of the discretization error has nothing to do with rounding error. If the last differences used in the algorithm are δt, then the error is usually of order $O(\delta t)$, even if the computations are performed exactly.

Another type of error occurs when the algorithm uses a series expansion. The infinite series may be exact, and in principle the evaluation of all terms would yield an exact result. The algorithm uses only a finite number of terms, and the resulting error is *truncation error.* When a truncated Taylor's series is used to evaluate a function at a given point x_0, the order of the truncation error is the derivative of the function that would appear in the first unused term of the series, evaluated at x_0.

Algorithms and Data

The performance of an algorithm may depend on the data. We have seen that even the simple problem of computing the roots of a quadratic polynomial,

$ax^2 + bx + c$, using the quadratic formula, equation (1.3), can lead to severe cancellation. For many values of a, b, and c, the quadratic formula works perfectly well. Data that are likely to cause computational problems are referred to as ill-conditioned data, and, more generally, we speak of the "condition" of data. The concept of condition is understood in the context of a particular set of operations. Heuristically, data for a given problem are ill-conditioned if small changes in the data may yield large changes in the solution.

Consider the problem of finding the roots of a high-degree polynomial, for example. Wilkinson (1959) gave an example of a polynomial that is very simple on the surface, yet whose solution is very sensitive to small changes of the values of the coefficients:

$$\begin{aligned} f(x) &= (x-1)(x-2)\cdots(x-20) \\ &= x^{20} - 210x^{19} + \cdots + 20! \end{aligned}$$

While the solution is easy to see from the factored form, the solution is very sensitive to perturbations of the coefficients. For example changing the coefficient 210 to $210+2^{-23}$ changes the roots drastically; in fact, 10 of them are now complex. Of course the extreme variation in the magnitudes of the coefficients should give us some indication that the problem may be ill-conditioned.

We attempt to quantify the condition of a set of data for a particular set of operations by means of a *condition number*. Condition numbers are defined to be positive and so that large values of the numbers means that the data or problems are ill-conditioned. A useful condition number for the problem of finding roots of a function can be defined in terms of the derivative of the function in the vicinity of a root. We will also see that condition numbers must be used with some care. For example, according to the condition number for finding roots, Wilkinson's polynomial is well-conditioned.

In the solution of a linear system of equations, the coefficient matrix determines the condition of this problem. In Sections 2.1 and 3.4 we will consider a condition number for a matrix with respect to the problem of solving a linear system of equations.

The ability of an algorithm to handle a wide range of data, and either to solve the problem as requested or to determine that the condition of the data does not allow the algorithm to be used is called the *robustness* of the algorithm. Another concept that is quite different from robustness is stability. An algorithm is said to be *stable* if it always yields a solution that is an *exact* solution to a perturbed problem; that is, for the problem of computing $f(x)$ using the input data x, an algorithm is stable if the result it yields, $\tilde{f}(x)$, is $f(x + \delta x)$ for some (bounded) perturbation δx of x. This concept of stability arises from backward error analysis. The stability of an algorithm depends on how continuous quantities are discretized, as when a range is gridded for solving a differential equation. See Higham (1996) for an extensive discussion of stability.

Reducing the Error in Numerical Computations

An objective in designing an algorithm to evaluate some quantity is to avoid accumulated rounding error and to avoid catastrophic cancellation. In the discussion of floating-point operations above, we have seen two examples of how an algorithm can be constructed to mitigate the effect of accumulated rounding error (using equations (1.2), page 22, for computing a sum) and to avoid possible catastrophic cancellation in the evaluation of the expression (1.3) for the roots of a quadratic equation.

Another example familiar to statisticians is the computation of the sample sum of squares:

$$\sum_{i=1}^{n}(x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - n\bar{x}^2 \tag{1.4}$$

This quantity is $(n-1)s^2$, where s^2 is the sample variance.

Either expression in equation (1.4) can be thought of as describing an algorithm. The expression on the left implies the "two-pass" algorithm:

$$\begin{array}{l} a = x_1 \\ \text{for } i = 2, \ldots, n \\ \{ \\ \quad a = x_i + a \\ \} \\ a = a/n \\ b = (x_1 - a)^2 \\ \text{for } i = 2, \ldots, n \\ \{ \\ \quad b = (x_i - a)^2 + b \\ \} \end{array} \tag{1.5}$$

Each of the sums computed in this algorithm may be improved by use of equations (1.2). A problem with this algorithm is the fact that it requires two passes through the data. Because the quantities in the second summation are squares of residuals, they are likely to be of relatively equal magnitude. They are of the same sign, so there will be no catastrophic cancellation in the early stages when the terms being accumulated are close in size to the current value of b. There will be some accuracy loss as the sum b grows, but the addends $(x_i - a)^2$ remain roughly the same size. The accumulated rounding error, however, may not be too bad.

The expression on the right of equation (1.4) implies the "one-pass" algorithm:

$$
\begin{array}{l}
a = x_1 \\
b = x_1^2 \\
\text{for } i = 2, \ldots, n \\
\{ \\
\quad a = x_i + a \\
\quad b = x_i^2 + b \\
\} \\
a = a/n \\
b = b - na^2
\end{array}
\tag{1.6}
$$

This algorithm requires only one pass through the data, but if the x_i's have magnitudes larger than 1, the algorithm has built up two relatively large quantities, b and na^2. These quantities may be of roughly equal magnitude; subtracting one from the other may lead to catastrophic cancellation. See Exercise 1.15, page 45.

Another algorithm is shown in (1.7). It requires just one pass through the data, and the individual terms are generally accumulated fairly accurately. Equations (1.7) are a form of the Kalman filter (see, for example, Grewal and Andrews, 1993).

$$
\begin{array}{l}
a = x_1 \\
b = 0 \\
\text{for } i = 2, \ldots, n \\
\{ \\
\quad d = (x_i - a)/i \\
\quad a = d + a \\
\quad b = i(i-1)d^2 + b \\
\}
\end{array}
\tag{1.7}
$$

Chan and Lewis (1979) propose a condition number to quantify the sensitivity in s, the sample standard deviation, to the data, the x_i's. Their condition number is

$$
\kappa = \frac{\sum_{i=1}^{n} x_i^2}{\sqrt{n-1}s}. \tag{1.8}
$$

It is clear that if the mean is large relative to the variance, this condition number will be large. (Recall that large condition numbers imply ill-conditioning; and also recall that condition numbers must be interpreted with some care.) Notice that this condition number achieves its minimum value of 1 for the data $x_i - \bar{x}$, so if the computations for $\bar{x}$ and $x_i - \bar{x}$ were exact, the data in the last part of the algorithm in (1.5) would be perfectly conditioned. A dataset with a large mean relative to the variance is said to be *stiff*.

Often when a finite series is to be evaluated, it is necessary to accumulate a set of terms of the series that have similar magnitude, and then combine this with similar partial sums. It may also be necessary to scale the individual terms by some very large or very small multiplicative constant while the terms are being accumulated, and then remove the scale after some computations have been performed.

Chan, Golub, and LeVeque (1982) propose a modification of the algorithm in (1.7) to use pairwise accumulations (as in the fan-in method discussed previously). Chan, Golub, and LeVeque (1983) make extensive comparisons of the methods, and give error bounds based on the condition number.

Efficiency

The *efficiency* of an algorithm refers to its usage of computer resources. The two most important resources are the processing units and memory. The amount of time the processing units are in use and the amount of memory required are the key measures of efficiency. A limiting factor for the time the processing units are in use is the number and type of operations required. Some operations take longer than others; for example, the operation of adding floating-point numbers may take more time than the operation of adding fixed-point numbers. This, of course, depends on the computer system and on what kinds of floating-point or fixed-point numbers we are dealing with. If we have a measure of the size of the problem, we can characterize the performance of a given algorithm by specifying the number of operations of each type, or just the number of operations of the slowest type.

If more than one processing unit is available, it may be possible to perform operations simultaneously. In this case the amount of time required may be drastically smaller for an efficient parallel algorithm than it would for the most efficient serial algorithm that utilizes only one processor at a time. An analysis of the efficiency must take into consideration how many processors are available, how many computations can be performed in parallel, and how often they can be performed in parallel.

Often instead of the exact number of operations, we use the *order* of the number of operations in terms of the measure of problem size. If n is some measure of the size of the problem, an algorithm has order $O(f(n))$ if, as $n \to \infty$, the number of computations $\to cf(n)$, where c is some constant. For example, to multiply two $n \times n$ matrices in the obvious way requires $O(n^3)$ multiplications and additions; to multiply an $n \times m$ matrix and an $m \times p$ matrix requires $O(nmp)$ multiplications and additions. In the latter case, n, m, and p are all measures of the size of the problem.

Notice that in the definition of order there is a constant c. Two algorithms that have the same order may have different constants, and in that case are said to "differ only in the constant". The order of an algorithm is a measure of how well the algorithm "scales"; that is, the extent to which the algorithm can deal with truly large problems.

Let n be a measure of the problem size, and let b and q be constants. An algorithm of order $O(b^n)$ has *exponential order*, one of order $O(n^q)$ has *polynomial order*, and one of order $O(\log n)$ has *log order*. Notice that for log order, it does not matter what the base is. Also, notice that $O(\log n^q) = O(\log n)$. For a given task with an obvious algorithm that has polynomial order, it is often possible to modify the algorithm to address parts of the problem so

that in the order of the resulting algorithm one n factor is replaced by a factor of $\log n$.

Although it is often relatively easy to determine the order of an algorithm, an interesting question in algorithm design involves the *order of the problem*, that is, the order of the most efficient algorithm possible. A problem of polynomial order is usually considered tractable, whereas one of exponential order may require a prohibitively excessive amount of time for its solution. An interesting class of problems are those for which a solution can be verified in polynomial time, yet for which no polynomial algorithm is known to exist. Such a problem is called a *nondeterministic polynomial*, or NP, problem. "Nondeterministic" does not imply any randomness; it refers to the fact that no polynomial algorithm for determining the solution is known. Most interesting NP problems can be shown to be equivalent to each other in order by reductions that require polynomial time. Any problem in this subclass of NP problems is equivalent in some sense to all other problems in the subclass and so such a problem is said to be *NP-complete*. (See Garey and Johnson, 1979, for a complete discussion of NP-completeness.)

For many problems it is useful to measure the size of a *problem* in some standard way and then to identify the order of an *algorithm* for the problem with separate components. A common measure of the size of a problem is L, the length of the stream of data elements. An $n \times n$ matrix would have length proportional to $L = n^2$, for example. To multiply two $n \times n$ matrices in the obvious way requires $O(L^{3/2})$ multiplications and additions, as we mentioned above.

In analyzing algorithms for more complicated problems, we may wish to determine the order in the form

$$O(f(n)g(L)),$$

because L is an essential measure of the problem size, and n may depend on how the computations are performed. For example, in the linear programming problem, with n variables and m constraints with a dense coefficient matrix, there are order nm data elements. Algorithms for solving this problem generally depend in the limit on n, so we may speak of a linear programming algorithm as being $O(n^3L)$, for example, or of some other algorithm as being $O(\sqrt{n}L)$. (In defining L, it is common to consider the magnitudes of the data elements or the precision with which the data are represented, so that L is the order of the total number of bits required to represent the data. This level of detail can usually be ignored, however, because the limits involved in the order are generally not taken on the magnitude of the data, only on the number of data elements.)

The order of an algorithm (or, more precisely, the "order of *operations* of an algorithm") is an asymptotic measure of the operation count as the size of the problem goes to infinity. The order of an algorithm is important, but in practice the actual count of the operations is also important. In practice, an algorithm whose operation count is approximately n^2 may be more useful than one whose

count is $1000(n \log n + n)$, although the latter would have order $O(n \log n)$, which is much better than that of the former, $O(n^2)$. When an algorithm is given a fixed-size task many times, the finite efficiency of the algorithm becomes very important.

The number of computations required to perform some tasks depends not only on the size of the problem, but also on the data. For example, for most sorting algorithms, it takes fewer computations (comparisons) to sort data that are already almost sorted than it does to sort data that are completely unsorted. We sometimes speak of the *average* time and the *worst-case* time of an algorithm. For some algorithms these may be very different, whereas for other algorithms or for some problems these two may be essentially the same.

Our main interest is usually not in how many computations occur, but rather in how long it takes to perform the computations. Because some computations can take place simultaneously, even if all kinds of computations required the same amount of time, the *order of time* may be different from the order of the number of computations.

The actual number of floating-point operations divided by the time required to perform the operations is called the FLOPS (floating-point operations per second) rate. Confusingly, "FLOP" also means "floating-point operation", and "FLOPs" is the plural of "FLOP". Of course, as we tend to use lowercase more often, we must use the context to distinguish "flops" as a rate from "flops", the plural of "flop".

In addition to the actual processing, the data may need to be copied from one storage position to another. Data movement slows the algorithm, and may cause it not to use the processing units to their fullest capacity. When groups of data are being used together, blocks of data may be moved from ordinary storage locations to an area from which they can be accessed more rapidly. The efficiency of a program is enhanced if all operations that are to be performed on a given block of data are performed one right after the other. Sometimes a higher-level language prevents this from happening. For example, to add two arrays (matrices) in Fortran 90, a single statement is sufficient:

```
A = B + C
```

Now, if also we want to add `B` to the array `E` we may write:

```
A = B + C
D = B + E
```

These two Fortran 90 statements together may be less efficient than writing a traditional loop in Fortran or in C, because the array `B` may be accessed a second time needlessly. (Of course, this is relevant only if these arrays are very large.)

Improving Efficiency

There are many ways to attempt to improve the efficiency of an algorithm. Often the best way is just to look at the task from a higher level of detail,

and attempt to construct a new algorithm. Many obvious algorithms are serial methods that would be used for hand computations, and so are not the best for use on the computer.

An effective general method of developing an efficient algorithm is called *divide and conquer*. In this method, the problem is broken into subproblems, each of which is solved, and then the subproblem solutions are combined into a solution for the original problem. In some cases, this can result in a net savings either in the number of computations, resulting in improved order of computations, or in the number of computations that must be performed serially, resulting in improved order of time.

Let the time required to solve a problem of size n be $t(n)$, and consider the recurrence relation

$$t(n) = pt(n/p) + cn,$$

for p positive and c nonnegative. Then $t(n) = O(n \log n)$ (see Exercise 1.17, page 45). Divide and conquer strategies can sometimes be used together with a simple method that would be $O(n^2)$ if applied directly to the full problem to reduce the order to $O(n \log n)$.

The "fan-in algorithm" is an example of a divide and conquer strategy that allows $O(n)$ operations to be performed in $O(\log n)$ time if the operations can be performed simultaneously. The number of operations does not change materially; the improvement is in the time.

Although there have been orders of magnitude improvements in the speed of computers because the hardware is better, the order of time required to solve a problem is dependent almost entirely on the algorithm. The improvement in efficiency resulting from hardware improvements are generally differences only in the constant. The practical meaning of the order of the time must be considered, however, and so the constant may be important. In the fan-in algorithm, for example, the improvement in order is dependent on the unrealistic assumption that as the problem size increases without bound the number of processors also increases without bound. (Not all divide and conquer strategies require multiple processors for their implementation, of course.)

Some algorithms are designed so that each step is as efficient as possible, without regard to what future steps may be part of the algorithm. An algorithm that follows this principle is called a *greedy algorithm*. A greedy algorithm is often useful in the early stages of computation for a problem, or when a problem lacks an understandable structure.

Bottlenecks and Limits

There is maximum FLOPS rate possible for a given computer system. This rate depends on how fast the individual processing units are, how many processing units there are, and how fast data can be moved around in the system. The more efficient an algorithm is, the closer its achieved FLOPS rate is to the maximum FLOPS rate.

For a given computer system, there is also a maximum FLOPS rate possible for a given problem. This has to do with the nature of the tasks within the given problem. Some kinds of tasks can utilize various system resources more easily than other tasks. If a problem can be broken into two tasks, T_1 and T_2, such that T_1 must be brought to completion before T_2 can be performed, the total time required for the problem depends more on the task that takes longer. This tautology has important implications for the limits of efficiency of algorithms. It is the basis of "Amdahl's law" or "Ware's law" (Amdahl, 1967) that puts limits on the speedup of problems that consist of both tasks that must be performed sequentially and tasks that can be performed in parallel. It is also the basis of the childhood riddle:

> You are to make a round trip to a city 100 miles away. You want to average 50 miles per hour. Going, you travel at a constant rate of 25 miles per hour. How fast must you travel coming back?

The efficiency of an algorithm may depend on the organization of the computer, on the implementation of the algorithm in a programming language, and on the way the program is compiled.

Iterations and Convergence

Many numerical algorithms are iterative; that is, groups of computations form successive approximations to the desired solution. In a program, this usually means a loop through a common set of instructions in which each pass through the loop changes the initial values of operands in the instructions.

We will generally use the notation $x^{(k)}$ to refer to the computed value of x at the k^{th} iteration.

An iterative algorithm terminates when some *convergence criterion* or *stopping criterion* is satisfied. An example is to declare that an algorithm has converged when

$$\Delta(x^{(k)}, x^{(k-1)}) \leq \epsilon,$$

where $\Delta(x^{(k)}, x^{(k-1)})$ is some measure of the difference of $x^{(k)}$ and $x^{(k-1)}$ and ϵ is a small positive number. Because x may not be a single number, we must consider general measures of the difference of $x^{(k)}$ and $x^{(k-1)}$. For example, if x is a vector, the measure may be some norm, such as we discuss in Chapter 2. In that case, $\Delta(x^{(k)}, x^{(k-1)})$ may be denoted by $\|x^{(k)} - x^{(k-1)}\|$.

An iterative algorithm may have more than one stopping criterion. Often, a maximum number of iterations is set, so that the algorithm will be sure to terminate whether it converges or not. (Some people define the term "algorithm" to refer only to methods that converge. Under this definition, whether or not a method is an "algortihm" may depend on the input data, unless a stopping rule based on something independent of the data, such as number of iterations, is applied. In any event, it is always a good idea, in addition to stopping criteria based on convergence of the solution, to have a stopping criterion that is independent of convergence and that limits the number of operations.)

The *convergence ratio* of the sequence $x^{(k)}$ to a constant x_0 is

$$\lim_{k\to\infty} \frac{\Delta(x^{(k+1)}, x_0)}{\Delta(x^{(k)}, x_0)},$$

if this limit exists. If the convergence ratio is greater than 0 and less than 1, the sequence is said to converge *linearly*. If the convergence ratio is 0, the sequence is said to converge *superlinearly*.

Other measures of the rate of convergence are based on

$$\lim_{k\to\infty} \frac{\Delta(x^{(k+1)}, x_0)}{(\Delta(x^{(k)}, x_0))^r} = c, \tag{1.9}$$

(again, assuming the limit exists, i.e., $c < \infty$.) In (1.9), the exponent r is called the *rate of convergence*, and the limit c is called the *rate constant*. If $r = 2$ (and c is finite), the sequence is said to converge *quadratically*. It is clear that for any $r > 1$ (and finite c), the convergence is superlinear.

The convergence rate is often a function of k, say $h(k)$. The convergence is then expressed as an order in k, $O(h(k))$.

Extrapolation

As we have noted, many numerical computations are preformed on a discrete set that approximates the reals or $\mathbb{R}^d$, resulting in discretization errors. By "discretization error" we do not mean a rounding error resulting from the computer's finite representation of numbers. The discrete set used in computing some quantity such as an integral is often a grid. If h is the interval width of the grid, the computations may have errors that can be expressed as a function of h. For example, if the true value is x, and because of the discretization, the *exact value* that would be computed is x_h, then we can write

$$x = x_h + e(h).$$

For a given algorithm, suppose the error $e(h)$ is proportional to some power of h, say h^n, and so we can write

$$x = x_h + ch^n, \tag{1.10}$$

for some constant c. Now, suppose we use a different discretization, with interval length rh, with $0 < r < h$. We have

$$x = x_{rh} + c(rh)^n,$$

and so, after subtracting,

$$0 = x_h - x_{rh} + c(h^n - (rh)^n),$$

or

$$ch^n = \frac{(x_h - x_{rh})}{r^n - 1}. \tag{1.11}$$

This analysis relies on the assumption that the error in the discrete algorithm is proportional to h^n. Under this assumption, ch^n in (1.11) is the discretization error in computing x, using exact computations, and is an estimate of the error due to discretization in actual computations. A more realistic regularity assumption is that the error is $O(h^n)$ as $h \to 0$; that is, instead of (1.10), we have

$$x = x_h + ch^n + O(h^{n+\alpha}),$$

for $\alpha > 0$.

Whenever this regularity assumption is satisfied, equation (1.11) provides us with with an inexpensive improved estimate of x:

$$x_R = \frac{x_{rh} - r^n x_h}{1 - r^n}. \tag{1.12}$$

It is easy to see that $|x - x_R|$ is less than the absolute error using an interval size of either h or rh.

This process described above is called Richardson extrapolation and the value in (1.12) is called the Richardson extrapolation estimate. Richardson extrapolation is also called "Richardson's deferred approach to the limit". It has general applications in numerical analysis, but is most widely used in numerical quadrature. Bickel and Yahav (1988) use Richardson extrapolation to reduce the computations in a bootstrap.

Extrapolation can be extended beyond just one step, as in the presentation above.

Reducing the computational burden by use of extrapolation is very important in higher dimensions. In many cases, for example in direct extensions of quadrature rules, the computational burden grows exponentially in the number of dimensions. This is sometimes called "the curse of dimensionality",and can render a fairly straightforward problem in one or two dimensions unsolvable in higher dimensions.

A direct extension of Richardson extrapolation in higher dimensions would involve extrapolation in each direction, with an exponential increase in the amount of computation. An approach that is particularly appealing in higher dimensions is splitting extrapolation, which avoids independent extrapolations in all directions. See Liem, Lü, and Shih (1995) for an extensive discussion of splitting extrapolation, with numerous applications.

Recursion

The algorithms for many computations perform some operation, update the operands, and perform the operation again.

1. perform operation
2. test for exit
3. update operands
4. go to 1

If we give this algorithm the name `doit`, and represent its operands by x, we could write the algorithm as

Algorithm `doit`(x)
1. operate on x
2. test for exit
3. update x: x'
4. `doit`(x')

The algorithm for computing the mean and the sum of squares (1.7) can be derived as a recursion. Suppose we have the mean a_k and the sum of squares, s_k, for k elements $x_1, x_2, \ldots, x_k$, and we have a new value x_{k+1} and wish to compute a_{k+1} and s_{k+1}. The obvious solution is

$$a_{k+1} = a_k + \frac{x_{k+1} - a_k}{k+1}$$

and

$$s_{k+1} = s_k + \frac{k(x_{k+1} - a_k)^2}{k+1}.$$

These are the same computations as in equations (1.7) on page 32.

Another example of how viewing the problem as an update problem can result in an efficient algorithm is in the evaluation of a polynomial of degree d,

$$p_d(x) = c_d x^d + c_{d-1} x^{d-1} + \cdots + c_1 x + c_0.$$

Doing this in a naive way would require $d-1$ multiplications to get the powers of x, d additional multiplications for the coefficients, and d additions. If we write the polynomial as

$$p_d(x) = x(c_d x^{d-1} + c_{d-1} x^{d-2} + \cdots + c_1) + c_0,$$

we see a polynomial of degree $d-1$ from which our polynomial of degree d can be obtained with but one multiplication and one addition; that is, the number of multiplications is equal to the increase in the degree — not two times the increase in the degree. Generalizing, we have

$$p_d(x) = x(\cdots x(x(c_d x + c_{d-1}) + \cdots) + c_1) + c_0, \tag{1.13}$$

which has a total of d multiplications and d additions. The method for evaluating polynomials in (1.13) is called *Horner's method.*

A computer subprogram that implements recursion invokes itself. Not only must the programmer be careful in writing the recursive subprogram, the programming system must maintain call tables and other data properly to allow for recursion. Once a programmer begins to understand recursion, there may be a tendency to overuse it. To compute a factorial, for example, the inexperienced C programmer may write

```
float Factorial(int n)
   {
    if(n==0)
      return 1;
    else
      return n*Factorial(n-1);
   }
```

The problem is that this is implemented by storing a stack of statements. Because n may be relatively large, the stack may become quite large and inefficient. It is just as easy to write the function as a simple loop, and it would be a much better piece of code.

Both C and Fortran 90 allow for recursion. Many versions of Fortran have supported recursion for years, but it was not part of the earlier Fortran standards.

Exercises

1.1. An important attitude in the computational sciences is that the computer is to be used as a tool of exploration and discovery. The computer should be used to check out "hunches" or conjectures, which then later should be subjected to analysis in the traditional manner. There are limits to this approach, however. An example is in limiting processes. Because the computer deals with finite quantities, the results of a computation may be misleading. Explore each of the situations below, using C or Fortran. A few minutes or even seconds of computing should be enough to give you a feel for the nature of the computations.

In these exercises, you may write computer programs in which you perform tests for equality. A word of warning is in order about such tests. If a test involving a quantity `x` is executed soon after the computation of `x`, the test may be invalid within the set of floating-point numbers with which the computer nominally works. This is because the test may be performed using the extended precision of the computational registers.

(a) Consider the question of the convergence of the series

$$\sum_{i=1}^{\infty} i.$$

Obviously, this series does not converge in $\mathbb{R}$. Suppose, however, that we begin summing this series using floating-point numbers. Will the series overflow? If so, at what value of i (approximately)? Or will the series converge in $\mathbb{F}$? If so, to what value, and at what value of i (approximately)? In either case, state your answer in terms of the standard parameters of the floating-point model, b, p, $e_{\min}$, and $e_{\max}$ (page 6).

(b) Consider the question of the convergence of the series

$$\sum_{i=1}^{\infty} 2^{-2i}.$$

(Same questions as above.)

(c) Consider the question of the convergence of the series

$$\sum_{i=1}^{\infty} \frac{1}{i}.$$

(Same questions.)

(d) Consider the question of the convergence of the series

$$\sum_{i=1}^{\infty} \frac{1}{i^x},$$

for $x \geq 1$. (Same questions, except address the variable x.)

1.2. We know, of course, that the harmonic series in Exercise 1.1c does not converge (although the naive program to compute it does). It is, in fact, true that

$$\begin{aligned} H_n &= \sum_{i=1}^{n} \frac{1}{i} \\ &= f(n) + \gamma + o(1), \end{aligned}$$

where f is an increasing function and γ is Euler's constant. For various n, compute H_n. Determine a function f that provides a good fit and obtain an approximation of Euler's constant.

1.3. Machine characteristics.

(a) Write a program to determine the smallest and largest relative spacings. Use it to determine them on the machine you are using.

(b) Write a program to determine whether your computer system implements gradual underflow.

(c) Write a program to determine the bit patterns of $+\infty$, $-\infty$, and NaN on a computer that implements the IEEE binary standard. (This may be more difficult than it seems.)

(d) Obtain the program MACHAR (Cody, 1988b) and use it to determine the smallest positive floating-point number on the computer you are using. (MACHAR is included in CALGO, which is available from `netlib`. See the bibliography.)

1.4. Write a program in Fortran or C to determine the bit patterns of fixed-point numbers, of floating-point numbers, and of character strings. Run your program on different computers and compare your results with those shown in Figures 1.1 through 1.3 and Figures 1.11 through 1.13.

1.5. What is the rounding unit ($\frac{1}{2}$ ulp) in the IEEE Standard 754 double precision?

1.6. Consider the standard model (1.1) for the floating-point representation:

$$\pm 0.d_1 d_2 \cdots d_p \times b^e,$$

with $e_{\min} \leq e \leq e_{\max}$. Your answers may depend on an additional assumption or two. Either choice of (standard) assumptions is acceptable.

(a) How many floating-point numbers are there?

(b) What is the smallest positive number?

(c) What is the smallest number larger than 1?

(d) What is the smallest number X, such that $X + 1 = X$?

(e) Suppose $p = 4$ and $b = 2$ (and $e_{\min}$ is very small and $e_{\max}$ is very large). What is the next number after 20 in this number system?

1.7. (a) Define parameters of a floating-point model so that the number of numbers in the system is less than the largest number in the system.

(b) Define parameters of a floating-point model so that the number of numbers in the system is greater than the largest number in the system.

1.8. Suppose that a certain computer represents floating point numbers in base 10, using eight decimal places for the mantissa, two decimal places for the exponent, one decimal place for the sign of exponent, and one decimal place for the sign of the number.

(a) What is the "smallest relative spacing" and the "largest relative spacing"? (Your answer may depend on certain additional assumptions about the representation; state any assumptions.)

(b) What is the largest number g, such that $417 + g = 417$?

(c) Discuss the associativity of addition using numbers represented in this system. Give an example of three numbers, a, b, and c, such that using this representation, $(a+b)+c \neq a+(b+c)$, unless the operations are chained. Then show how chaining could make associativity hold for some more numbers, but still not hold for others.

(d) Compare the maximum rounding error in the computation $x + x + x + x$ with that in $4 * x$. (Again, you may wish to mention the possibilities of chaining operations.)

1.9. Consider the same floating-point system of Exercise 1.8.

(a) Let X be a random variable uniformly distributed over the interval

$$[1 - .000001,\ 1 + .000001].$$

Develop a probability model for the representation $[X]_c$. (This is a discrete random variable with 111 mass points.)

(b) Let X and Y be random variables uniformly distributed over the same interval as above. Develop a probability model for the representation $[X + Y]_c$. (This is a discrete random variable with 121 mass points.)

(c) Develop a probability model for $[X]_c\ [+]_c\ [Y]_c$. (This is also a discrete random variable with 121 mass points.)

1.10. Give an example to show that the sum of three floating-point numbers can have a very large relative error.

1.11. Write a single program in Fortran or C to compute

(a)

$$\sum_{i=0}^{5} \binom{10}{i} 0.25^i 0.75^{20-i}$$

(b)

$$\sum_{i=0}^{10} \binom{20}{i} 0.25^i 0.75^{20-i}$$

(c)

$$\sum_{i=0}^{50} \binom{100}{i} 0.25^i 0.75^{20-i}$$

1.12. Suppose you have a program to compute the cumulative distribution function for the chi-squared distribution (the input is x and df, and the output is $\Pr(X \leq x)$). Suppose you are interested in probabilities in the extreme upper range and high accuracy is very important. What is wrong with the design of the program?

1.13. Write a program in Fortran or C to compute e^{-12} using a Taylor's series directly, and then compute e^{-12} as the reciprocal of e^{12}, which is also computed using a Taylor's series. Discuss the reasons for the differences in the results. To what extent is truncation error a problem?

1.14. Errors in computations.

(a) Explain the difference in truncation and cancellation.

(b) Why is cancellation not a problem in multiplication?

1.15. Assume we have a computer system that can maintain 7 digits of precision. Evaluate the sum of squares for the data set $\{9000, 9001, 9002\}$.

(a) Use the algorithm in (1.5), page 31.

(b) Use the algorithm in (1.6), page 32.

(c) Now assume there is one guard digit. Would the answers change?

1.16. Develop algorithms similar to (1.7) on page 32 to evaluate the following.

(a) The weighted sum of squares:

$$\sum_{i=1}^{n} w_i (x_i - \bar{x})^2$$

(b) The third central moment:

$$\sum_{i=1}^{n} (x_i - \bar{x})^3$$

(c) The sum of cross products:

$$\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

Hint: Look at the difference in partial sums,

$$\sum_{i=1}^{j} (\cdot) - \sum_{i=1}^{j-1} (\cdot)$$

1.17. Given the recurrence relation

$$t(n) = pt(n/p) + cn,$$

for p positive and c nonnegative. Show that $t(n)$ is $O(n \log n)$. *Hint*: First assume n is a power of p.

1.18. In statistical data analysis, it is common to have some missing data. This may be because of nonresponse in a survey questionnaire or because an experimental or observational unit dies or discontinues participation in the study. When the data are recorded, some form of missing-data indicator must be used. Discuss the use of NaN as a missing-value indicator. What are some advantages and disadvantages?

Chapter 2

Basic Vector/Matrix Computations

Vectors and matrices are useful in representing multivariate data, and they occur naturally in working with linear equations or when expressing linear relationships among objects. Numerical algorithms for a variety of tasks involve matrix and vector arithmetic. An optimization algorithm to find the minimum of a function, for example, may use a vector of approximate first derivatives and a matrix of second derivatives; and a method to solve a differential equation may use a matrix with a few diagonals for computing differences. There are various precise ways of defining vectors and matrices, but we will think of them merely as arrays of numbers, or scalars, on which an algebra is defined.

We assume the reader has a working knowledge of linear algebra, but in this first section, going all the way to page 81, we give several definitions and state many useful facts about vectors and matrices. Many of these properties will be used in later chapters. Some general references covering properties of vectors and matrices, with particular attention to applications in statistics, include Basilevsky (1983), Graybill (1983), Harville (1997), Schott (1996), and Searle (1982).

In this chapter the presentation is informal; neither definitions nor facts are highlighted by such words as "Definition", "Theorem", "Lemma", and so forth. The facts generally have simple proofs, but formal proofs are usually not given — although sometimes they appear as exercises!

In Section 2.2, beginning on page 81, we discuss some of the basic issues of vector/matrix storage and computations on a computer. After consideration of numerical methods for solving linear systems and for eigenanalysis in Chapters 3 and 4, we resume the discussion of computer manipulations and software in Chapter 5.

General references on numerical linear algebra include Demmel (1997), Forsythe and Moler (1967), Golub and Van Loan (1996), Higham (1996), Lawson

and Hanson (1974 and 1995), Stewart (1973), Trefethen and Blau (1997),and Watkins (1991).

References that emphasize computations for statistical applications include Chambers (1977), Heiberger (1989), Kennedy and Gentle (1980), Maindonald (1984), Thisted (1988), and Tierney (1990).

References that describe parallel computations for linear algebra include Fox et al. (1988), Gallivan et al. (1990), and Quinn (1994).

We occasionally refer to two standard software packages for linear algebra, LINPACK (Dongarra et al., 1979) and LAPACK. (Anderson et al., 1995). We discuss these further in Chapter 5.

2.1 Notation, Definitions, and Basic Properties

A vector (or n-vector) is an n-tuple, or ordered (multi)set, or array, of n numbers, called *elements*. The number of elements is sometimes called the *order*, or sometimes the "length", of the vector. An n-vector can be thought of as representing a point in n-dimensional space. In this setting, the length of the vector may also mean the Euclidean distance from the origin to the point represented by the vector, that is, the square root of the sum of the squares of the elements of the vector. This Euclidean distance will generally be what we mean when we refer to the length of a vector.

The first element of an n-vector is the first (1^{st}) element and the last is the n^{th} element. (This statement is not a tautology; in some computer systems, the first element of an object used to represent a vector is the 0^{th} element of the object. This sometimes makes it difficult to preserve the relationship between the computer entity and the object that is of interest.) We will use paradigms and notation that maintain the priority of the object of interest, rather than the computer entity representing it.

We may write the n-vector x as

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix},$$

or as

$$x = (x_1, x_2, \ldots, x_n).$$

We make no distinction between these two notations, although in some contexts we think of a vector as a "column", so the first notation may be more natural.

2.1.1 Operations on Vectors; Vector Spaces

The elements of a vector are elements of a field, and most vector operations are defined in terms of operations in the field. The elements of the vectors we will use in this book are real numbers, that is, elements of $\mathbb{R}$.

Two vectors can be added if they are of the same length (that is, have the same number of elements); the sum of two vectors is the vector whose elements are the sums of the corresponding elements of the addends. Vectors with the same number of elements are said to be *conformable* for addition. A scalar multiple of a vector, that is, the product of an element from the field and a vector, is the vector whose elements are the multiples of the corresponding elements of the original vector.

We overload the usual symbols for the operations on the reals for the corresponding operations on vectors or matrices when the operations are defined, so "+", for example, can mean addition of scalars or addition of conformable vectors.

A very common operation in working with vectors is the addition of a scalar multiple of one vector to another vector:

$$ax + y,$$

where a is a scalar and x and y are vectors of equal length. Viewed as a single operation with three operands, this is called an "axpy" for obvious reasons. (Because the Fortran versions of BLAS to perform this operation were called `saxpy` and `daxpy`, the operation is also sometimes called "saxpy" or "daxpy". See Section 5.1.1, page 140, for a description of the BLAS.) Such *linear combinations* of vectors are important operations.

If a given vector can be formed by a linear combination of one or more vectors, the set of vectors (including the given one) is said to be linearly dependent; conversely, if in a set of vectors no one vector can be represented as a linear combination of any of the others, the set of vectors is said to be *linearly independent*. It is easy to see that the maximum number of n-vectors that can form a set that is linearly independent is n. Linear independence is one of the most important concepts in linear algebra.

Let V be a set of n-vectors such that for any vectors in V, any linear combination of those vectors is also in V. Then the set V together with the usual vector algebra is called a *vector space*. (Technically, the "usual algebra" is for the operations of vector addition and scalar times vector multiplication. It has closure of the space under axpy, commutativity and associativity of addition, an additive identity and inverses, a multiplicative identity, distribution of multiplication over both vector addition and scalar addition, and associativity of scalar multiplication and scalar times vector multiplication. See, for example, Thrall and Tornheim, 1957.)

The length or order of the vectors is the *order of the vector space*, and the maximum number of linearly independent vectors in the space is the *dimension of the vector space*.

We generally use a calligraphic font to denote a vector space; $\mathcal{V}$, for example.

Although a vector space is a set together with operations, we often speak of a vector space as if it were a set; and we use some of the same notation to refer to vector spaces as the notation used to refer to sets. For example, if $\mathcal{V}$ is a vector space, the notation $\mathcal{W} \subseteq \mathcal{V}$ indicates that $\mathcal{W}$ is a vector space, that the

set of vectors in the vector space $\mathcal{W}$ is a subset of the vectors in $\mathcal{V}$, and that the operations in the two objects are the same. A subset of a vector space $\mathcal{V}$ that is itself a vector space is called a *subspace* of $\mathcal{V}$.

The intersection of two vector spaces is a vector space, but their union is not necessarily a vector space. If $\mathcal{V}_1$ and $\mathcal{V}_2$ are vector spaces, the space of vectors

$$\mathcal{V} = \{v \ ; \ v = v_1 + v_2, \ v_1 \in \mathcal{V}_1, \ v_2 \in \mathcal{V}_2\}$$

is called the *sum* (or *direct sum*) of the vector spaces $\mathcal{V}_1$ and $\mathcal{V}_2$. The relation is denoted by

$$\mathcal{V} = \mathcal{V}_1 \oplus \mathcal{V}_2.$$

If each vector in the vector space $\mathcal{V}$ can be expressed as a linear combination of the vectors in the set G, then G is said to be a *generating set* or *spanning set* of $\mathcal{V}$, and this construction of the vector space may be denoted by $\mathcal{V}(G)$. This vector space is also denoted by "span(G)". A set of linearly independent vectors that span a space is said to be a *basis* for the space.

We denote the additive identity in a vector space of order n by 0_n, or sometimes by 0. This is the vector consisting of all zeros. Likewise, we denote the vector consisting of all ones, by 1_n, or sometimes by 1. Whether 0 and 1 represent vectors or scalars is usually clear from the context. The vector space consisting of all n-vectors with real elements is denoted $\mathbb{R}^n$.

Points in a Cartesian geometry can be identified with vectors. Geometrically, a point with Cartesian coordinates $(x_1, \ldots, x_n)$ is associated with a vector from the origin to the point, that is, the vector $(x_1, \ldots, x_n)$.

The elements of a vector often represent coefficients of scalar variables; for example, given the variables $x_1, x_2, \ldots, x_n$, we may be interested in the linear combination

$$c_1x_1 + c_2x_2 + \ldots + c_nx_n.$$

The vector $c = (c_1, c_2, \ldots, c_n)$ is the coefficient vector and the sum $\sum_i c_i x_i$ is the *dot product* or *inner product* of the vectors c and x. (The dot product is actually a special type of inner product, but it is the most commonly used inner product.) We denote the dot product of c and x by $\langle c, x \rangle$. The dot product is also sometimes written as $c \cdot x$, hence the name. Yet another notation for the dot product is $c^{\mathrm{T}}x$, and we see later that this notation is natural in the context of matrix multiplication.

The dot product is a mapping from a vector space $\mathcal{V}$ into $\mathbb{R}$ that has the following properties:

1. Nonnegativity and mapping of the identity:
   if $x \neq 0$, then $\langle x, x \rangle > 0$ and $\langle 0, 0 \rangle = 0$.

2. Commutativity:
   $\langle x, y \rangle = \langle y, x \rangle$.

3. Factoring of scalar multiplication in dot products:
   $\langle ax, y \rangle = a\langle x, y \rangle$ for real a.

4. Relation of vector addition to addition of dot products:
   $\langle x + y, z\rangle = \langle x, z\rangle + \langle y, z\rangle$.

These properties in fact define the more general inner product. A vector space together with such an operator is called an *inner product space.*

We also denote the dot product by $c^{\mathrm{T}}x$, as we do with matrix multiplication. (The dot product is *not* the same as matrix multiplication, because the product is a scalar.)

A useful property of inner products is the Cauchy-Schwarz inequality:

$$\langle x, y\rangle \leq \langle x, x\rangle^{\frac{1}{2}} \langle y, y\rangle^{\frac{1}{2}}. \tag{2.1}$$

This is easy to see, by first observing for every real number t,

$$\begin{aligned} 0 &\leq (\langle (tx + y), (tx + y)\rangle)^2 \\ &= \langle x, x\rangle t^2 + 2\langle x, y\rangle t + \langle y, y\rangle \\ &= at^2 + bt + c, \end{aligned}$$

where the constants a, b, and c correspond to the dot products in the preceding equation. This quadratic in t cannot have two distinct real roots, hence the discriminant, $b^2 - 4ac$, must be less than or equal to zero; that is,

$$\left(\frac{1}{2}b\right)^2 \leq ac.$$

By substituting and taking square roots, we get the Cauchy-Schwarz inequality. It is also clear from this proof that equality holds only if $x = 0$ or if $y = ax$, for some scalar x.

The *length* of the vector x is $\sqrt{\langle x, x\rangle}$. The length is also called the *norm* of the vector, although as we see below, it is just one of many norms. The angle θ between the vectors x and y is defined by

$$\cos(\theta) = \frac{\langle x, y\rangle}{\sqrt{\langle x, x\rangle \langle y, y\rangle}}.$$

(This definition is consistent with the geometric interpretation of the vectors.)

Subsets of points defined by linear equations are called *flats.* In an n-dimensional Cartesian system (or a vector space of order n), the flat consisting of the points that satisfy an equation

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n = 0$$

is called a *hyperplane.* Lines and other flat geometric objects can be defined by systems of linear equations. See Kendall (1961) for discussions of n-dimensional geometric objects such as flats. Thrall and Tornheim (1957) discuss these objects in the context of vector spaces.

2.1.2 Vectors and Matrices

A matrix is a rectangular array. The number of dimensions of an array is often called the *rank* of the array. Thus, a vector is an array of rank 1 and a matrix is an array of rank 2. A scalar has rank 0. When referring to computer software objects, "rank" is generally used in this sense. On page 59 we discuss a different meaning of the work "rank", and one that is more often used in linear algebra.

The elements or components of either a vector or a matrix are elements of a field. We generally assume the elements are real numbers, although sometimes we have occasion to work with matrices whose elements are complex numbers.

We speak of the *rows* and *columns* of a matrix. An $n \times m$ matrix is one with n rows and m columns. The number of rows and the number of columns determine the *shape* of the matrix. If the number of rows is the same as the number of columns, the matrix is said to be square; otherwise, it is called nonsquare.

We usually use a lower-case letter to represent a vector, and we use the same letter with a single subscript to represent an element of the vector. We usually use an upper-case letter to represent a matrix. To represent an element of the matrix, we use the corresponding lower-case letter with a subscript to denote the row and a second subscript to represent the column. If a nontrivial expression is used to denote the row or the column, we separate the row and column subscripts with a comma.

We also use the notation a_j to correspond to the j^{th} column of the matrix A, and a_i^{T} to represent the vector that corresponds to the i^{th} row. The objects are vectors, but this notation does not uniquely identify the type of object, because we use the same notation for an element of a vector. The context, however, almost always makes the meaning clear.

The first row is the 1^{st} (first) row, and the first column is the 1^{st} (first) column. (Again, we remark that computer entities used in some systems to represent matrices and to store elements of matrices as computer data sometimes index the elements beginning with 0. Further, some systems use the first index to represent the column and the second index to indicate the row. We are not speaking here of the *storage order* — "row major" versus "column major" — we address that later. Rather, we are speaking of the mechanism of *referring to* the abstract entities. In image processing, for example, it is common practice to reverse use the first index to represent the column and the second index to represent the row. In the software package PV-Wave, for example, there are two different kinds of two-dimensional objects: arrays, in which the indexing is done as in image processing, and matrices, in which the indexing is done as we have described.)

The $n \times m$ matrix A can be written

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}.$$

We also write the matrix A above as

$$(a_{ij}),$$

with the indices i and j ranging over $\{1, 2, \ldots, n\}$ and $\{1, 2, \ldots, m\}$, respectively.

The vector space generated by the columns of the $n \times m$ matrix A is of order n and of dimension m or less, and is called the *column space* of A, the *range* of A, or the *manifold* of A. This vector space is often denoted by $\mathcal{V}(A)$ or by $\mathrm{span}(A)$.

We use a superscript "T" to denote the *transpose* of a matrix; thus, if $A = (a_{ij})$, then $A^{\mathrm{T}} = (a_{ji})$. (In other literature, the transpose is often denoted by a prime, as in $A' = (a_{ji})$.) If $A = A^{\mathrm{T}}$, A is said to be *symmetric*. A symmetric matrix is necessarily square. If the elements of the matrix are from the field of complex numbers, the *conjugate transpose* is a useful concept. We use a superscript "H" to denote the conjugate transpose of a matrix; thus, if $A = (a_{ij})$, then $A^{\mathrm{H}} = (\bar{a}_{ji})$, where $\bar{a}$ represents the conjugate of the complex number a. (The conjugate transpose is often denoted by an asterisk, as in $A^* = (\bar{a}_{ji})$. This notation is more common when a prime is used to denote transpose.) If $A = A^{\mathrm{H}}$, A is said to be *Hermitian*. A Hermitian matrix is square as is a symmetric matrix.

The a_{ii} elements of a matrix are called *diagonal elements*; an element, a_{ij}, with $i < j$ is said to be "above the diagonal", and one with $i > j$ is said to be "below the diagonal". The vector consisting of all of the a_{ii}'s is called the *principal diagonal*, or just the diagonal.

If all except the principal diagonal elements of a square matrix are 0, the matrix is called a *diagonal matrix*. If all elements below the diagonal are 0, the matrix is called an *upper triangular matrix*; and a *lower triangular matrix* is defined similarly. If all elements are 0 except $a_{i,i+c_k}$ for some small number of integers, c_k, the matrix is called a *band matrix* (or *banded matrix*). The elements $a_{i,i+c_k}$ are called "codiagonals". In many applications $c_k \in \{-w_l, -w_l + 1, \ldots, -1, 0, 1, \ldots, w_u - 1, w_u\}$. In such a case, w_l is called the *lower band width* and w_u is called the *upper band width*. These patterned matrices arise in solutions of differential equations and so are very important in applications of linear algebra. Although it is often the case that band matrices are symmetric, or at least have the same number of codiagonals that are nonzero, neither of these conditions always occurs in applications of band matrices.

A band matrix with lower and upper band width of 1, and such that all elements $a_{i,i\pm1}$ are nonzero, is called a matrix of type 2. It can be shown that the inverses of certain matrices arising in statistical applications are matrices of type 2 (see Graybill, 1983).

A diagonal matrix can be specified by listing the diagonal elements with the

"diag" constructor function that operates on a vector:

$$\mathrm{diag}\big((d_1, d_2, \ldots, d_n)\big) = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ & & \ddots & \\ 0 & 0 & \cdots & d_n \end{bmatrix}.$$

(Notice that the argument of diag is a vector; that is why there are two sets of parentheses in the expression above.) For an integer constant $c \neq 0$, a vector consisting of all of the $x_{i,i+c}$'s is also called a diagonal, or a "minor diagonal". These phrases are used with both square and nonsquare matrices.

If $a_{i,i+c_k} = d_{c_k}$, where d_{c_k} is constant for fixed c_k, the matrix is called a *Toeplitz* matrix:

$$\begin{bmatrix} d_0 & d_1 & d_2 & \cdots & d_{n-1} \\ d_{-1} & d_0 & d_1 & \cdots & d_{n-2} \\ & & & \ddots & \\ d_{-n+1} & d_{-n+2} & d_{-n+3} & \cdots & d_0 \end{bmatrix};$$

that is, a Toeplitz matrix is a matrix with constant codiagonals. A Toeplitz matrix may or may not be a band matrix (have many 0 codiagonals) and it may or may not be symmetric.

Because the matrices with special patterns are usually characterized by the locations of zeros and nonzeros, we often use an intuitive notation with `X` and `0` to indicate the pattern. Thus, a band matrix may be written as

$$\begin{bmatrix} \mathtt{X} & \mathtt{X} & \mathtt{0} & \cdots & \mathtt{0} & \mathtt{0} \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \cdots & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{X} & \mathtt{X} & \cdots & \mathtt{0} & \mathtt{0} \\ & & \ddots & \ddots & & \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \cdots & \mathtt{X} & \mathtt{X} \end{bmatrix},$$

In this notation `X` is not the same object each place it occurs. The `X` and `0` may also indicate "submatrices", which we discuss in the section on partitioned matrices.

It is sometimes useful to consider the elements of a matrix to be elements of a single vector. The most common way this is done is to string the columns of the matrix end-to-end into a vector. The "vec" function does this:

$$\mathrm{vec}(A) = (a_1^{\mathrm{T}}, a_2^{\mathrm{T}}, \ldots, a_m^{\mathrm{T}}),$$

where $(a_1, a_2, \ldots, a_m)$ are the column vectors of the matrix A. For a symmetric matrix A, with elements a_{ij}, the "vech" function stacks the unique elements into a vector:

$$\mathrm{vech}(A) = (a_{11}, a_{21}, a_{22}, a_{31}, \ldots, a_{m1}, \ldots, a_{mm}).$$

Henderson and Searle (1979) derive several interesting properties of vec and vech.

The sum of the diagonal elements of a square matrix is called the *trace* of the matrix. We use the notation "trace(A)" to denote the trace of the matrix A:

$$\text{trace}(A) = \sum_i a_{ii}.$$

For an $n \times n$ (square) matrix A, consider the product $a_{1j_1} a_{2j_2} \cdots a_{nj_n}$, where $j_1, j_2, \ldots, j_n$ is some permutation of the integers from 1 to n. Define a permutation to be *even* if the number of times that consecutive pairs have a larger first element is an even number, and define the permutation to be *odd* otherwise. (For example, 1,3,2 is an odd permutation; and 3,2,1 is an even permutation.) Let $\sigma(j_1, j_2, \ldots, j_n) = 1$ if $j_1, j_2, \ldots, j_n$ is an even permutation, and let $\sigma(j_1, j_2, \ldots, j_n) = -1$ otherwise. Then the *determinant* of A, denoted by "det(A)" is defined by:

$$\det(A) = \sum_{\text{all permutations}} \sigma(j_1, j_2, \ldots, j_n) a_{1j_1} a_{2j_2} \cdots a_{nj_n}.$$

The determinant is also sometimes written as $|A|$. The determinant of a triangular matrix is just the product of the diagonal elements.

For an arbitrary matrix, the determinant is more difficult to compute than is the trace. The method for computing a determinant is not the one that would arise directly from the definition given above; rather, it involves first decomposing the matrix, as we discuss in later sections. Neither the trace nor the determinant is very often useful in computations; but, although it may not be obvious from their definitions, both objects are very useful in establishing properties of matrices.

Useful, and obvious, properties of the trace and determinant are:

- $\text{trace}(A) = \text{trace}(A^{\mathrm{T}})$
- $\det(A) = \det(A^{\mathrm{T}})$

2.1.3 Operations on Vectors and Matrices

The elements of a vector or matrix are elements of a field; and, as we have seen, most matrix and vector operations are defined in terms of operations in the field.

The sum of two matrices of the same shape is the matrix whose elements are the sums of the corresponding elements of the addends. Addition of matrices is also indicated by "+", as with scalar and vector addition. We assume throughout that writing a sum of matrices, $A + B$, implies that they are of the same shape, that is, that they are *conformable for addition*. A scalar multiple of a matrix is the matrix whose elements are the multiples of the corresponding elements of the original matrix.

There are various kinds of multiplication of matrices that may be useful. If the number of columns of the matrix A, with elements a_{ij}, and the number of rows of the matrix B, with elements b_{ij}, are equal, then the *(Cayley) product* of A and B, is defined as the matrix C with elements

$$c_{ij} = \sum_k a_{ik} b_{kj}. \tag{2.2}$$

This is the most common type of product, and it is what we refer to by the unqualified phrase "matrix multiplication". Matrix multiplication is also indicated by juxtaposition, with no intervening symbol for the operation.

If the matrix A is $n \times m$ and the matrix B is $m \times p$, the product $C = AB$ is $n \times p$:

$$\begin{array}{ccc} C & = & A \quad B \\ \left[\quad\right]_{n\times p} & = & \left[\quad\right]_{n\times m} \left[\quad\right]_{m\times p} \end{array}$$

We assume throughout that writing a product of matrices AB implies that the number of columns of the first matrix is the same as the number of rows of the second, that is, they are *conformable for multiplication* in the order given.

It is obvious that while the product $C = AB$ may be well defined, the product BA is defined only if $n = p$, that is, if the matrices AB and BA are square. It is easy to see from the definition of matrix multiplication (2.2) that in general, even for square matrices, $AB \neq BA$. It is also obvious that if $C = AB$, then $B^{\mathrm{T}}A^{\mathrm{T}}$ exists and, in fact, $C^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$. If (but not only if) A and B are symmetric, then $AB = BA$; that is, the product of symmetric matrices is symmetric.

For a square matrix, its product with itself is defined; and so for a positive integer k, we write A^k to mean $k-1$ multiplications: $AA\cdots A$.

Here, as throughout the field of numerical analysis, we must remember that the definition of an operation, such as matrix multiplication, does not necessarily define a good algorithm for evaluating the operation.

Because matrix multiplication is not commutative, we often use the terms "premultiply" and "postmultiply", and the corresponding noun forms of these terms. Thus in the product AB, we may say B is premultiplied by A, or, equivalently, A is postmultiplied by B.

Although matrix multiplication is not *commutative*, it is *associative*; that is, if the matrices are conformable,

$$A(BC) = (AB)C;$$

and it is *distributive* over addition; that is,

$$A(B + C) = AB + AC.$$

Useful properties of the trace and determinant are:

- $\text{trace}(A + B) = \text{trace}(A) + \text{trace}(B)$
- $\det(AB) = \det(A)\det(B)$, if A and B are square matrices conformable for multiplication

Two additional properties of the trace, for the matrices A, B, and C that are conformable for the multiplications indicated, and such that the appropriate products are square, are

- $\text{trace}(AB) = \text{trace}(BA)$
- $\text{trace}(ABC) = \text{trace}(BCA) = \text{trace}(CAB)$

Three other types of matrix multiplication that are useful are *Hadamard multiplication*, *Kronecker multiplication*, and *dot product multiplication*. Hadamard multiplication is defined for matrices of the same shape as the multiplication of each element of one matrix by the corresponding element of the other matrix. Hadamard multiplication immediately inherits the commutativity, associativity, and distribution over addition of the ordinary multiplication of the underlying field of scalars. Hadamard multiplication is also called array multiplication and element-wise multiplication.

Kronecker multiplication, denoted by $\otimes$, is defined for any two matrices $A_{n\times m}$ and $B_{p\times q}$ as

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1m}B \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1}B & a_{n2}B & \cdots & a_{nm}B \end{bmatrix}.$$

The Kronecker product of A and B is $np \times mq$. Kronecker multiplication is also called "direct multiplication". Kronecker multiplication is associative and distributive over addition, but it is not commutative. A relationship between the vec function and Kronecker multiplication is

$$\text{vec}(ABC) = (C^{\mathrm{T}} \otimes A)\text{vec}(B),$$

for matrices A, B, and C that are conformable for the multiplication indicated.

The dot product of matrices is defined for matrices of the same shape as the sum of the dot products of the vectors formed from the columns of one matrix with vectors formed from the corresponding columns of the other matrix. The dot product of real matrices is a real number, as is the dot product of real vectors. The dot product of the matrices A and B with the same shape is denoted by $A \cdot B$, or $\langle A, B\rangle$, just as the dot product of vectors. For conformable matrices A, B, and C, the following properties of the dot product of matrices are straightforward:

- $\langle A, B\rangle = \langle B, A\rangle$

- $\langle A, B\rangle = \text{trace}(A^{\mathrm{T}}B)$
- $\langle A, A\rangle \geq 0$, with equality only if $A = 0$
- $\langle sA, B\rangle = s\langle A, B\rangle$, for a scalar s
- $\langle (A + B), C\rangle = \langle A, C\rangle + \langle B, C\rangle$

Dot products of matrices also obey the Cauchy-Schwarz inequality (compare (2.1), page 51):

$$\langle A, B\rangle \leq \langle A, A\rangle^{\frac{1}{2}} \langle B, B\rangle^{\frac{1}{2}}, \tag{2.3}$$

with equality holding only if $A = 0$ or $B = sA$, for some scalar s. This is easy to prove by the same argument as used for inequality (2.1) on page 51. (You are asked to write out the details in Exercise 2.3.)

It is often convenient to think of a vector as a matrix with the length of one dimension being 1. This provides for an immediate extension of the definition of matrix multiplication to include vectors as either or both factors. In this scheme, we adopt the convention that a vector corresponds to a *column*, that is, if x is a vector and A is a matrix, Ax or $x^{\mathrm{T}}A$ may be well-defined; but Ax^{T} would not represent anything, except in the case when all dimensions are 1. The dot product or inner product, $\langle c, x\rangle$, of the vectors x and y can be represented as $x^{\mathrm{T}}y$. The *outer product* of the vectors x and y is the matrix xy^{T}.

A variation of the vector dot product, $x^{\mathrm{T}}Ay$, is called a *bilinear form*, and the special bilinear form $x^{\mathrm{T}}Ax$ is called a *quadratic form*. Although in the definition of quadratic form we do not require A to be symmetric — because for a given value of x and a given value of the quadratic form, $x^{\mathrm{T}}Ax$, there is a unique symmetric matrix A_s such that $x^{\mathrm{T}}A_s x = x^{\mathrm{T}}Ax$ — we generally work only with symmetric matrices in dealing with quadratic forms. (The matrix A_s is $\frac{1}{2}(A + A^{\mathrm{T}})$. See Exercise 2.4.) Quadratic forms correspond to sums of squares, and, hence, play an important role in statistical applications.

2.1.4 Partitioned Matrices

We often find it useful to partition a matrix into submatrices, and we usually denote those submatrices with capital letters with subscripts indicating the relative positions of the submatrices. Hence, we may write

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

where the matrices A_{11} and A_{12} have the same number of rows, A_{21} and A_{22} have the same number of rows, A_{11} and A_{21} have the same number of columns, and A_{12} and A_{22} have the same number of columns. A submatrix that contains the $(1, 1)$ element of the original matrix is called a *principal submatrix*; A_{11} is a principal submatrix in the example above.

Multiplication and other operations with matrices, such as transposition, are carried out with their submatrices in the obvious way. Thus,

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} A_{11}^{\mathrm{T}} & A_{21}^{\mathrm{T}} \\ A_{12}^{\mathrm{T}} & A_{22}^{\mathrm{T}} \\ A_{13}^{\mathrm{T}} & A_{23}^{\mathrm{T}} \end{bmatrix},$$

and, assuming the submatrices are conformable for multiplication,

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Sometimes a matrix may be partitioned such that one partition is just a single column or row, that is, a vector or the transpose of a vector. In that case, we may use a notation such as

$$[X\ y]$$

or

$$[X \mid y],$$

where X is a matrix and y is a vector. We develop the notation in the obvious fashion; for example,

$$[X\ y]^{\mathrm{T}}[X\ y] = \begin{bmatrix} X^{\mathrm{T}}X & X^{\mathrm{T}}y \\ y^{\mathrm{T}}X & y^{\mathrm{T}}y \end{bmatrix}. \tag{2.4}$$

Partitioned matrices may also have useful patterns. A "block diagonal" matrix is one of the form

$$\begin{bmatrix} \mathtt{X} & \mathtt{0} & \cdots & \mathtt{0} \\ \mathtt{0} & \mathtt{X} & \cdots & \mathtt{0} \\ & & \ddots & \\ \mathtt{0} & \mathtt{0} & \cdots & \mathtt{X} \end{bmatrix},$$

where 0 represents a submatrix with all zeros, and X represents a general submatrix, with at least some nonzeros. The diag(·) function previously introduced for a vector is also defined for a list of matrices:

$$\mathrm{diag}(A_1, A_2, \ldots, A_k)$$

denotes the block diagonal matrix with submatrices $A_1, A_2, \ldots, A_k$ along the diagonal and zeros elsewhere.

2.1.5 Matrix Rank

The linear dependence or independence of the vectors forming the rows or columns of a matrix is an important characteristic of the matrix. The maximum

number of linearly independent vectors (either those forming the rows or the columns) is called the *rank* of the matrix. (We have used the term "rank" before to denote dimensionality of an array. "Rank" as we have just defined it applies only to a matrix or to a set of vectors. The meaning is clear from the context.) Although some people use the terms "row rank" or "column rank", the single word "rank" is sufficient because they are the same. It is obvious that the rank of a matrix can never exceed its smaller dimension. Whether or not a matrix has more rows than columns, the rank of the matrix is the same as the dimension of the column space of the matrix.

We use the notation "rank(A)" to denote the rank of the matrix A.

If the rank of a matrix is the same as its smaller dimension, we say the matrix is of *full rank*. In this case we may say the matrix is of full row rank or full column rank. A full rank matrix is also called *nonsingular*, and one that is not nonsingular is called *singular*. These words are often restricted to square matrices, and the phrase "full row rank" or "full column rank", as appropriate, is used to indicate that a nonsquare matrix is of full rank.

In practice, it is not always clear whether a matrix is nonsingular. Because of rounding on the computer, a matrix that is mathematically nonsingular may appear to be singular. We sometimes use the phrase "nearly singular" or "algorithmically singular" to describe such a matrix. In general, the numerical determination of the rank of a matrix is not an easy task.

The rank of the product of two matrices is less than or equal to the lesser of the ranks of the two:

$$\text{rank}(AB) \leq \min\{\text{rank}(A), \text{rank}(B)\}.$$

The rank of an outer product matrix is 1.

For a square matrix A, $\det(A) = 0$ if and only if A is singular.

2.1.6 Identity Matrices

An $n \times n$ matrix consisting of 1's along the diagonal and 0's everywhere else is a *multiplicative identity* for the set of $n \times n$ matrices and Cayley multiplication. Such a matrix is called the *identity matrix of order* n, and is denoted by I_n, or just by I. If A is $n \times m$, then $I_n A = A I_m = A$. The identity matrix is a multiplicative identity for any matrix so long as the matrices are conformable for the multiplication.

The columns of the identity matrix are called *unit vectors*. The i^{th} unit vector, denoted by e_i, has a 1 in the i^{th} position and 0's in all other positions:

$$e_i = (0, \ldots, 0, 1, 0, \ldots, 0).$$

(There is an implied number of elements of a unit vector that is inferred from the context. Also parenthetically, we remark that the phrase "unit vector" is sometimes used to refer to a vector the sum of whose squared elements is 1, that is, whose length, in the Euclidean distance sense, is 1. We refer to vectors with length of 1 as "normalized vectors".)

Identity matrices for Hadamard and Kronecker multiplication are of less interest. The identity for Hadamard multiplication is the matrix of appropriate shape whose elements are all 1's. The identity for Kronecker multiplication is the 1×1 matrix with the element 1; that is, it is the same as the scalar 1.

2.1.7 Inverses

The elements in a set that has an identity with respect to some operation may have inverses with respect to that operation. The only type of matrix multiplication for which an inverse is of common interest is Cayley multiplication of square matrices. The inverse of the $n \times n$ matrix A is the matrix A^{-1} such that

$$A^{-1}A = AA^{-1} = I_n.$$

A matrix has an inverse if and only if the matrix is square and of full rank.

As we have indicated, important applications of vectors and matrices involve systems of linear equations:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + \cdots + & a_{1m}x_m & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + \cdots + & a_{2m}x_m & = & b_2 \\ \vdots & + & \vdots & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + \cdots + & a_{nm}x_m & = & b_n \end{array} \tag{2.5}$$

An objective with such a system is to determine x's that satisfy these equations for given a's and b's. In vector/matrix notation, these equations are written as

$$Ax = b,$$

and if $n = m$ and A is nonsingular, the solution is

$$x = A^{-1}b.$$

We discuss the solution of systems of equations in Chapter 3.

If A is nonsingular, and can be partitioned as

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

where both A_{11} and A_{22} are nonsingular, it is easy to see (Exercise 2.5, page 85) that the inverse of A is given by

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}Z^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}Z^{-1} \\ -Z^{-1}A_{21}A_{11}^{-1} & Z^{-1} \end{bmatrix}, \tag{2.6}$$

where $Z = A_{22} - A_{21}A_{11}^{-1}A_{12}$. In this partitioning Z is called the *Schur complement* of A_{11} in A. If

$$A = [Xy]^{\mathrm{T}}[Xy]$$

and is partitioned as in (2.4) on page 59, then the Schur complement of $X^{\mathrm{T}}X$ in $[Xy]^{\mathrm{T}}[Xy]$ is

$$y^{\mathrm{T}}y - y^{\mathrm{T}}X(X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y.$$

This particular partitioning is useful in linear regression analysis, where this Schur complement is the residual sum of squares.

Often in linear regression analysis we need inverses of various sums of matrices. This is often because we wish to update regression estimates based on additional data or because we wish to delete some observations. If A and B are full rank matrices of the same size, the following relationships are easy to show. (They are easily proven if taken in the order given.)

$$\begin{aligned}
(I + A^{-1})^{-1} &= A(A + I)^{-1} \\
(A + BB^{\mathrm{T}})^{-1}B &= A^{-1}B(I + B^{\mathrm{T}}A^{-1}B)^{-1} \\
(A^{-1} + B^{-1})^{-1} &= A(A + B)^{-1}B \\
A - A(A + B)^{-1}A &= B - B(A + B)^{-1}B \\
A^{-1} + B^{-1} &= A^{-1}(A + B)B^{-1} \\
(I + AB)^{-1} &= I - A(I + BA)^{-1}B \\
(I + AB)^{-1}A &= A(I + BA)^{-1}
\end{aligned} \tag{2.7}$$

From the relationship $\det(AB) = \det(A)\det(B)$ mentioned earlier, it is easy to see that for nonsingular A,

$$\det(A) = 1/\det(A^{-1}).$$

2.1.8 Linear Systems

Often in statistical applications, the number of equations in the system (2.5) is not equal to the number of variables. If $n > m$ and $\mathrm{rank}([A \mid b]) > \mathrm{rank}(A)$, the system is said to be *overdetermined*. There is no x that satisfies such a system, but approximate solutions are useful. We discuss approximate solutions of such systems in Sections 3.7 and 6.2.

A system (2.5) for which

$$\mathrm{rank}([A \mid b]) = \mathrm{rank}(A)$$

is said to be *consistent*. A consistent system has a solution. Furthermore, any system admitting a solution is consistent. The square system in which A is nonsingular, for example, is clearly consistent.

The vector space generated by all solutions, x, of the system

$$Ax = 0$$

is called the *null space* of the $n \times m$ matrix A. The dimension of the null space is $n - \mathrm{rank}(A)$.

A consistent system in which $n < m$ is said to be *underdetermined.* For such a system there will be more than one solution. In fact, there will be infinitely many solutions, because if the vectors x_1 and x_2 are solutions, the vector $wx_1 + (1 - w)x_2$ is likewise a solution for any scalar w.

Underdetermined systems arise in analysis of variance in statistics, and it is useful to have a compact method of representing the solution to the system. It is also desirable to identify a unique solution that has some kind of optimal properties.

2.1.9 Generalized Inverses

Suppose the system $Ax = b$ is consistent, and A^- is any matrix such that $AA^-A = A$. Then $x = A^-b$ is a solution to the system. Furthermore, if Gb is any solution, then $AGA = A$. The former statement is true because if $AA^-A = A$, then $AA^-Ax = Ax$ and since $Ax = b$, $AA^-b = b$. The latter statement can be seen by the following argument. Let a_j be the j^{th} column of A. The m systems of n equations, $Ax = a_j$, $j = 1, \ldots, m$, all have solutions (a vector with 0's in all positions except the j^{th} position, in which is a 1). Now, if Gb is a solution to the original system, then Ga_j is a solution to the system $Ax = a_j$. So $AGa_j = a_j$ for all j; hence $AGA = A$.

A matrix A^- such that $AA^-A = A$ is called a *generalized inverse* or a g_1 *inverse* of A. A g_1 inverse is not unique, but if we impose three more conditions we arrive at a unique matrix, denoted by A^+, that yields a solution that has some desirable properties. (For example, the length of A^+b, in the sense of the Euclidean distance, is the smallest of any solution to $Ax = b$. See Section 3.7.)

For matrix A, the conditions that yield a unique generalized inverse, called the *Moore-Penrose inverse*, and denoted by A^+, are

1. $AA^+A = A$ (i.e., it is a g_1 inverse).
2. $A^+AA^+ = A^+$. (A g_1 inverse that satisfies this condition is called a g_2 *inverse*, and is denoted by A^*.)
3. A^+A is symmetric.
4. AA^+ is symmetric.

(The name derives from the work of Moore and Penrose in E. H. Moore, 1920, "On the reciprocal of the general algebraic matrix," *Bulletin of the American Mathematical Society*, **26**, 394–395, and R. Penrose, 1955, "A generalized inverse for matrices," *Proceedings of the Cambridge Philosophical Society*, **51**, 406–413.) The Moore-Penrose inverse is also called the *pseudoinverse* or the g_4 *inverse.*

For any matrix A, the Moore-Penrose inverse exists and is unique. If A is nonsingular, obviously $A^+ = A^{-1}$.

If A is partitioned as

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

then, similarly to equation (2.6), a generalized inverse of A is given by

$$A^- = \begin{bmatrix} A_{11}^- + A_{11}^- A_{12} Z^- A_{21} A_{11}^- & -A_{11}^- A_{12} Z^- \\ -Z^- A_{21} A_{11}^- & Z^- \end{bmatrix}, \tag{2.8}$$

where $Z = A_{22} - A_{21} A_{11}^- A_{12}$ (see Exercise 2.6, page 85).

2.1.10 Other Special Vectors and Matrices

There are a number of other special vectors and matrices that are useful in numerical linear algebra. The geometric property of the angle between vectors has important implications for certain operations, both because it may indicate that rounding will have deleterious effects and because it may indicate a deficiency in the understanding of the application. Two vectors, v_1 and v_2, whose dot product is 0 are said to be *orthogonal*, written $v_1 \perp v_2$ because this is equivalent to the corresponding geometric property. (Sometimes we exclude the zero vector from this definition, but it is not important to do so.) A vector whose dot product with itself is 1, is said to be *normalized*. (The word "normal" is also used to denote this property, but because this word is used to mean several other things, "normalized" is preferred.) Normalized vectors that are all orthogonal to each other are called *orthonormal* vectors. (If the elements of the vectors are from the field of complex numbers, orthogonality and normality are defined in terms of the dot products of a vector with a complex conjugate of a vector.)

A set of vectors that are mutually orthogonal are necessarily linearly independent. A basis for a vector space is often chosen to be an orthonormal set.

All vectors in the null space of the matrix A are orthogonal to all vectors in the column space of A. In general, two vector spaces $\mathcal{V}_1$ and $\mathcal{V}_2$ are said to be *orthogonal*, written $\mathcal{V}_1 \perp \mathcal{V}_2$, if each vector in one is orthogonal to every vector in the other. The intersection of two orthogonal vector spaces consists only of the zero vector. If $\mathcal{V}_1 \perp \mathcal{V}_2$ and $\mathcal{V}_1 \oplus \mathcal{V}_2 = \mathbb{R}^n$, then $\mathcal{V}_2$ is called the *orthogonal complement* of $\mathcal{V}_1$, and this is written as $\mathcal{V}_2 = \mathcal{V}_1^\perp$. The null space of the matrix A is the *orthogonal complement* of A^{T}.

Instead of defining orthogonality in terms of dot products, we can define it more generally in terms of a bilinear form. If the bilinear form $x^{\mathrm{T}} A y = 0$, we say x and y are orthogonal with respect to the matrix A. In this case we often use a different term, and say that the vectors are *conjugate* with respect to A. The usual definition of orthogonality in terms of a dot product is equivalent to the definition in terms of a bilinear form in the identity matrix.

A matrix whose rows or columns constitute a set of orthonormal vectors is said to be an *orthogonal* matrix. If Q is an $n \times m$ matrix, then $QQ^{\mathrm{T}} = I_n$ if $n \leq m$, and $Q^{\mathrm{T}} Q = I_m$ if $n \geq m$. Such a matrix is also called a *unitary matrix*. (For matrices whose elements are complex numbers, a matrix is said to be *unitary* if the matrix times its conjugate transpose is the identity, that is, if

$QQ^H = I$. Both of these definitions are in terms of orthogonality of the rows or columns of the matrix.)

The determinant of a square orthogonal matrix is 1.

The definition given above for orthogonal matrices is sometimes relaxed to require only that the columns or rows be orthogonal (rather than also normal). If normality is not required, the determinant is not necessarily 1. If Q is a matrix that is "orthogonal" in this weaker sense of the definition, and Q has more rows than columns, then

$$Q^{\mathrm{T}}Q = \begin{bmatrix} \mathtt{X} & \mathtt{0} & \cdots & \mathtt{0} \\ \mathtt{0} & \mathtt{X} & \cdots & \mathtt{0} \\ & & \ddots & \\ \mathtt{0} & \mathtt{0} & \cdots & \mathtt{X} \end{bmatrix}.$$

The definition of orthogonality is also sometimes made more restrictive to require that the matrix be square.

In the course of performing computations on a matrix, it is often desirable to interchange the rows or columns of the matrix. Interchange of two rows of a matrix can be accomplished by premultiplying the matrix by a matrix that is the identity with those same two rows interchanged. For example,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{23} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}.$$

The first matrix in the expression above is called an *elementary permutation matrix*. It is the identity matrix with its second and third rows (or columns) interchanged. An elementary permutation matrix that is the identity with the j^{th} and k^{th} rows interchanged is denoted by E_{jk}. That is, E_{jk} is the identity, except the j^{th} row is e_k^{T} and the k^{th} row is e_j^{T}. Note $E_{jk} = E_{kj}$. Thus, for example,

$$E_{23} = E_{32} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Premultiplying a matrix A by a (conformable) E_{jk} results in an interchange of the j^{th} and k^{th} rows of A as we see above. Postmultiplying a matrix A by a (conformable) E_{jk} results in an interchange of the j^{th} and k^{th} columns of A:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{13} & a_{12} \\ a_{21} & a_{23} & a_{22} \\ a_{31} & a_{33} & a_{32} \\ a_{41} & a_{43} & a_{42} \end{bmatrix}.$$

It is easy to see from the definition that an elementary permutation matrix is symmetric and orthogonal. A more general permutation matrix can be built as

the product of elementary permutation matrices. Such a matrix is not necessarily symmetric, but its transpose is also a permutation matrix. A permutation matrix is orthogonal.

A special, often useful vector is the *sign vector*, which is formed from signs of the elements of a given vector. It is denoted by "sign(·)", and defined by

$$\begin{aligned} \text{sign}(x)_i &= 1 \quad \text{if } x_i > 0, \\ &= 0 \quad \text{if } x_i = 0, \\ &= -1 \quad \text{if } x_i < 0. \end{aligned}$$

A matrix A such that $AA = A$ is called an *idempotent matrix*. An idempotent matrix is either singular or it is the identity matrix.

For a given vector space $\mathcal{V}$, a symmetric idempotent matrix A whose columns span $\mathcal{V}$ is said to be an *orthogonal projection matrix* onto $\mathcal{V}$. It is easy to see that for any vector x, the vector Ax is in $\mathcal{V}$ and $x - Ax$ is in $\mathcal{V}^\perp$ (the vectors Ax and $x - Ax$ are orthogonal). A matrix is a projection matrix if and only if it is symmetric and idempotent. Two matrices that occur in linear regression analysis (see Section 6.2, page 163),

$$X(X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}$$

and

$$I - X(X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}},$$

are projection matrices. The first matrix above is called the "hat matrix" because it projects the observed response vector, often denoted by y, onto a predicted response vector, often denoted by $\widehat{y}$. In geometrical terms, the second matrix above projects a vector from the space spanned by the columns of X onto a set of vectors that constitute what is called the residual vector space. Incidentally, it is obvious that if A is a projection matrix, $I - A$ is also a projection matrix, and

$$\text{span}(I - A) = \text{span}(A)^\perp.$$

A symmetric matrix A such that for any (conformable) vector $x \neq 0$, the quadratic form

$$x^{\mathrm{T}}Ax > 0,$$

is called a *positive definite matrix*. A positive definite matrix is necessarily nonsingular. There are two related terms, *positive semidefinite matrix* and *nonnegative definite matrix*, which are not used consistently in the literature. In this text, we use the term *nonnegative definite matrix* for any symmetric matrix A for which for any (conformable) vector x, the quadratic form $x^{\mathrm{T}}Ax$ is nonnegative, that is,

$$x^{\mathrm{T}}Ax \geq 0.$$

(Some authors call this "positive semidefinite", but other authors use the term "positive semidefinite" to refer to a nonnegative definite matrix that is not positive definite.)

It is obvious from the definitions that any square submatrix whose principal diagonal is a subset of the principal diagonal of a positive definite matrix is positive definite, and similarly for nonnegative definite matrices. In particular, any square principal submatrix of a positive definite matrix is positive definite.

The *Helmert matrix* is an orthogonal matrix that partitions sums of squares. The Helmert matrix of order n has the form

$$
\begin{aligned}
H_n &= \begin{bmatrix}
n^{-\frac{1}{2}} & n^{-\frac{1}{2}} & n^{-\frac{1}{2}} & \cdots & n^{-\frac{1}{2}} \\
1/\sqrt{2} & -1/\sqrt{2} & 0 & \cdots & 0 \\
1/\sqrt{6} & 1/\sqrt{6} & -2/\sqrt{6} & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\frac{1}{\sqrt{n(n-1)}} & \frac{1}{\sqrt{n(n-1)}} & \frac{1}{\sqrt{n(n-1)}} & \cdots & -\frac{(n-1)}{\sqrt{n(n-1)}}
\end{bmatrix} \\
&= \begin{bmatrix} n^{-\frac{1}{2}} 1^{\mathrm{T}} \\ K_{n-1} \end{bmatrix}.
\end{aligned}
$$

For the n-vector x, with $\bar{x} = 1^{\mathrm{T}}x/n$,

$$
\begin{aligned}
s_x^2 &= \sum (x_i - \bar{x})^2 \\
&= x^{\mathrm{T}} K_{n-1}^{\mathrm{T}} K_{n-1} x.
\end{aligned}
$$

Obviously, the sums of squares are never computed by forming the Helmert matrix explicitly and then computing the quadratic form, but the computations in partitioned Helmert matrices are performed indirectly in analysis of variance.

2.1.11 Eigenanalysis

If A is an $n \times n$ (square) matrix, v is a vector not equal to 0, and λ is a scalar, such that

$$Av = \lambda v, \tag{2.9}$$

then v is called an *eigenvector* of the matrix A and λ is called an *eigenvalue* of the matrix A. An eigenvalue is also called a *singular value*, a *latent root*, a *characteristic value*, or a *proper value*, and similar synonyms exist for an eigenvector. The set of all the eigenvalues of a matrix is called the *spectrum* of the matrix.

An eigenvalue of A is a root of the *characteristic equation*,

$$\det(A - \lambda I) = 0, \tag{2.10}$$

which is a polynomial of degree n or less.

The number of nonzero eigenvalues is equal to the rank of the matrix. All eigenvalues of a positive definite matrix are positive, and all eigenvalues of a nonnegative definite matrix are nonnegative.

It is easy to see that any scalar multiple of an eigenvector of A is likewise an eigenvector of A. It is often desirable to scale an eigenvector v so that $v^{\mathrm{T}}v = 1$. Such a normalized eigenvector is also called a *unit eigenvector*.

Because most of the matrices in statistical applications are real, in the following we will generally restrict our attention to real matrices. It is important to note that the eigenvalues and eigenvectors of a real matrix are not necessarily real. They are real if the matrix is symmetric, however.

If λ is an eigenvalue of a real matrix A, we see immediately from the definition or from (2.10) that

- $c\lambda$ is an eigenvalue of cA,
- λ^2 is an eigenvalue of A^2,
- λ is an eigenvalue of A^{T} (the eigenvectors of A^{T}, however, are not the same as those of A),
- $\bar{\lambda}$ is an eigenvalue of A (where $\bar{\lambda}$ is the complex conjugate of λ),
- $\lambda\bar{\lambda}$ is an eigenvalue of $A^{\mathrm{T}}A$,
- λ is real if A is symmetric, and
- $1/\lambda$ is an eigenvalue of A^{-1}, if A is nonsingular.

If V is a matrix whose columns correspond to the eigenvectors of A and Λ is a diagonal matrix whose entries are the eigenvalues corresponding to the columns of V, then it is clear from equation (2.9) that

$$AV = V\Lambda.$$

If V is nonsingular,

$$A = V\Lambda V^{-1}. \tag{2.11}$$

Expression (2.11) represents a *factorization of the matrix* A. This representation is sometimes called the *similar canonical form* of A.

Not all matrices can be factored as in (2.11). If a matrix can be factored as in (2.11), it is called a *simple matrix* or a *regular matrix*; a matrix that cannot be factored in that way is called a *deficient matrix* or a *defective matrix*. Any symmetric matrix or any matrix all of whose eigenvalues are unique is regular, or simple. For a matrix to be simple, however, it is not necessary that it either be square or have all unique eigenvalues.

If m eigenvalues are equal to each other, that is, a single value occurs as a root of the characteristic equation (2.10) m times, we say the eigenvalue has *multiplicity* m. The necessary and sufficient condition for a matrix to be simple can be stated in terms of the unique eigenvalues and their multiplicities. Suppose for the $n \times n$ matrix A, the distinct eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_k$ have multiplicities $m_1, m_2, \ldots, m_k$. If, for $i = 1, \ldots, k$,

$$\mathrm{rank}(A - \lambda_i I) = n - m_i$$

then A is simple; this condition is also necessary for A to be simple.

The factorization (2.11) implies that the eigenvectors of a simple matrix are linearly independent.

If A is symmetric, the eigenvectors are orthogonal to each other. Actually, for an eigenvalue with multiplicity m, there are m eigenvectors that are linearly independent of each other, but without uniquely determined directions. Any vector in the space spanned by these vectors is an eigenvector, and there is a set of m orthogonal eigenvectors. In the case of a symmetric A, the V in (2.11) can be chosen to be orthogonal, and so the similar canonical form for a symmetric matrix can be chosen as $V\Lambda V^{\mathrm{T}}$.

When A is symmetric, and the eigenvectors v_i are chosen to be orthonormal,

$$I = \sum_i v_i v_i^{\mathrm{T}},$$

so

$$\begin{aligned} A &= A\sum_i v_i v_i^{\mathrm{T}} \\ &= \sum_i A v_i v_i^{\mathrm{T}} \\ &= \sum_i \lambda_i v_i v_i^{\mathrm{T}}. \end{aligned} \tag{2.12}$$

This representation is called the *spectral decomposition* of A. It also applies to powers of A:

$$A^k = \sum_i \lambda_i^k v_i v_i^{\mathrm{T}},$$

where k is an integer. If A is nonsingular, k can be negative in the expression above.

An additional factorization applicable to nonsquare matrices is the *singular value decomposition* (SVD). For the $n \times m$ matrix A, this factorization is

$$A = U\Sigma V^{\mathrm{T}}, \tag{2.13}$$

where U is an $n \times n$ orthogonal matrix, V is an $m \times m$ orthogonal matrix, and Σ is a diagonal matrix with nonnegative entries. The elements of Σ are called the *singular values* of A. All matrices have a factorization of the form (2.13). Forming the diagonal matrix $\Sigma^{\mathrm{T}}\Sigma$ or $\Sigma\Sigma^{\mathrm{T}}$, and using the factorization in equation (2.11), it is easy to see that the nonzero singular values of A are the square roots of the nonzero eigenvalues of symmetric matrix $A^{\mathrm{T}}A$ (or AA^{T}). If A is square, the singular values are the eigenvalues.

2.1.12 Similarity Transformations

Two $n \times n$ matrices, A and B, are said to be *similar* if there exists a nonsingular matrix P such that

$$A = P^{-1}BP. \tag{2.14}$$

The transformation in (2.14) is called a *similarity transformation.* It is clear from this definition that the similarity relationship is both commutative and transitive. We see from equation (2.11) that a matrix A with eigenvalues $\lambda_1, \ldots, \lambda_n$ is similar to the matrix $\mathrm{diag}(\lambda_1, \ldots, \lambda_n)$.

If A and B are similar, as in (2.14), then

$$\begin{aligned} B - \lambda I &= P^{-1}BP - \lambda P^{-1}IP \\ &= A - \lambda I, \end{aligned}$$

and, hence, A and B have the same eigenvalues. This fact also follows immediately from the transitivity of the similarity relationship and the fact that a matrix is similar to the diagonal matrix formed from its eigenvalues.

An important type of similarity transformation is based on an orthogonal matrix. If Q is orthogonal and

$$A = Q^{\mathrm{T}}BQ,$$

A and B are said to be *orthogonally similar.*

Similarity transformations are used in algorithms for computing eigenvalues (see, for example, Section 4.2).

2.1.13 Norms

For a set of objects S that has an addition-type operator, $+_S$, a corresponding additive identity, 0_S, and a scalar multiplication, that is, a multiplication of the objects by a real (or complex) number, a *norm* is a function, $\|\cdot\|$, from S to the reals that satisfies the following three conditions.

1. Nonnegativity and mapping of the identity:
   if $x \neq 0_S$, then $\|x\| > 0$, and $\|0_S\| = 0$

2. Relation of scalar multiplication to real multiplication:
   $\|ax\| = |a|\|x\|$ for real a

3. Triangle inequality:
   $\|x +_S y\| \leq \|x\| + \|y\|$

Sets of various types of objects (functions, for example) can have norms, but our interest in the present context is in norms for vectors and matrices. For vectors, 0_S is the zero vector (of the appropriate length) and $+_S$ is vector addition (which implies that the vectors are of the same length).

The triangle inequality suggests the origin of the concept of a norm. It clearly has its roots in vector spaces. For some types of objects the norm of an object may be called its "length" or its "size". (Recall the ambiguity of "length" of a vector that we mentioned at the beginning of this chapter.)

There are many norms that could be defined for vectors. One type of norm is called an L_p norm, often denoted as $\|\cdot\|_p$. For $p \geq 1$, it is defined as

$$\|x\|_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}} \tag{2.15}$$

It is easy to see that this satisfies the first two conditions above. For general $p \geq 1$ it is somewhat more difficult to prove the triangular inequality (which for the L_p norms is also called the Minkowski inequality), but for some special cases it is straightforward, as we see below.

The most common L_p norms, and in fact, the most commonly used vector norms, are:

- $\|x\|_1 = \sum_i |x_i|$, also called the Manhattan norm because it corresponds to sums of distances along coordinate axes, as one would travel along the rectangular street plan of Manhattan.
- $\|x\|_2 = \sqrt{\sum_i x_i^2} = \sqrt{\langle x, x\rangle}$, also called the Euclidean norm, or the vector length.
- $\|x\|_\infty = \max_i |x_i|$, also called the max norm.

The L_∞ norm is defined by taking the limit in an L_p norm. An L_p norm is also called a p-norm, or 1-norm, 2-norm, or ∞-norm in those special cases.

The triangle inequality is obvious for the L_1 and L_∞ norms. For the L_2 norm it can be shown using the Cauchy-Schwarz inequality (2.1), page 51. The triangle inequality for the L_2 norm on vectors is

$$\sqrt{\sum (x_i + y_i)^2} \leq \sqrt{\sum x_i^2} + \sqrt{\sum y_i^2} \tag{2.16}$$

or

$$\sum (x_i + y_i)^2 \leq \sum x_i^2 + 2\sqrt{\sum x_i^2}\sqrt{\sum y_i^2} + \sum y_i^2.$$

Now,

$$\sum (x_i + y_i)^2 = \sum x_i^2 + 2\sum x_i y_i + \sum y_i^2,$$

and by the Cauchy-Schwartz inequality,

$$\sum x_i y_i \leq \sqrt{\sum x_i^2}\sqrt{\sum y_i^2},$$

so the triangle inequality follows.

The L_p vector norms have the relationship,

$$\|x\|_1 \geq \|x\|_2 \geq \|x\|_\infty, \tag{2.17}$$

for any vector x.

The L_2 norm of a vector is the square root of the quadratic form of the vector with respect to the identity matrix. A generalization, called an *elliptic norm* for the vector x, is defined as the square root of the quadratic form $x^{\mathrm{T}}Ax$, for any symmetric positive-definite matrix A. It is easy to see that $\sqrt{x^{\mathrm{T}}Ax}$ satisfies the definition of a norm given earlier.

2.1.14 Matrix Norms

A matrix norm is required to have another property in addition to the three general properties on page 70 that define a norm in general. A matrix norm must also satisfy the *consistency property*:

4. $\|AB\| \leq \|A\| \, \|B\|$,

where AB represents the usual Cayley product of the conformable matrices A and B.

A matrix norm is often defined in terms of a vector norm. Given the vector norm $\|\cdot\|_v$, the matrix norm $\|\cdot\|_M$ *induced* by $\|\cdot\|_v$ is defined by

$$\|A\|_M = \max_{x \neq 0} \frac{\|Ax\|_v}{\|x\|_v}. \tag{2.18}$$

It is easy to see that an induced norm is indeed a matrix norm (i.e., that it satisfies the consistency property). We usually drop the v or M subscript and the notation $\|\cdot\|$ is overloaded to mean either a vector or matrix norm. Matrix norms are somewhat more complicated than vector norms because, for matrices that are not square, there is a dependence of the definition of the norm on the shape of the matrix.

The induced norm of A given in equation (2.18) is sometimes called the *maximum magnification* by A. The expression looks very similar to the maximum eigenvalue, and indeed it is in some cases.

For any vector norm and its induced matrix norm it is easy to see that

$$\|Ax\| \leq \|A\| \, \|x\|. \tag{2.19}$$

The matrix norms that correspond to the L_p vector norms are defined for the matrix A as

$$\|A\|_p = \max_{\|x\|_p = 1} \|Ax\|_p. \tag{2.20}$$

(Notice that the restriction on $\|x\|_p$ makes this an induced norm as defined in equation (2.18). Notice also the overloading of the symbols; the norm on the left that is being defined is a matrix norm, whereas those on the right of the equation are vector norms.) It is clear that the L_p norms satisfy the consistency property, because they are induced norms.

The L_1 and L_∞ norms have interesting simplifications:

- $\|A\|_1 = \max_j \sum_i |a_{ij}|$, also called the "column-sum norm", and
- $\|A\|_\infty = \max_i \sum_j |a_{ij}|$, also called the "row-sum norm".

Alternative formulations of the L_2 norm of a matrix are not so obvious from (2.20). It is related to the eigenvalues (or the singular values) of the matrix. For a square matrix A, the squared L_2 norm is the maximum eigenvalue of $A^{\mathrm{T}}A$.

The L_p matrix norms do not satisfy inequalities (2.17) for the L_p vector norms.

For the $n \times n$ matrix A, with eigenvalues, $\lambda_1, \lambda_2, \ldots, \lambda_n$, the maximum, $\max |\lambda_i|$, is called the *spectral radius*, and is denoted by $\rho(A)$:

$$\rho(A) = \max |\lambda_i|.$$

It can be shown (see Exercise 2.10, page 85) that

$$\|A\|_2 = \sqrt{\rho(A^{\mathrm{T}}A)}.$$

If A is symmetric

$$\|A\|_2^2 = \rho(A).$$

The spectral radius is a measure of the condition of a matrix for certain iterative algorithms. The L_2 matrix norm is also called the spectral norm.

For Q orthogonal, the L_2 norm has the important property,

$$\|Qx\|_2 = \|x\|_2 \tag{2.21}$$

(see Exercise 2.15a, page 86). For this reason, an orthogonal matrix is sometimes called an *isometric matrix*. By proper choice of x, it is easy to see from (2.21) that

$$\|Q\|_2 = 1. \tag{2.22}$$

These properties do not in general hold for other norms.

The L_2 matrix norm is a Euclidean-type norm since it is based on the Euclidean vector norm, but a different matrix norm is often called the Euclidean matrix norm. This is the *Frobenius norm*:

$$\|A\|_F = \sqrt{\sum_{i,j} a_{ij}^2}.$$

It is easy to see that the Frobenius norm has the consistency property and that for any square matrix A with real elements

$$\|A\|_2 \leq \|A\|_F.$$

(See Exercises 2.12 and 2.13, page 85.) A useful property of the Frobenius norm, which is obvious from the definition above, is

$$\begin{aligned} \|A\|_F &= \sqrt{\mathrm{trace}(A^{\mathrm{T}}A)} \\ &= \sqrt{\langle A, A \rangle}. \end{aligned}$$

If A and B are orthogonally similar, then

$$\|A\|_F = \|B\|_F.$$

To see this, let $A = Q^{\mathrm{T}}BQ$, where Q is an orthogonal matrix. Then

$$\begin{aligned} \|A\|_F^2 &= \mathrm{trace}(A^{\mathrm{T}}A) \\ &= \mathrm{trace}(Q^{\mathrm{T}}B^{\mathrm{T}}QQ^{\mathrm{T}}BQ) \\ &= \mathrm{trace}(B^{\mathrm{T}}BQQ^{\mathrm{T}}) \\ &= \mathrm{trace}(B^{\mathrm{T}}B) \\ &= \|B\|_F^2 \end{aligned}$$

(The norms are nonnegative, of course.)

2.1.15 Orthogonal Transformations

In the previous section we observed some interesting properties of orthogonal matrices. From equation (2.21), we see that orthogonal transformations preserve lengths.

If Q is orthogonal, for vectors x and y, we have

$$\langle Qx, Qy\rangle = (xQ)^{\mathrm{T}}(Qy) = x^{\mathrm{T}}Q^{\mathrm{T}}Qy = x^{\mathrm{T}}y = \langle x, y\rangle,$$

hence,

$$\arccos\left(\frac{\langle Qx, Qy\rangle}{\|Qx\|_2\,\|Qy\|_2}\right) = \arccos\left(\frac{\langle x, y\rangle}{\|x\|_2\ \|y\|_2}\right).$$

Thus we see that orthogonal transformations preserve angles.

From equation (2.22) we see $\|Q^{-1}\|_2 = 1$, and thus $\kappa_2(Q) = 1$ for the orthogonal matrix Q. This means use of computations with orthogonal matrices will not make problems more ill-conditioned.

It is easy to see from (2.21) that if A and B are orthogonally similar,

$$\kappa_2(A) = \kappa_2(B).$$

Later we use orthogonal transformations that preserve lengths and angles while reflecting regions of $\mathbb{R}^n$, and others that rotate $\mathbb{R}^n$. The transformations are appropriately called reflectors and rotators, respectively.

2.1.16 Orthogonalization Transformations

Given two nonnull, linearly independent vectors, x_1 and x_2, it is easy to form two orthonormal vectors, $\tilde{x}_1$ and $\tilde{x}_2$, that span the same space:

$$\begin{aligned} \tilde{x}_1 &= \frac{x_1}{\|x_1\|_2} \\ \tilde{x}_2 &= \frac{(x_2 - \tilde{x}_1^{\mathrm{T}}x_2\tilde{x}_1)}{\|x_2 - \tilde{x}_1^{\mathrm{T}}x_2\tilde{x}_1\|_2}. \end{aligned} \tag{2.23}$$

These are called *Gram-Schmidt transformations*. They can easily be extended to more than two vectors. The Gram-Schmidt transformations are the basis for other computations we will discuss in Section 3.2, on page 102.

2.1.17 Condition of Matrices

Data are said to be "ill-conditioned" for a particular computation if the data were likely to cause problems in the computations, such as severe loss of precision. More generally, the term "ill-conditioned" is applied to a problem in which small changes to the input result in large changes in the output. In the case of a linear system

$$Ax = b$$

the problem of solving the system is ill-conditioned if small changes to some elements of A or of b will cause large changes in the solution x.

Consider, for example, the system of equations

$$\begin{aligned} 1.000x_1 + 0.500x_2 &= 1.500 \\ 0.667x_1 + 0.333x_2 &= 1.000 \end{aligned} \tag{2.24}$$

The solution is easily seen to be $x_1 = 1.000$ and $x_2 = 1.000$.

Now consider a small change in the right-hand side:

$$\begin{aligned} 1.000x_1 + 0.500x_2 &= 1.500 \\ 0.667x_1 + 0.333x_2 &= 0.999 \end{aligned} \tag{2.25}$$

This system has solution $x_1 = 0.000$ and $x_2 = 3.000$.

Alternatively, consider a small change in one of the elements of the coefficient matrix:

$$\begin{aligned} 1.000x_1 + 0.500x_2 &= 1.500 \\ 0.667x_1 + 0.334x_2 &= 1.000 \end{aligned} \tag{2.26}$$

The solution now is $x_1 = 2.000$ and $x_2 = -1.000$.

In both cases, small changes of the order of 10^{-3} in the input (the elements of the coefficient matrix or the right-hand side) result in relatively large changes (of the order of 1) in the output (the solution). Solving the system (either one of them) is an ill-conditioned problem.

The nature of the data that causes ill-conditioning depends on the type of problem. In this case, the problem is that the lines represented by the equations are almost parallel, as seen in Figure 2.1, and so their point of intersection is very sensitive to slight changes in the coefficients defining the lines.

For a specific problem such as solving a system of equations, we may quantify the condition of the matrix by a *condition number*. To develop this quantification for the problem of solving linear equations, consider a linear system $Ax = b$, with A nonsingular and $b \neq 0$, as above. Now perturb the system slightly by adding a small amount, δb, to b, and let $\tilde{b} = b + \delta b$. The system

$$A\tilde{x} = \tilde{b}$$

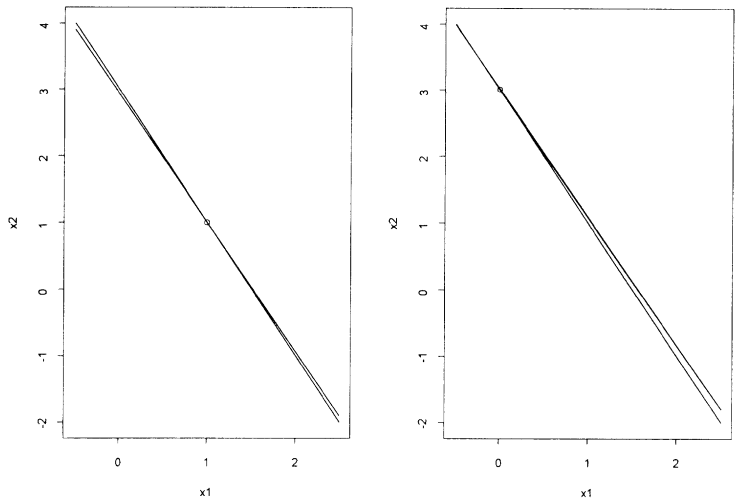


Figure 2.1: Almost Parallel Lines: Ill-Conditioned Coefficient Matrices, Equations (2.24) and (2.25)

has a solution $\tilde{x} = \delta x + x = A^{-1}\tilde{b}$. (Notice that δb and δx do not necessarily represent scalar multiples of the respective vectors.) If the system is well-conditioned, for any reasonable norm, if $\|\delta b\|/\|b\|$ is small, then $\|\delta x\|/\|x\|$ is likewise small.

From $\delta x = A^{-1}\delta b$ and the inequality in (2.19) (page 72), for the induced norm on A, we have

$$\|\delta x\| \leq \|A^{-1}\| \, \|\delta b\|. \tag{2.27}$$

Likewise, because $b = Ax$, we have

$$\frac{1}{\|x\|} \leq \|A\| \frac{1}{\|b\|}; \tag{2.28}$$

and (2.27) and (2.28) together imply

$$\frac{\|\delta x\|}{\|x\|} \leq \|A\| \, \|A^{-1}\| \frac{\|\delta b\|}{\|b\|}. \tag{2.29}$$

This provides a bound on the change in the solution $\|\delta x\|/\|x\|$ in terms of the perturbation $\|\delta b\|/\|b\|$.

The bound in (2.29) motivates us to define the *condition number with respect to inversion,*, $\kappa(A)$, by

$$\kappa(A) = \|A\| \, \|A^{-1}\|, \tag{2.30}$$

for nonsingular A. In the context of linear algebra the condition number with respect to inversion is so dominant in importance that we generally just refer to it as the "condition number". A condition number is a useful measure of the condition of A for the problem of solving a linear system of equations. There are other condition numbers useful in numerical analysis, however, such as the condition number for computing the sample variance in equation (1.8) on page 32, or the condition number for a root of a function.

We can write (2.29) as

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A)\frac{\|\delta b\|}{\|b\|}, \tag{2.31}$$

and, following a similar development as above, write

$$\frac{\|\delta b\|}{\|b\|} \leq \kappa(A)\frac{\|\delta x\|}{\|x\|}. \tag{2.32}$$

These inequalities, as well as the other ones we write in this section, are sharp, as we can see by letting $A = I$.

Because the condition number is an upper bound on a quantity that we would not want to be large, a large condition number is "bad".

Notice our definition of the condition number does not specify the norm; it only required that the norm be an induced norm. (An equivalent definition does not rely on the norm being an induced norm.) We sometimes specify a condition number with regard to a particular norm, and just as we sometimes denote a specific norm by a special symbol, we may use a special symbol to denote a specific condition number. For example, $\kappa_p(A)$ may denote the condition number of A in terms of an L_p norm. Most of the properties of condition numbers are independent of the norm used.

The coefficient matrix in equations (2.24) and (2.25) is

$$A = \begin{bmatrix} 1.000 & 0.500 \\ 0.667 & 0.333 \end{bmatrix},$$

and its inverse is

$$A^{-1} = \begin{bmatrix} -666 & 1000 \\ 1344 & -2000 \end{bmatrix}.$$

It is easy to see that

$$\|A\|_1 = 1.667,$$

and

$$\|A^{-1}\|_1 = 3000,$$

hence,

$$\kappa_1(A) = 5001.$$

Likewise,

$$\|A\|_\infty = 1.500,$$

and

$$\|A^{-1}\|_\infty = 3344,$$

hence,

$$\kappa_\infty(A) = 5016.$$

Notice that the condition numbers are not exactly the same, but they are close. Although we used this matrix in an example of ill-conditioning, these condition numbers, although large, are not so large as to cause undue concern for numerical computations. Indeed, the systems of equations in (2.24), (2.25), and (2.26) would not cause problems for a computer program to solve them. Notice also that the condition numbers are of the order of magnitude of the ratio of the output perturbation to the input perturbation in those equations.

An interesting relationship for the condition number is

$$\kappa(A) = \frac{\max_{x\neq 0} \frac{\|Ax\|}{\|x\|}}{\min_{x\neq 0} \frac{\|Ax\|}{\|x\|}} \tag{2.33}$$

(see Exercise 2.16, page 86).

The numerator and denominator in (2.33) look somewhat like the maximum and minimum eigenvalues, as we have suggested. Indeed, the L_2 condition number is just the ratio of the largest eigenvalue in absolute value to the smallest (see page 73).

The eigenvalues of the coefficient matrix in equations (2.24) and (2.25) are 1.333375 and -0.0003750, and so

$$\kappa_2(A) = 3555.67,$$

which is the same order of magnitude as $\kappa_\infty(A)$ and $\kappa_1(A)$ computed above.

Other facts about condition numbers are:

- $\kappa(A) = \kappa(A^{-1})$
- $\kappa(cA) = \kappa(A), \quad \text{for } c \neq 0$
- $\kappa(A) \geq 1$
- $\kappa_1(A) = \kappa_\infty(A^{\mathrm{T}})$
- $\kappa_2(A^{\mathrm{T}}) = \kappa_2(A)$
- $\kappa_2(A^{\mathrm{T}}A) = \kappa_2^2(A)$
  $\geq \kappa_2(A)$
- if A and B are orthogonally similar then, $\|A\|_2 = \|B\|_2$ (see equation (2.21))

Even though the condition number provides a very useful indication of the condition of the problem of solving a linear system of equations, it can be misleading at times. Consider, for example, the coefficient matrix

$$A = \begin{bmatrix} 1 & 0 \\ 0 & \epsilon \end{bmatrix}.$$

It is easy to see that

$$\kappa_1(A) = \kappa_2(A) = \kappa_\infty(A) = \frac{1}{\epsilon},$$

and so if ϵ is small, the condition number is large. It is easy to see, however, that small changes to the elements of A or of b in the system $Ax = b$ do not cause undue changes in the solution (our heuristic definition of ill-conditioning). In fact, the simple expedient of multiplying the second row of A by $1/\epsilon$ (that is, multiplying the second equation, $a_{21}x_1 + a_{22}x_2 = b_2$, by $1/\epsilon$) yields a linear system that is very well-conditioned.

This kind of apparent ill-conditioning is called *artificial ill-conditioning*. It is due to the different rows (or columns) of the matrix having a very different *scale*; the condition number can be changed just by scaling the rows or columns. This usually does not make a linear system any better or any worse conditioned.

In Section 3.4 we relate the condition number to bounds on the numerical accuracy of the solution of a linear system of equations.

The relationship between the size of the matrix and its condition number is interesting. In general, we would expect the condition number to increase as the size increases. This is the case, but the nature of the increase depends on the type of elements in the matrix. If the elements are randomly and independently distributed as normal or uniform with mean of zero and variance of one, the increase in the condition number is approximately linear in the size of the matrix (see Exercise 2.19, page 86).

Our definition of condition number given above is for nonsingular matrices. We can formulate a useful alternate definition that extends to singular matrices and to nonsquare matrices: the *condition number* of a matrix is the ratio of the largest singular value in absolute value to the smallest nonzero singular value in absolute value.

The condition number, like the determinant, is not easy to compute (see page 115 in Section 3.8).

2.1.18 Matrix Derivatives

The derivative of a vector or matrix with respect to a scalar variable is just the array with the same shape (vector or matrix) whose elements are the ordinary derivative with respect to the scalar.

The derivative of a scalar-valued function with respect to a vector is a vector of the partial derivatives of the function with respect to the elements of the

vector. If f is a function, and $x = (x_1, \ldots, x_n)$ is a vector,

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right).$$

This vector is called the *gradient*, and is sometimes denoted by g_f or by ∇f. The expression

$$\frac{\mathrm{d}f}{\mathrm{d}x}$$

may also be written as

$$\frac{\mathrm{d}}{\mathrm{d}x} f.$$

The gradient is used in finding the maximum or minimum of a function. Some methods of solving linear systems of equations formulate the problem as a minimization problem. We discuss one such method in Section 3.3.2.

For a vector-valued function f, the matrix whose rows are the transposes of the gradients is called the *Jacobian*. We denote the Jacobian of the function f by J_f. The transpose of the Jacobian, that is, the matrix whose columns are the gradients, is denoted by ∇f for the vector-valued function f. (Note that the ∇ symbol can denote either a vector or a matrix.) Thus, the Jacobian for the system above is

$$\begin{aligned} J_f &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ & & \cdots & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \\ &= (\nabla f)^{\mathrm{T}}. \end{aligned}$$

Derivatives of vector/matrix expressions with respect to a vector are similar to derivatives of similar expressions with respect to a scalar. For example, if A is a matrix and x is a conformable vector, we have:

$$\begin{aligned} \frac{\mathrm{d}x^{\mathrm{T}}A}{\mathrm{d}x} &= A \\ \frac{\mathrm{d}Ax}{\mathrm{d}x} &= A^{\mathrm{T}} \\ \frac{\mathrm{d}x^{\mathrm{T}}Ax}{\mathrm{d}x} &= Ax + A^{\mathrm{T}}x \end{aligned}$$

The normal equations that determine a least squares fit of a linear regression model are obtained by taking the derivative of $(y - Xb)^{\mathrm{T}}(y - Xb)$ and equating it to zero:

$$\frac{\mathrm{d}(y - Xb)^{\mathrm{T}}(y - Xb)}{\mathrm{d}b} = \frac{\mathrm{d}(y^{\mathrm{T}}y - 2b^{\mathrm{T}}X^{\mathrm{T}}y + b^{\mathrm{T}}X^{\mathrm{T}}Xb)}{\mathrm{d}b}$$

$$\begin{aligned} &= -2X^{\mathrm{T}}y + 2X^{\mathrm{T}}Xb \\ &= 0. \end{aligned}$$

We discuss these equations further in Section 6.2.

The derivative of a function with respect to a matrix is a matrix with the same shape consisting of the partial derivatives of the function with respect to the elements of the matrix. The derivative of a matrix Y with respect to the matrix X is thus

$$\frac{\mathrm{d}Y}{\mathrm{d}X} = Y \otimes \frac{\mathrm{d}}{\mathrm{d}X}.$$

Rogers (1980) and Magnus and Neudecker (1988) provide extensive discussions of matrix derivatives.

2.2 Computer Representations and Basic Operations

Most scientific computational problems involve vectors and matrices. It is necessary to work with either the elements of vectors and matrices individually or with the arrays themselves. Programming languages such as Fortran 77 and C provide the capabilities for working with the individual elements, but not directly with the arrays. Fortran 90 and higher-level languages such as Matlab allow direct manipulation with vectors and matrices.

We measure error in a scalar quantity either as *absolute error*, $|\tilde{r} - r|$, where r is the true value and $\tilde{r}$ is the computed or rounded value, or as *relative error*, $|\tilde{r} - r|/r$ (as long as $r \neq 0$). The errors in vectors or matrices are generally expressed in terms of norms. The relative error in the representation of the vector v, or as a result of computing v, may be expressed as $\|\tilde{v} - v\|/\|v\|$ (as long as $\|v\| \neq 0$), where $\tilde{v}$ is the computed vector. We often use the notation $\tilde{v} = v + \delta v$, and so $\|\delta v\|/\|v\|$ is the relative error. The vector norm used may depend on practical considerations about the errors in the individual elements.

2.2.1 Computer Representation of Vectors and Matrices

The elements of vectors and matrices are represented as ordinary numeric data as we described in Section 1.1, in either fixed-point or floating-point representation. The elements are generally stored in a logically contiguous area of the computer memory. What is logically contiguous may not be physically contiguous, however. There are no convenient mappings of computer memory that would allow matrices to be stored in a logical rectangular grid, so matrices are usually stored either as columns strung end-to-end (a "column-major" storage) or as rows strung end-to-end (a "row-major" storage). In using a computer language or a software package, sometimes it is necessary to know which way the matrix is stored. For some software to deal with matrices of varying sizes, the user must specify the length of one dimension of the array containing the

matrix. (In general, the user must specify the lengths of all dimensions of the array except one.) In Fortran subroutines it is common to have an argument specifying the leading dimension (number of rows), and in C functions it is common to have an argument specifying the column dimension. (See the examples in Figure 5.1 on page 145 and Figure 5.2 on page 146 for illustrations of the leading dimension argument.)

Sometimes in accessing a partition of a given matrix, the elements occur at fixed distances from each other. If the storage is row-major for an $n \times m$ matrix, for example, the elements of a given column occur at a fixed distance of m from each other. This distance is called the "stride", and it is often more efficient to access elements that occur with a fixed stride than it is to access elements randomly scattered. Just accessing data from computer memory contributes significantly to the time it takes to perform computations.

If a matrix has many elements that are zeros, and if the positions of those zeros are easily identified, many operations on the matrix can be speeded up. Matrices with many zero elements are called *sparse matrices*; they occur often in certain types of problems, for example in the solution of differential equations and in statistical designs of experiments. The first consideration is how to represent the matrix and to store the matrix and the location information. Different software systems may use different schemes to store sparse matrices. The method used in the IMSL Libraries, for example, is described on page 144. Another important consideration is how to preserve the sparsity during intermediate computations. Pissanetzky (1984) considers these and other issues in detail.

2.2.2 Multiplication of Vectors and Matrices

Arithmetic on vectors and matrices involves arithmetic on the individual elements. The arithmetic on the elements is performed as we have discussed in Section 1.2.

The way the storage of the individual elements is organized is very important for the efficiency of computations. Also, the way the computer memory is organized and the nature of the numerical processors affect the efficiency and may be an important consideration in the design of algorithms for working with vectors and matrices.

The best methods for performing operations on vectors and matrices in the computer may not be the methods that are suggested by the definitions of the operations.

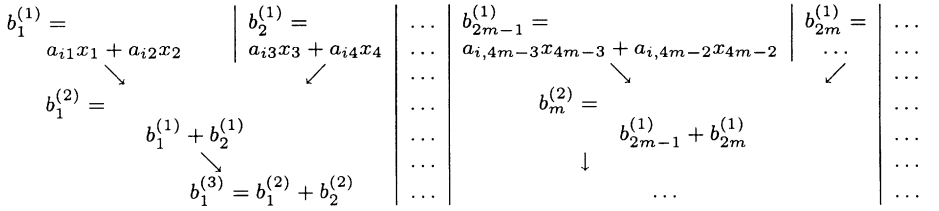
In most numerical computations with vectors and matrices there is more than one way of performing the operations on the scalar elements. Consider the problem of evaluating the matrix times vector product, $b = Ax$, where A is $n \times m$. There are two obvious ways of doing this:

- compute each of the n elements of b, one at a time, as an inner product of m-vectors, $b_i = a_i^{\mathrm{T}} x = \sum_j a_{ij} x_j$, or

- update the computation of all of the elements of b simultaneously as

1. For $i = 1, \ldots, n$, let $b_i^{(0)} = 0$.
2. For $j = 1, \ldots, m$,
   {
   for $i = 1, \ldots, n$,
   {
   let $b_i^{(i)} = b_i^{(i-1)} + a_{ij}x_j$.
   }
   }

If there are p processors available for parallel processing, we could use a fan-in algorithm (see page 22) to evaluate Ax as a set of inner products:



The order of the computations is nm (or n^2).

Multiplying two matrices can be considered as a problem of multiplying several vectors by a matrix, as described above. The order of computations is $O(n^3)$. Another way that can be faster for large matrices is the so-called *Strassen algorithm*. Suppose A and B are square matrices with equal and even dimensions. Partition them into submatrices of equal size, and consider the block representation of the product:

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

where all blocks are of equal size. Form

$$\begin{aligned} P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ P_2 &= (A_{21} + A_{22})B_{11} \\ P_3 &= A_{11}(B_{12} - B_{22}) \\ P_4 &= A_{22}(B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{12})B_{22} \\ P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

Then we have (see the discussion on partitioned matrices in Section 2.1):

$$\begin{aligned} C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{12} &= P_3 + P_5 \\ C_{21} &= P_2 + P_4 \\ C_{22} &= P_1 + P_3 - P_2 + P_6. \end{aligned}$$

Notice that the total number of multiplications of matrices is seven, instead of eight as it would be in forming

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

directly. Whether the blocks are matrices or scalars, the same analysis holds. Of course, in either case there are more additions. Addition of two $k \times k$ matrices is $O(k^2)$, so for a large enough value of n the total number of operations using the Strassen algorithm is less than the number required for performing the multiplication in the usual way.

This idea can also be used recursively. (If the dimension, n, contains a factor 2^e, the algorithm can be used directly e times and then use conventional matrix multiplication on any submatrix of dimension $\leq n/2^e$.)

If the dimension of the matrices is not even, or if the matrices are not square, it is a simple matter to pad the matrices with zeros, and use this same idea.

The order of computations of the Strassen algorithm is $O(n^{2.81})$, instead of $O(n^3)$ as in the ordinary method. The algorithm can be implemented in parallel (see Bailey, Lee, and Simon, 1990).

Exercises

2.1. Give an example of two vector spaces whose union is not a vector space.

2.2. Let $\{v_i \ ; \ i = 1, 2, \ldots, n\}$ be an orthonormal basis for the n-dimensional vector space V. Let $x \in V$ have the representation

$$x = \sum c_i v_i.$$

Show that the coefficients c_i can be computed as

$$c_i = \langle x, v_i \rangle.$$

2.3. Prove the Cauchy-Schwarz inequality for the dot product of matrices, (2.3), page 58.

2.4. Show that for any quadratic form, $x^{\mathrm{T}}Ax$, there is a symmetric matrix A_s, such that $x^{\mathrm{T}}A_sx = x^{\mathrm{T}}Ax$. (The proof is by construction, with $A_s = \frac{1}{2}(A+A^{\mathrm{T}})$, first showing A_s is symmetric, and then that $x^{\mathrm{T}}A_sx = x^{\mathrm{T}}Ax$.)

2.5. By writing $AA^{-1} = I$, derive the expression for the inverse of a partitioned matrix given in equation (2.6).

2.6. Show that the expression given for the generalized inverse in equation (2.8) on page 64 is correct.

2.7. Prove that the eigenvalues of a symmetric matrix are real. *Hint:* $A^{\mathrm{T}}A = A^2$.

2.8. Let A be a matrix with an eigenvalue λ and corresponding eigenvector v. Consider the matrix polynomial in A,

$$f(A) = c_pA^p + \cdots + c_1A + c_0I.$$

Show that $f(\lambda)$, that is,

$$c_p\lambda^p + \cdots + c_1\lambda + c_0,$$

is an eigenvalue of $f(A)$ with corresponding eigenvector v.

2.9. Prove that the induced norm (page 72) is a matrix norm; that is, prove that it satisfies the consistency property.

2.10. Prove that, for the square matrix A,

$$\|A\|_2^2 = \rho(A^{\mathrm{T}}A)$$

Hint: Let $v_1, v_2, \ldots, v_n$ be orthonormal eigenvectors and $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ be the eigenvalues of $A^{\mathrm{T}}A$; represent an arbitrary normalized vector x as $\sum c_iv_i$; show that $\|A\|_2^2 = \max x^{\mathrm{T}}A^{\mathrm{T}}Ax = \sum \lambda_ic_i$, and that this latter quantity is always less than or equal to λ_n, but indeed is equal to λ_n when $x = v_n$.

2.11. The triangle inequality for matrix norms: $\|A + B\| \leq \|A\| + \|B\|$.

(a) Prove the triangle inequality for the matrix L_1 norm.

(b) Prove the triangle inequality for the matrix L_∞ norm.

(c) Prove the triangle inequality for the matrix Frobenius norm. (See the proof of inequality 2.16, on page 71.)

2.12. Prove that the Frobenius norm satisfies the consistency property.

2.13. Prove for any square matrix A with real elements,

$$\|A\|_2 \leq \|A\|_F.$$

Hint: Use the Cauchy-Schwarz inequality.

2.14. Prove the inequality (2.19) on page 72:

$$\|Ax\| \leq \|A\| \, \|x\|.$$

Hint: Obtain the inequality from the definition of the induced matrix norm.

2.15. Let Q be an $n \times n$ orthogonal matrix and let x be an n-vector.

(a) Prove equation (2.21):

$$\|Qx\|_2 = \|x\|_2.$$

Hint: Write $\|Qx\|_2$ as $\sqrt{(Qx)^{\mathrm{T}} Qx}$.

(b) Give examples to show that this does not hold for other norms.

2.16. Let A be nonsingular, and let $\kappa(A) = \|A\| \, \|A^{-1}\|$.

(a) Prove equation (2.33):

$$\kappa(A) = \frac{\max_{x \neq 0} \frac{\|Ax\|}{\|x\|}}{\min_{x \neq 0} \frac{\|Ax\|}{\|x\|}}.$$

(b) Using the relationship above, explain heuristically why $\kappa(A)$ is called the "condition number" of A.

2.17. Consider the four properties of a dot product beginning on page 50. For each one, state whether the property holds in computer arithmetic. Give examples to support your answers.

2.18. Assuming the model (1.1) on page 6 for the floating-point number system, give an example of a nonsingular 2×2 matrix that is algorithmically singular.

2.19. A Monte Carlo study of condition number and size of the matrix.
For $n = 5, 10, \ldots, 30$, generate 100 $n \times n$ matrices whose elements have independent $N(0, 1)$ distributions. For each, compute the L_2 condition number and plot the mean condition number versus the size of the matrix. At each point, plot error bars representing the sample "standard error" (the standard deviation of the sample mean at that point). How would you describe the relationship between the condition number and the size?

Chapter 3

Solution of Linear Systems

One of the most common problems in numerical computing is to solve the linear system

$$Ax = b,$$

that is, for given A and b, to find x such that the equation holds. The system is said to be *consistent* if there exists such an x, and in that case a solution x may be written as A^-b, where A^- is some inverse of A. If A is square and of full rank, we can write the solution as $A^{-1}b$.

It is important to distinguish the expression $A^{-1}b$ or A^+b, which represents the solution, from the method of computing the solution. We would never compute A^{-1} just so we could multiply it by b to form the solution $A^{-1}b$.

There are two general methods of solving a system of linear equations: direct methods and iterative methods. A direct method uses a fixed number of computations that would in exact arithmetic lead to the solution; an iterative method generates a sequence of approximations to the solution. Iterative methods often work well for very large sparse matrices.

3.1 Gaussian Elimination

The most common direct method for the solution of linear systems is Gaussian elimination. The basic idea in this method is to form equivalent sets of equations, beginning with the system to be solved, $Ax = b$, or

$$\begin{aligned} a_1^{\mathrm{T}}x &= b_1 \\ a_2^{\mathrm{T}}x &= b_2 \\ \cdots &= \cdots \\ a_n^{\mathrm{T}}x &= b_n, \end{aligned}$$

where a_j^{T} is the j^{th} row of A. An equivalent set of equations can be formed by a sequence of *elementary operations* on the equations in the given set. There are two kinds of elementary operations: an interchange of two equations,

$$\begin{aligned} a_j^{\mathrm{T}}x = b_j \quad &\leftarrow \quad a_k^{\mathrm{T}}x = b_k \\ a_k^{\mathrm{T}}x = b_k \quad &\leftarrow \quad a_j^{\mathrm{T}}x = b_j, \end{aligned}$$

which affects two equations simultaneously, or the replacement of a single equation with a linear combination of it and another equation:

$$a_j^{\mathrm{T}}x = b_j \quad \leftarrow \quad c_j a_j^{\mathrm{T}}x + c_k a_k^{\mathrm{T}}x = c_j b_j + c_k b_k,$$

where $c_j \neq 0$. If $c_k = 0$ in this operation, it is the simple elementary operation of scalar multiplication of a single equation.

The interchange operation can be accomplished by premultiplication by an elementary permutation matrix (see page 65):

$$E_{jk}Ax = E_{jk}b.$$

Likewise, the linear combination elementary operation can be effected by premultiplication by a matrix formed from the identity matrix by replacing its j^{th} row by a row with all zeros except for c_j in the j^{th} column and c_k in the k^{th} column. Such a matrix is denoted by $E_{jk}(c_j, c_k)$, for example,

$$E_{23}(c_2, c_3) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_2 & 0 & 0 \\ 0 & 0 & c_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Both E_{jk} and $E_{jk}(c_j, c_k)$ are called *elementary operator matrices.*

The elementary operation on the equation

$$a_2^{\mathrm{T}}x = b_2$$

in which the first equation is combined with it using $c_1 = -a_{21}/a_{11}$ and $c_2 = 1$ will yield an equation with a zero coefficient for x_1. Generalizing this, we perform elementary operations on the second through the n^{th} equations to yield a set of equivalent equations in which all but the first have zero coefficients for x_1.

Next, we perform elementary operations using the second equation with the third through the n^{th} equations, so that the new third through the n^{th} equations have zero coefficients for x_2.

The sequence of equivalent equations is

$$(1) \qquad \begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + \cdots + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + \cdots + & a_{2n}x_n & = & b_2 \\ \vdots & + & \vdots & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + \cdots + & a_{nn}x_n & = & b_n \end{array}$$

$$
(2)\qquad
\begin{array}{rcrcrcl}
a_{11}x_1 & + & a_{12}x_2 & +\cdots+ & a_{1n}x_n & = & b_1 \\
 & & a_{22}^{(1)}x_2 & +\cdots+ & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\
 & & \vdots & +\cdots+ & \vdots & & \vdots \\
 & & a_{n2}^{(1)}x_2 & +\cdots+ & a_{nn}^{(1)}x_n & = & b_n^{(1)}
\end{array}
$$

$$\vdots$$

$$
(n)\qquad
\begin{array}{rcrcrcrcl}
a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\
 & & a_{22}^{(1)}x_2 & + & \cdots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\
 & & & & & & \vdots & & \vdots \\
 & & & & a_{n-1,n-1}^{(n-2)}x_{n-1} & + & a_{n-1,n}^{(n-2)}x_n & = & b_{n-1}^{(n-2)} \\
 & & & & & & a_{nn}^{(n-1)}x_n & = & b_n^{(n-1)}
\end{array}
$$

This last system is easy to solve. It is upper triangular. The last equation in the system yields

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}.$$

By back substitution we get

$$x_{n-1} = \frac{(b_{n-1}^{(n-2)} - a_{n-1,n}^{(n-2)}x_n)}{a_{n-1,n-1}^{(n-2)}},$$

and the rest of the x's in a similar manner.

Thus, Gaussian elimination consists of two steps, the forward reduction, which is order $O(n^3)$, and the back substitution, which is order $O(n^2)$.

The only obvious problem with this method arises if some of the $a_{kk}^{(k-1)}$'s used as divisors are zero (or very small in magnitude). These divisors are called "pivot elements".

Suppose, for example, we have the equations

$$
\begin{array}{rcrcl}
0.0001x_1 & + & x_2 & = & 1 \\
x_1 & + & x_2 & = & 2
\end{array}
$$

The solution is $x_1 = 1.0001$ and $x_2 = 0.9999$. Suppose we are working with 3 digits of precision (so our solution is $x_1 = 1.00$ and $x_2 = 1.00$). After the first step in Gaussian elimination we have

$$
\begin{array}{rcrcr}
0.0001x_1 & + & x_2 & = & 1 \\
 & & -10,000x_2 & = & -10,000
\end{array}
$$

and so the solution by back substitution is $x_2 = 1.00$ and $x_1 = 0.000$. The L_2 condition number of the coefficient matrix is 2.618, so even though the coefficients do vary greatly in magnitude, we certainly would not expect any difficulty in solving these equations.

A simple solution to this potential problem is to interchange the equation having the small leading coefficient with an equation below it. Thus, in our example, we first form

$$\begin{aligned} x_1 &+ x_2 = 2 \\ 0.0001x_1 &+ x_2 = 1 \end{aligned}$$

so that after the first step we have

$$\begin{aligned} x_1 &+ x_2 = 2 \\ & x_2 = 1 \end{aligned}$$

and the solution is $x_2 = 1.00$ and $x_1 = 1.00$.

Another strategy would be to interchange the column having the small leading coefficient with a column to its right. Both the row interchange and the column interchange strategies could be used simultaneously, of course. These processes, which obviously do not change the solution, are called *pivoting.* The equation or column to move into the active position may be chosen in such a way that the magnitude of the new diagonal element is the largest possible.

Performing only row interchanges, so that at the k^{th} stage the equation with

$$\max_{i=k}^{n} |a_{ik}^{(k-1)}|$$

is moved into the k^{th} row, is called *partial pivoting.* Performing both row interchanges and column interchanges, so that

$$\max_{i=k;j=k}^{n;n} |a_{ij}^{(k-1)}|$$

is moved into the k^{th} diagonal position, is called *complete pivoting.* See Exercises 3.3a and 3.3b.

It is always important to distinguish descriptions of effects of actions from the actions that are actually carried out in the computer. Pivoting is "interchanging" rows or columns. We would usually do something like that in the computer only when we are finished and want to produce some output. In the computer, a row or a column is determined by the index identifying the row or column. All we do for pivoting is to keep track of the indices that we have permuted.

There are many more computations required in order to perform complete pivoting than are required to perform partial pivoting. Gaussian elimination with complete pivoting can be shown to be stable (i.e., the algorithm yields an exact solution to a slightly perturbed system, $(A + \delta A)x = b$). For Gaussian elimination with partial pivoting there exist examples to show that it is not

stable. These examples are somewhat contrived, however, and experience over many years has indicated that Gaussian elimination with partial pivoting is stable for most problems occurring in practice. For this reason together with the computational savings, Gaussian elimination with partial pivoting is one of the most commonly used methods for solving linear systems. See Golub and Van Loan (1996) for a further discussion of these issues.

There are two modifications of partial pivoting that result in stable algorithms. One is to add one step of iterative refinement (see Section 3.5, page 109) following each pivot. It can be shown that Gaussian elimination with partial pivoting together with one step of iterative refinement is unconditionally stable (Skeel, 1980). Another modification is to consider two columns for possible interchange in addition to the rows to be interchanged. This does not require nearly as many computations as complete pivoting does. Higham (1997) shows that this method, suggested by Bunch and Kaufman (1977) and used in LINPACK and LAPACK, is stable.

Each step in Gaussian elimination is equivalent to multiplication of the current coefficient matrix, $A^{(k)}$, by some matrix L_k. If we ignore pivoting (i.e., assume it is handled by permutation vectors), the L_k matrix has a particularly simple form:

$$L_k = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ & \ddots & & & & \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\frac{a^{(k)}_{k+1,k}}{a^{(k)}_{kk}} & 1 & \cdots & 0 \\ & & & & \ddots & \\ 0 & \cdots & -\frac{a^{(k)}_{nk}}{a^{(k)}_{kk}} & 0 & \cdots & 1 \end{bmatrix}.$$

Each L_k is nonsingular, with a determinant of 1. The whole process of forward reduction can be expressed as a matrix product,

$$U = L_{n-1}L_{n-2}\ldots L_2L_1A,$$

and by the way we have performed the forward reduction, U is an upper triangular matrix. The matrix $L_{n-1}L_{n-2}\ldots L_2L_1$ is nonsingular and is unit lower triangular (all 1's on the diagonal). Its inverse is also, therefore, unit lower triangular. Call its inverse L. The forward reduction is equivalent to expressing A as LU,

$$A = LU; \tag{3.1}$$

hence this process is called an *LU factorization* or an *LU decomposition.* (We use the terms "matrix factorization" and "matrix decomposition" interchangeably.)

Notice, of course, that we do not necessarily store the two matrix factors in the computer.

3.2 Matrix Factorizations

Direct methods of solution of linear systems all use some form of matrix factorization, similar to the LU factorization in the last section. Matrix factorizations are also performed for reasons other than to solve a linear system. The important matrix factorizations are:

- LU factorization and LDU factorization (primarily, but not necessarily, for square matrices)
- Cholesky factorization (for nonnegative definite matrices)
- QR factorization
- Singular value factorization

In this section we discuss each of these factorizations.

3.2.1 LU and LDU Factorizations

The LU factorization is the most commonly used method to solve a linear system. For any matrix (whether square or not) that is expressed as LU, where L is unit lower triangular and U is upper triangular, the product LU is called the LU factorization. If an LU factorization exists, it is clear that the upper triangular matrix, U, can be made unit upper triangular (all 1's on the diagonal), by putting the diagonal elements of the original U into a diagonal matrix D, and then writing the factorization as LDU, where U is now a unit upper triangular matrix.

The computations leading up to equation (3.1) provide a method of computing an LU factorization. This method, based on Gaussian elimination over rows, consists of a sequence of outer products. Another way of arriving at the LU factorization is by use of the inner product. From equation (3.1), we see

$$a_{ij} = \sum_{k=1}^{i-1} l_{ik}u_{kj} + u_{ij},$$

so

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}}{u_{jj}}, \quad \text{for } i = j+1, j+2, \ldots, n. \tag{3.2}$$

The use of computations implied by equation (3.2) is called the Doolittle method or the Crout method. (There is a slight difference in the Doolittle method and the Crout method: the Crout method yields a decomposition in which the 1's are on the diagonal of the U matrix, rather than the L matrix.)

Whichever method is used to compute the LU decomposition, $n^3/3$ multiplications and additions are required.

It is neither necessary nor sufficient that a matrix be nonsingular for it to have an LU factorization. An example of a singular matrix that has an LU

factorization is any upper triangular matrix with all zeros on the diagonal. In this case, U can be chosen as the matrix itself, and L chosen as the identity:

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

An example of a nonsingular matrix that does not have an LU factorization is an identity matrix with permuted rows or columns:

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

If a nonsingular matrix has an LU factorization, L and U are unique.

A sufficient condition for an $n \times m$ matrix A to have an LU factorization is that for $k = 1, 2, \ldots, \min(n-1, m)$, each $k \times k$ principal submatrix of A, A_k, be nonsingular. Note this fact also provides a way of constructing a singular matrix that has an LU factorization. Furthermore, for $k = 1, 2, \ldots, \min(n, m)$,

$$\det(A_k) = u_{11} u_{22} \cdots u_{kk}.$$

3.2.2 Cholesky Factorization

If the coefficient matrix A is symmetric and *positive definite*, that is, if $x^{\mathrm{T}} A x > 0$, for all $x \neq 0$, another important factorization is the *Cholesky decomposition.* In this factorization,

$$A = T^{\mathrm{T}} T, \tag{3.3}$$

where T is an upper triangular matrix with positive diagonal elements.

The factor T in the Cholesky decomposition is sometimes called the *square root* for obvious reasons. A factor of this form is unique up to the sign, just as a square root is. To make the Cholesky factor unique, we require that the diagonal elements be positive. The elements along the diagonal of T will be square roots. Notice, for example, t_{11} is $\sqrt{a_{11}}$. The Cholesky decomposition can also be formed as $\tilde{T}^{\mathrm{T}} D \tilde{T}$, where D is a diagonal matrix that allows the diagonal elements of $\tilde{T}$ to be computed without taking square roots. This modification is sometimes called a *Banachiewicz factorization* or *root-free Cholesky.* The Banachiewicz factorization can be computed in essentially the same way as the Cholesky factorization shown in Algorithm 3.1: just put 1's along the diagonal of T, and store the unsquared quantities in a vector d.

In Exercise 3.2 you are asked to prove that there exists a unique T. The algorithm for computing the Cholesky factorization serves as a constructive proof of the existence and uniqueness. (The uniqueness is seen by factoring the principal square submatrices.)

Algorithm 3.1 Cholesky Factorization

1. Let $t_{11} = \sqrt{a_{11}}$.
2. For $j = 2, \ldots, n$, let $t_{1j} = a_{1j}/t_{11}$.
3. For $i = 2, \ldots, n$,
   {
   let $t_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} t_{ki}^2}$, and
   for $j = i + 1, \ldots, n$,
   {
   let $t_{ij} = (a_{ij} - \sum_{k=1}^{i-1} t_{ki}t_{kj})/t_{ii}$.
   }
   }

∎

There are other algorithms for computing the Cholesky decomposition. The method given in Algorithm 3.1 is sometimes called the inner-product formulation because the sums in step 3 are inner products. The algorithm for computing the Cholesky decomposition is numerically stable. Although the order of the number of computations is the same, there are only about half as many computations in the Cholesky factorization as in the LU factorization. Another advantage of the Cholesky factorization is that there are only $n(n+1)/2$ unique elements, as opposed to $n^2 + n$ in the LU decomposition. An important difference, however, is that Cholesky applies only to symmetric, positive definite matrices.

For a symmetric matrix, the LDU factorization is $U^{\mathrm{T}}DU$; hence we have for the Cholesky factor,

$$T = D^{\frac{1}{2}}U,$$

where $D^{\frac{1}{2}}$ is the matrix whose elements are the square roots of the corresponding elements of D.

Any symmetric nonnegative definite matrix has a decomposition similar to the Cholesky for a positive definite matrix. If A is $n \times n$ with rank r, there exists a unique matrix T, such that $A = T^{\mathrm{T}}T$, where T is an upper triangular matrix with r positive diagonal elements and $n - r$ rows containing all zeros. The algorithm is the same as Algorithm 3.1, except in step 3 if $t_{ii} = 0$, the entire row is set to zero. The algorithm serves as a constructive proof of the existence and uniqueness.

The LU and Cholesky decompositions generally are applied to square matrices. However, many of the linear systems that occur in scientific applications are *overdetermined*; that is, there are more equations than there are variables, resulting in a nonsquare coefficient matrix.

An overdetermined system may be written as

$$Ax \approx b,$$

where A is $n \times m$ $(n \geq m)$, or it may be written as

$$Ax = b + e,$$

where e is an n-vector of possibly arbitrary "errors". Because all equations cannot be satisfied simultaneously, we must define a meaningful "solution". A useful solution is an x such that e has a small norm. The most common definition is an x such that e has the least Euclidean norm, that is, such that the sum of squares of the e_i's is minimized.

It is easy to show that such an x satisfies the square system $A^{\mathrm{T}}Ax = A^{\mathrm{T}}b$. This expression is important and allows us to analyze the overdetermined system (not just to solve for the x, but to gain some better understanding of the system). It is easy to show that if A is full rank (i.e., of rank m, or all of its columns are linearly independent, or, redundantly, "full column rank"), then $A^{\mathrm{T}}A$ is positive definite. Therefore, we could apply either Gaussian elimination or the Cholesky decomposition to obtain the solution.

As we have emphasized many times before, however, useful conceptual expressions are not necessarily useful as computational formulations. That is sometimes true in this case also. Among other indications that it may be better to work directly on A is the fact that the condition number of $A^{\mathrm{T}}A$ is the square of the condition number of A. We discuss solution of overdetermined systems in Section 3.7.

3.2.3 QR Factorization

A very useful factorization is

$$A = QR, \tag{3.4}$$

where Q is orthogonal and R is upper triangular. This is called the QR factorization.

If A is nonsquare in (3.4), then R is such that its leading square matrix is upper triangular; for example if A is $n \times m$, and $n \geq m$, then

$$R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, \tag{3.5}$$

where R_1 is upper triangular.

For the $n \times m$ matrix A, with $n \geq m$, we can write

$$\begin{aligned} A^{\mathrm{T}}A &= R^{\mathrm{T}}Q^{\mathrm{T}}QR \\ &= R^{\mathrm{T}}R, \end{aligned}$$

so we see that the matrix R in the QR factorization is (or at least can be) the same as the matrix T in the Cholesky factorization of $A^{\mathrm{T}}A$.

There is some ambiguity in the Q and R matrices, but if the diagonal entries of R are required to be nonnegative, the ambiguity disappears, and the matrices in the QR decomposition are unique.

It is interesting to note that the Moore-Penrose inverse of A is immediately available from the QR factorization:

$$A^{+} = \begin{bmatrix} R_1^{-1} & 0 \end{bmatrix} Q^{\mathrm{T}}. \tag{3.6}$$

If A is not of full rank, we apply permutations to the columns of A by multiplying on the right by a permutation matrix. The permutations can be taken out by a second multiplication on the right. If A is of rank r ($\leq m$), the resulting decomposition consists of three matrices, an orthogonal Q, a T with an $r \times r$ upper triangular submatrix, and a permutation matrix P^{T}:

$$A = QTP^{\mathrm{T}}. \tag{3.7}$$

The matrix T has the form

$$T = \begin{bmatrix} T_1 & T_2 \\ 0 & 0 \end{bmatrix}, \tag{3.8}$$

where T_1 is upper triangular and is $r \times r$. The decomposition in (3.7) is not unique because of the permutation matrix. Choice of the permutation matrix is the same as the pivoting that we discussed in connection with Gaussian elimination. A generalized inverse of A is immediately available from (3.7):

$$A^- = P \begin{bmatrix} T_1^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^{\mathrm{T}}. \tag{3.9}$$

Additional orthogonal transformations can be applied from the right side of A in the form (3.7) to yield

$$A = QRU^{\mathrm{T}}, \tag{3.10}$$

where R has the form

$$R = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix}, \tag{3.11}$$

where R_1 is $r \times r$ upper triangular, Q is as in (3.7), and U^{T} is orthogonal. (The permutation matrix in (3.7) is also orthogonal, of course.) The decomposition (3.10) is unique. This decomposition provides the Moore-Penrose generalized inverse of A:

$$A^+ = U \begin{bmatrix} R_1^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^{\mathrm{T}}. \tag{3.12}$$

It is often of interest to know the rank of a matrix. Given a decomposition of the form (3.7), the rank is obvious, and in practice, this QR decomposition with pivoting is a good way to determine the rank of a matrix. The QR decomposition is said to be "rank revealing". The computations are quite sensitive to rounding, however, and the pivoting must be done with some care (see Section 2.7.3 of Björck, 1996, and see Hong and Pan, 1992).

There are three good methods for obtaining the QR factorization: Householder transformations, or reflections, Givens transformations, or rotations, and the (modified) Gram-Schmidt procedure. Different situations may make one or the other of these procedures better than the other two. For example, if the data are available only one row at a time, the Givens transformations are very convenient.

Whichever method is used to compute the QR decomposition, at least $2n^3/3$ multiplications and additions are required (and this is possible only when clever formulations are used). The operation count is therefore about twice as great as that for an LU decomposition.

The QR factorization is particularly useful in computations for overdetermined systems, as we see in Section 3.7, page 111, and in other computations involving nonsquare matrices.

3.2.4 Householder Transformations (Reflections)

Let u and v be orthonormal vectors, and let x be a vector in the space spanned by u and v, so

$$x = c_1 u + c_2 v$$

for some scalars c_1 and c_2. The vector

$$\tilde{x} = -c_1 u + c_2 v$$

is a *reflection* of x through the line defined by the vector u.

Now consider the matrix

$$Q = I - 2uu^{\mathrm{T}},$$

and note that

$$\begin{aligned} Qx &= c_1 u + c_2 v - 2c_1 uuu^{\mathrm{T}} - 2c_2 vuu^{\mathrm{T}} \\ &= c_1 u + c_2 v - 2c_1 u^{\mathrm{T}} uu - 2c_2 u^{\mathrm{T}} vu \\ &= -c_1 u + c_2 v \\ &= \tilde{x}. \end{aligned}$$

The matrix Q is a reflector. A reflection is also called a Householder reflection or a Householder transformation, and the matrix Q is called a Householder matrix. The following properties of Q are obvious.

- $Qu = -u$
- $Qv = v$ for any v orthogonal to u
- $Q = Q^{\mathrm{T}}$ (symmetric)
- $Q^{\mathrm{T}} = Q^{-1}$ (orthogonal)

The matrix uu^{T} is symmetric, idempotent, and of rank 1. (A transformation by a matrix of the form $A - uv^{\mathrm{T}}$ is often called a "rank-one" update, because uv^{T} is of rank 1. Thus, a Householder reflection is a special rank-one update.)

The usefulness of Householder reflections results from the fact that it is easy to construct a reflection that will transform a vector

$$x = (x_1, x_2, \ldots, x_n)$$

into a vector

$$\tilde{x} = (\tilde{x}_1, 0, \ldots, 0).$$

Now, if $Qx = \tilde{x}$, then $\|x\|_2 = \|\tilde{x}\|_2$ (see equation 2.21), so $\tilde{x}_1 = \pm\|x\|_2$. To construct the reflector, form the normalized vector $(x - \tilde{x})$, that is, let

$$v = (x_1 + \text{sign}(x_1)\|x\|,\ x_2,\ \ldots,\ x_n),$$

and $u = v/\|v\|$, where all norms are the L_2 norm. Notice that we could have chosen $\|x\|$ or $-\|x\|$ for the first element in $\tilde{x}$. *We choose the sign so as not to add quantities of different signs and possibly similar magnitudes.* (See the discussions of catastrophic cancellation in Chapter 1.)

Consider, for example, the vector

$$x = (3, 1, 2, 1, 1).$$

We have

$$\|x\| = 4,$$

so we form the vector

$$u = \frac{1}{\sqrt{56}}(7, 1, 2, 1, 1),$$

and the reflector,

$$\begin{aligned} Q &= I - 2uu^{\mathrm{T}} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} - \frac{1}{28}\begin{bmatrix} 49 & 7 & 14 & 7 & 7 \\ 7 & 1 & 2 & 1 & 1 \\ 14 & 2 & 4 & 2 & 2 \\ 7 & 1 & 2 & 1 & 1 \\ 7 & 1 & 2 & 1 & 1 \end{bmatrix} \\ &= \frac{1}{28}\begin{bmatrix} -21 & -7 & -14 & -7 & -7 \\ -7 & 27 & -2 & -1 & -1 \\ -14 & -2 & 24 & -2 & -2 \\ -7 & -1 & -2 & 27 & -1 \\ -7 & -1 & -2 & -1 & 27 \end{bmatrix}, \end{aligned}$$

to yield $Qx = (-4, 0, 0, 0, 0)$.

To use reflectors to compute a QR factorization, we form in sequence the reflector for the i^{th} column that will produce 0's below the (i, i) element. For a convenient example, consider the matrix

$$A = \begin{bmatrix} 3 & -\frac{98}{28} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 1 & \frac{122}{28} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 2 & -\frac{8}{28} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 1 & \frac{66}{28} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 1 & \frac{10}{28} & \mathtt{X} & \mathtt{X} & \mathtt{X} \end{bmatrix}.$$

The first transformation applied would be P_1, given as Q above, yielding

$$P_1 A = \begin{bmatrix} -4 & 1 & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & 5 & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & 1 & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & 3 & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & 1 & \mathtt{X} & \mathtt{X} & \mathtt{X} \end{bmatrix}.$$

We now choose a reflector to transform $(5, 1, 3, 1)$ to $(-6, 0, 0, 0)$. Forming the vector $(11, 1, 3, 1)/\sqrt{132}$, and proceeding as before, we get the reflector

$$\begin{aligned} Q_2 &= I - \frac{1}{66}(11, 1, 3, 1)(11, 1, 3, 1)^{\mathrm{T}} \\ &= \frac{1}{66} \begin{bmatrix} -55 & -11 & -33 & -11 \\ -11 & 65 & -3 & -1 \\ -33 & -3 & 57 & -3 \\ -11 & -1 & -3 & 65 \end{bmatrix}. \end{aligned}$$

We do not want to disturb the first column in $P_1 A$ shown above, so we form P_2 as

$$P_2 = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & Q_2 & \\ 0 & & & \end{bmatrix}.$$

Now we have

$$P_2 P_1 A = \begin{bmatrix} -4 & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & -6 & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & 0 & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & 0 & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ 0 & 0 & \mathtt{X} & \mathtt{X} & \mathtt{X} \end{bmatrix}.$$

Continuing in this way for three more steps we would have the QR decomposition of A, with $Q = P_5 P_4 P_3 P_2 P_1$.

The number of computations for the QR factorization of an $n \times n$ matrix using Householder reflectors is $2n^3/3$ multiplications and $2n^3/3$ additions. Carrig and Meyer (1997) describe two variants of the Householder transformations that take advantage of computer architectures that have a cache memory or that have a bank of floating-point registers whose contents are immediately available to the computational unit.

3.2.5 Givens Transformations (Rotations)

Another way of forming the QR decomposition is by use of orthogonal transformations that rotate a vector in such a way that a specified element becomes 0 and only one other element in the vector is changed. Such a method may be particularly useful if only part of the matrix to be transformed is available.

These transformations are called *Givens transformations*, or *Givens rotations*, or sometimes *Jacobi transformations*.

The basic idea of the rotation can be seen in the case of a vector of length 2. Given the vector $x = (x_1, x_2)$, we wish to rotate it to $\tilde{x} = (\tilde{x}_1, 0)$. As with a reflector, $\tilde{x}_1 = \|x\|$. Geometrically, we have the picture shown in Figure 3.1.

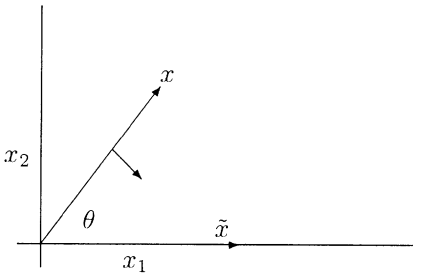


Figure 3.1: Rotation of x

It is easy to see that the orthogonal matrix

$$Q = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{3.13}$$

will perform this rotation of x, if $\cos\theta = x_1/\|x\|$ and $\sin\theta = x_2/\|x\|$. So we have

$$\begin{aligned} \tilde{x}_1 &= \frac{x_1^2}{\|x\|} + \frac{x_2^2}{\|x\|} \\ &= \|x\| \end{aligned}$$

and

$$\begin{aligned} \tilde{x}_2 &= -\frac{x_2 x_1}{\|x\|} + \frac{x_1 x_2}{\|x\|} \\ &= 0. \end{aligned}$$

As with the Householder reflection that transforms a vector

$$x = (x_1, x_2, x_3, \ldots, x_n)$$

into a vector

$$\tilde{x}_H = (\tilde{x}_{H1}, 0, 0, \ldots, 0),$$

it is easy to construct a Givens rotation that transforms x into

$$\tilde{x}_G = (\tilde{x}_{G1}, 0, x_3, \ldots, x_n).$$

More generally, we can construct an orthogonal matrix, G_{pq}, similar to that shown in (3.13), that will transform the vector

$$x = (x_1, \ldots, x_p, \ldots, x_q, \ldots, x_n)$$

to

$$\tilde{x} = (x_1, \ldots, \tilde{x}_p, \ldots, 0, \ldots, x_n).$$

The orthogonal matrix that will do this is

$$Q_{pq}(\theta) = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ & & \ddots & & & & & & & & & \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \cos\theta & 0 & \cdots & 0 & \sin\theta & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ & & & & & & \ddots & & & & & \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & -\sin\theta & 0 & \cdots & 0 & \cos\theta & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ & & & & & & & & & & \ddots & \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}, \quad (3.14)$$

where the entries in the p^{th} and q^{th} rows and columns are

$$\cos\theta = \frac{x_p}{\|x\|}$$

and

$$\sin\theta = \frac{x_q}{\|x\|}.$$

A rotation matrix is the same as an identity matrix with four elements changed.

Considering x to be the p^{th} column in a matrix X, we can easily see how to zero out the q^{th} element of that column while affecting only the p^{th} and q^{th} rows and columns of X.

Just as we built the QR factorization by applying a succession of Householder reflections, we can also apply a succession of Givens rotations to achieve the factorization. If the Givens rotations are applied directly, however, the number of computations is about twice as many as for the Householder reflections. A succession of "fast Givens rotations" can be constructed, however, that will reduce the total number of computations by about one half. To see how this is done, first write the matrix Q in (3.13) as CT,

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & \tan\theta \\ -\tan\theta & 1 \end{bmatrix}. \quad (3.15)$$

If instead of working with matrices such as Q, which require 4 multiplications and 2 additions, we work with matrices such as T, involving the tangents, which require only 2 multiplications and 2 additions. The diagonal matrices such as C must be accumulated and multiplied in at some point. If this is done cleverly, the number of computations for Givens rotations is not much greater than that for Householder reflections. The fast Givens rotations must be performed with some care, otherwise excessive loss of accuracy can occur. See Golub and Van Loan (1996) for a discussion of the fast Givens transformations. The BLAS routines (see Section 5.1.1) `rotmg` and `rotm` respectively set up and apply fast Givens rotations.

3.2.6 Gram-Schmidt Transformations

Gram-Schmidt transformations yield a set of orthonormal vectors that span the same space as a given set of linearly independent vectors, $\{x_1, x_2, \ldots, x_m\}$. Application of these transformations is called Gram-Schmidt orthogonalization. If the given linearly independent vectors are the columns of a matrix A, the Gram-Schmidt transformations ultimately yield the QR factorization of A. The basic Gram-Schmidt transformation is shown in equation (2.23), page 74.

At the k^{th} stage of the Gram-Schmidt method, the vector $x_k^{(k)}$ is taken as $x_k^{(k-1)}$ and the vectors $x_{k+1}^{(k)}, x_{k+2}^{(k)}, \ldots, x_m^{(k)}$ are all made orthogonal to $x_k^{(k)}$. After the first stage all vectors have been transformed. (This method is sometimes called "modified Gram-Schmidt", because some people have performed the basic transformations in a different way, so that at the k^{th} iteration, starting at $k = 2$, the first $k-1$ vectors are unchanged, i.e., $x_i^{(k)} = x_i^{(k-1)}$ for $i = 1, 2, \ldots, k-1$, and $x_k^{(k)}$ is made orthogonal to the $k-1$ previously orthogonalized vectors $x_1^{(k)}, x_2^{(k)}, \ldots, x_{k-1}^{(k)}$. This method is called "classical Gram-Schmidt", for no particular reason. The "classical" method is not as stable, and should not be used. See Rice, 1966, and Björck, 1967, for discussions.) In the following, "Gram-Schmidt" is the same as what is sometimes called "modified Gram-Schmidt".

The Gram-Schmidt algorithm for forming the QR factorization is just a simple extension of equation (2.23); see Exercise 3.9 on page 119.

3.2.7 Singular Value Factorization

Another useful factorization is the singular value decomposition shown in (2.13), page 69. For the $n \times m$ matrix A, this is

$$A = U\Sigma V^{\mathrm{T}},$$

where U is an $n \times n$ orthogonal matrix, V is an $m \times m$ orthogonal matrix, and Σ is a diagonal matrix of the singular values. Golub and Kahan (1965) showed how to use a QR-type factorization to compute a singular value decomposition. This method, with refinements as presented in Golub and Reinsch (1970), is

the best algorithm for singular value decomposition. We discuss this method in Section 4.4, on page 131.

3.2.8 Choice of Direct Methods

An important consideration for the various direct methods is the efficiency of the method for certain patterned matrices. If a matrix begins with many zeros, it is important to preserve zeros to avoid unnecessary computations. Pissanetzky (1984) discusses some of the ways of doing this. The iterative methods discussed in the next section are often more useful for sparse matrices.

Another important consideration is how easily an algorithm lends itself to implementation on advanced computer architectures. Many of the algorithms for linear algebra can be vectorized easily. It is now becoming more important to be able to parallelize the algorithms (see Quinn, 1994). The iterative methods discussed in the next section can often be parallelized more easily.

3.3 Iterative Methods

An iterative method for solving the linear system $Ax = b$ obtains the solution by a sequence of successive approximations.

3.3.1 The Gauss-Seidel Method with Successive Overrelaxation

One of the simplest iterative procedures is the *Gauss-Seidel method*. In this method, we begin with an initial approximation to the solution, $x^{(0)}$. We then compute an update for the first element of x:

$$x_1^{(1)} = \frac{1}{a_{11}} \left(b_1 - \sum_{j=2}^{n} a_{1j} x_j^{(0)} \right).$$

Continuing in this way for the other elements of x, we have for $i = 1, \ldots, n$

$$x_i^{(1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(0)} \right),$$

where no sums are performed if the upper limit is smaller than the lower limit.

After getting the approximation $x^{(1)}$, we then continue this same kind of iteration for $x^{(2)}, x^{(3)}, \ldots$. We continue the iterations until a convergence criterion is satisfied. As we discussed on page 37, this criterion may be of the form

$$\Delta(x^{(k)}, x^{(k-1)}) \leq \epsilon,$$

where $\Delta(x^{(k)}, x^{(k-1)})$ is a measure of the difference of $x^{(k)}$ and $x^{(k-1)}$, such as $\|x^{(k)} - x^{(k-1)}\|$. We may also base the convergence criterion on $\|r^{(k)} - r^{(k-1)}\|$, where $r^{(k)} = b - Ax^{(k)}$.

The Gauss-Seidel iterations can be thought of as beginning with a rearrangement of the original system of equations as

$$\begin{array}{rcccccccc}
a_{11}x_1 & & & & & = & b_1 & - & a_{12}x_2 \quad \cdots - \quad a_{1n}x_n \\
a_{21}x_1 & + & a_{22}x_2 & & & = & b_2 & & \cdots - \quad a_{2n}x_n \\
\vdots & + & \vdots & & \vdots & & \vdots & & \\
a_{(n-1)1}x_1 & + & a_{(n-1)2}x_2 & + \cdots & & = & b_{n-1} & & - \quad a_{nn}x_n \\
a_{n1}x_1 & + & a_{n2}x_2 & + \cdots + & a_{nn}x_n & = & b_n & &
\end{array}$$

In this form, we identify three matrices – a diagonal matrix D, a lower triangular L with 0's on the diagonal, and an upper triangular U with 0's on the diagonal:

$$(D + L)x = b - Ux.$$

We can write this entire sequence of Gauss-Seidel iterations in terms of these three fixed matrices,

$$x^{(k+1)} = (D + L)^{-1}(-Ux^{(k)} + b). \tag{3.16}$$

This method will converge for any arbitrary starting value $x^{(0)}$ if and only if the spectral radius of $(D + L)^{-1}U$ is less than 1. (See Golub and Van Loan, 1996, for a proof of this.) Moreover, the rate of convergence increases with decreasing spectral radius.

Gauss-Seidel may be unacceptably slow, so it may be modified so that the update is a weighted average of the regular Gauss-Seidel update and the previous value. This kind of modification is called *successive overrelaxation*, or *SOR*. The update is given by

$$\frac{1}{\omega}(D + L)x^{(k+1)} = \frac{1}{\omega}\left((1 - \omega)D - \omega U\right)x^{(k)} + b,$$

where the relaxation parameter ω is usually chosen between 0 and 1. For $\omega = 1$ the method is the ordinary Gauss-Seidel method. See Exercises 3.3c, 3.3d, and 3.3e.

3.3.2 Solution of Linear Systems as an Optimization Problem; Conjugate Gradient Methods

The problem of solving the linear system $Ax = b$ is equivalent to finding the minimum of the function

$$f(x) = \frac{1}{2}x^{\mathrm{T}}Ax - x^{\mathrm{T}}b. \tag{3.17}$$

By setting the derivative of f to 0, we see that a stationary point of f occurs at x such that $Ax = b$ (see Section 2.1.18, page 79). If A is nonsingular, the minimum of f is at $x = A^{-1}b$, and the value of f at the minimum is $-\frac{1}{2}b^{\mathrm{T}}Ab$.

The minimum point can be approached iteratively by starting at a point $x^{(0)}$, moving to a point $x^{(1)}$ that yields a smaller value of the function, and continuing to move to points yielding smaller values of the function. The k^{th} point is $x^{(k-1)}+\alpha_k d_k$, where α_k is a scalar and d_k is a vector giving the direction of the movement. Hence, for the k^{th} point we have the linear combination,

$$x^{(k)} = x^{(0)} + \alpha_1 d_1 + \cdots + \alpha_k d_k$$

The convergence criterion is based on $\|x^{(k)} - x^{(k-1)}\|$ or on $\|r^{(k)} - r^{(k-1)}\|$, where $r^{(k)} = b - Ax^{(k)}$.

At the point $x^{(k)}$, the function f decreases most rapidly in the direction of the negative gradient, $-\nabla f(x^{(k)})$. The negative gradient is just the residual,

$$r^{(k)} = b - Ax^{(k)}.$$

If this residual is 0, no movement is indicated, because we are at the solution. Moving in the direction of steepest descent may cause a slow convergence to the minimum. (The curve that leads to the minimum on the quadratic surface is obviously not a straight line.)

A good choice for the sequence of directions $d_1, d_2, \ldots$ is such that

$$d_k^{\mathrm{T}} A d_i = 0, \quad \text{for } i = 1, \ldots, k-1.$$

Such a vector d_k is said to be A *conjugate* to $d_1, d_2, \ldots d_{k-1}$. The path defined by the directions $d_1, d_2, \ldots$ and the distances $\alpha_1, \alpha_2, \ldots$ is called the conjugate gradient. A conjugate gradient method for solving the linear system is shown in Algorithm 3.2.

Algorithm 3.2 The Conjugate Gradient Method for Solving $Ax = b$, Starting with $x^{(0)}$

0. Set $k = 0$; $r^{(k)} = b - Ax^{(k)}$; $s^{(k)} = A^{\mathrm{T}}r^{(k)}$; $p^{(k)} = s^{(k)}$; and $\gamma^{(k)} = \|s^{(k)}\|_2^2$.
1. If $\gamma^{(k)} \leq \epsilon$, set $x = x^{(k)}$ and terminate.
2. Set $q^{(k)} = Ap^{(k)}$.
3. Set $\alpha^{(k)} = \frac{\gamma^{(k)}}{\|q^{(k)}\|_2^2}$.
4. Set $x^{(k+1)} = x^{(k)} + \alpha^{(k)}p^{(k)}$.
5. Set $r^{(k+1)} = r^{(k)} - \alpha^{(k)}q^{(k)}$.
6. Set $s^{(k+1)} = A^{\mathrm{T}}r^{(k+1)}$.
7. Set $\gamma^{(k+1)} = \|s^{(k+1)}\|_2^2$.

8. Set $p^{(k+1)} = s^{(k+1)} + \frac{\gamma^{(k+1)} p^{(k)}}{\gamma^{(k)}}$.

9. Set $k = k + 1$ and go to 1. ∎

For example, the function (3.17) arising from the system

$$\begin{bmatrix} 5 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 18 \\ 16 \end{bmatrix}$$

has level contours as shown in Figure 3.2, and the conjugate gradient method would move along the line shown, toward the solution at $x = (2, 4)$.

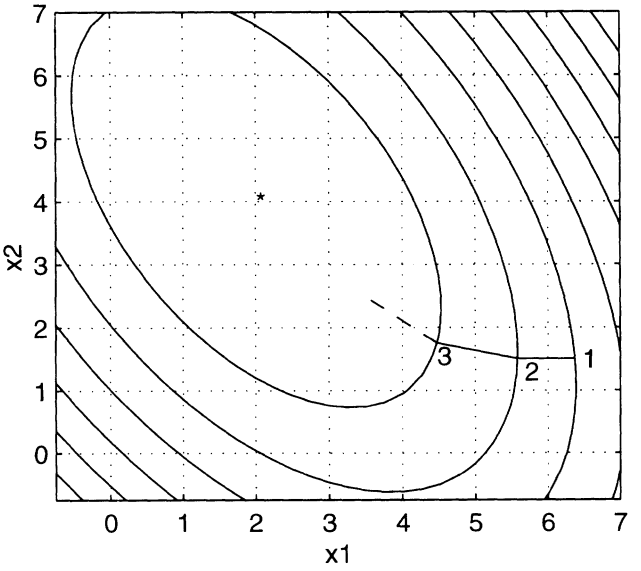


Figure 3.2: Solution of a Linear System Using a Conjugate Gradient Method

The conjugate gradient method and related procedures, called *Lanczos methods*, move through a *Krylov space* in the progression to the solution (see Freund, Golub, and Nachtigal, 1992). A Krylov space is the k-dimensional vector space of order n generated by the $n \times n$ matrix A and the vector v by forming the basis $\{v, Av, A^2v, \ldots, A^{k-1}v\}$. We often denote this space as $\mathcal{K}_k(A, v)$, or just as $\mathcal{K}_k$.

The generalized minimal residual (GMRES) method of Saad and Schultz (1986) for solving $Ax = b$ begins with an approximate solution $x^{(0)}$ and takes $x^{(k)}$ as $x^{(k-1)} + z^{(k)}$, where $z^{(k)}$ is the solution to the minimization problem,

$$\min_{z \in \mathcal{K}_k(A, r^{(k-1)})} \|r^{(k-1)} - Az\|,$$

where, as before, $r^{(k)} = b - Ax^{(k)}$. This minimization problem is a constrained least squares problem. In the original implementations, the convergence of

GMRES could be very slow, but modifications have speeded it up considerably. See Walker (1988) and Walker and Zhou (1994) for details of the methods. Brown and Walker (1997) consider the behavior of GMRES when the coefficient matrix is singular, and give conditions for GMRES to converge to a solution of minimum length (the solution corresponding to the Moore-Penrose inverse, see Section 3.7.2, page 113).

Iterative methods have important applications in solving differential equations. The solution of differential equations by a finite difference discretization involves the formation of a grid. The solution process may begin with a fairly coarse grid, on which a solution is obtained. Then a finer grid is formed, and the solution is interpolated from the coarser grid to the finer grid to be used as a starting point for a solution over the finer grid. The process is then continued through finer and finer grids. If all of the coarser grids are used throughout the process, the technique is a *multigrid* method. There are many variations of exactly how to do this. Multigrid methods are useful solution techniques for differential equations.

Iterative methods are particularly useful for large, sparse systems. Another advantage of many of the iterative methods is that they can be parallelized more readily (see Heath, Ng, and Peyton, 1991). An extensive discussion of iterative methods is given in Axelsson (1994).

3.4 Numerical Accuracy

The condition numbers we defined in Section 2.1 are useful indicators of the accuracy we may expect when solving a linear system, $Ax = b$. Suppose the entries of the matrix A and the vector b are accurate to approximately p decimal digits, so we have the system

$$(A + \delta A)(x + \delta x) = b + \delta b,$$

with

$$\frac{\|\delta A\|}{\|A\|} \approx 10^{-p}$$

and

$$\frac{\|\delta b\|}{\|b\|} \approx 10^{-p}.$$

Assume A is nonsingular, and suppose that the condition number with respect to inversion, $\kappa(A)$, is approximately 10^t, so

$$\kappa(A)\frac{\|\delta A\|}{\|A\|} \approx 10^{t-p}.$$

Ignoring the approximation of b, that is, assuming $\delta b = 0$, we can write

$$\delta x = -A^{-1}\delta A(x + \delta x),$$

which, together with the triangular inequality and inequality (2.19), page 72, yields the bound

$$\|\delta x\| \leq \|A^{-1}\| \|\delta A\| (\|x\| + \|\delta x\|).$$

Using equation (2.30) with this we have

$$\|\delta x\| \leq \kappa(A) \frac{\|\delta A\|}{\|A\|} (\|x\| + \|\delta x\|),$$

or

$$\left(1 - \kappa(A) \frac{\|\delta A\|}{\|A\|}\right) \|\delta x\| \leq \frac{\|\delta A\|}{\|A\|} \|x\|.$$

If the condition number is not too large relative to the precision, that is, if $10^{t-p} \ll 1$, then we have

$$\begin{aligned} \frac{\|\delta x\|}{\|x\|} &\approx \kappa(A) \frac{\|\delta A\|}{\|A\|} \\ &\approx 10^{t-p}. \end{aligned} \tag{3.18}$$

Expression (3.18) provides a rough bound on the accuracy of the solution in terms of the precision of the data and the condition number of the coefficient matrix. This result must be used with some care, however. Rust (1994), among others, points out failures of the condition number for setting bounds on the accuracy of the solution.

Another consideration in the practical use of (3.18) is the fact that the condition number is usually not known, and methods for computing it suffer from the same rounding problems as the solution of the linear system itself. In Section 3.8 we describe ways of estimating the condition number, but as the discussion there indicates, these estimates are often not very reliable.

We would expect the norms in the expression (3.18) to be larger for larger size problems. The approach taken above addresses a type of "total" error. It may be appropriate to scale the norms to take into account the number of elements. Chaitin-Chatelin and Frayssé (1996) discuss error bounds for individual elements of the solution vector and condition measures for elementwise error.

Another approach to determining the accuracy of a solution is to use random perturbations of A and/or b and then to estimate the effects of the perturbations on x. Stewart (1990) discusses ways of doing this. Stewart's method estimates error measured by a norm, as in expression (3.18). Kenney and Laub (1994) and Kenney, Laub, and Reese (1998) describe an estimation method to address elementwise error.

Higher accuracy in computations for solving linear systems can be achieved in various ways: multiple precision (Brent, 1978, Smith, 1991, and Bailey, 1993); interval arithmetic (Kulisch and Miranker, 1981 and 1983); and residue arithmetic (Szabó and Tanaka, 1967). Stallings and Boullion (1972) and Keller-McNulty and Kennedy (1986) describe ways of using residue arithmetic in some linear computations for statistical applications.

Another way of improving the accuracy is by use of iterative refinement, which we now discuss.

3.5 Iterative Refinement

Once an approximate solution, $x^{(0)}$, to the linear system $Ax = b$ is available, iterative refinement can yield a solution that is closer to the true solution. The residual

$$r = b - Ax^{(0)}$$

is used for iterative refinement. Clearly, if $h = A^+r$, then $x^{(0)} + h$ is a solution to the original system.

The problem considered here is not just an iterative solution to the linear system, as we discussed in Section 3.3. Here, we assume $x^{(0)}$ was computed accurately given the finite precision of the computer. In this case it is likely that r cannot be computed accurately enough to be of any help. If, however, r can be computed using a higher precision, then a useful value of h can be computed. This process can then be iterated as shown in Algorithm 3.3.

Algorithm 3.3 Iterative Refinement of the Solution to $Ax = b$, Starting with $x^{(0)}$

0. Set $k = 0$.
1. Compute $r^{(k)} = b - Ax^{(k)}$ in higher precision.
2. Compute $h^{(k)} = A^+r^{(k)}$.
3. Set $x^{(k+1)} = x^{(k)} + h^{(k)}$.
4. If $\|h^{(k)}\| > \epsilon\|x^{(k+1)}\|$, then
   4.a. set $k = k + 1$ and go to step 1;
   otherwise
   4.b. set $x = x^{(k+1)}$ and terminate. ∎

In step 2, if A is full rank then A^+ is A^{-1}. Also, as we have emphasized already, because we write an expression such as A^+r does not mean that we compute A^+. The norm in step 4 is usually chosen to be the ∞-norm. The algorithm may not converge, so it is necessary to have an alternative exit criterion, such as a maximum number of iterations.

Use of iterative refinement as a general-purpose method is severely limited by the need for higher precision in step 1. On the other hand, if computations in higher precision can be performed, they can be applied to step 2 — or just in the original computations for $x^{(0)}$. In terms of both accuracy and computational efficiency, use of higher precision throughout is usually better.

3.6 Updating a Solution

In applications of linear systems, it is often the case that after the system $Ax = b$ has been solved, the right-hand side is changed, and the system $Ax = c$ must be solved. If the linear system $Ax = b$ has been solved by a direct method using

one of the factorizations discussed in Section 3.2, the factors of A can be used to solve the new system $Ax = c$. If the right-hand side is a small perturbation of b, say $c = b + \delta b$, an iterative method can be used to solve the new system quickly, starting from the solution to the original problem.

If the coefficient matrix in a linear system $Ax = b$ is perturbed to result in the system $(A + \delta A)x = b$, it may be possible to use the solution x_0 to the original system to arrive efficiently at the solution to the perturbed system. One way, of course, is to use x_0 as the starting point in an iterative procedure. Often in applications, the perturbations are of a special type, such as

$$\tilde{A} = A - uv^{\mathrm{T}},$$

where u and v are vectors. (This is a "rank-one" perturbation of A, and when the perturbed matrix is used as a transformation, it is called a "rank-one" update. As we have seen, a Householder reflection is a special rank-one update.) Assuming A is an $n \times n$ matrix of full rank, it is easy to write $\tilde{A}^{-1}$ in terms of A^{-1}:

$$\tilde{A}^{-1} = A^{-1} + \alpha(A^{-1}u)(v^{\mathrm{T}}A^{-1}), \tag{3.19}$$

with

$$\alpha = \frac{1}{1 - v^{\mathrm{T}}A^{-1}u}.$$

These are called the Sherman-Morrison formulas (from J. Sherman and W. J. Morrison, 1950, "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix," *Annals of Mathematical Statistics* **21**, 124–127). $\tilde{A}^{-1}$ exists so long as $v^{\mathrm{T}}A^{-1}u \neq 1$. Because $x_0 = A^{-1}b$, the solution to the perturbed system is

$$\tilde{x}_0 = x_0 + \frac{(A^{-1}u)(v^{\mathrm{T}}x_0)}{(1 - v^{\mathrm{T}}A^{-1}u)}.$$

If the perturbation is more than rank one, that is, if the perturbation is

$$\tilde{A} = A - UV^{\mathrm{T}},$$

where U and V are $n \times m$ matrices with $n \geq m$, a generalization of the Sherman-Morrison formula, sometimes called the Woodbury formula, is

$$\tilde{A}^{-1} = A^{-1} + A^{-1}U(I_m - V^{\mathrm{T}}A^{-1}U)^{-1}V^{\mathrm{T}}A^{-1} \tag{3.20}$$

(from M. A. Woodbury, 1950, "Inverting Modified Matrices", Memorandum Report 42, Statistical Research Group, Princeton University). The solution to the perturbed system is easily seen to be

$$\tilde{x}_0 = x_0 + A^{-1}U(I_m - V^{\mathrm{T}}A^{-1}U)^{-1}V^{\mathrm{T}}x_0.$$

As we have emphasized many times, we rarely compute the inverse of a matrix, and so the Sherman-Morrison-Woodbury formulas are not used directly.

Because of having already solved $Ax = b$, it should be easy to solve another system, say $Ay = u_i$ where u_i is a column of U. If m is relatively small, as it is in most applications of this kind of update, there are not many systems $Ay = u_i$ to solve. Solving these systems, of course, yields $A^{-1}U$, the most formidable component of the Sherman-Morrison-Woodbury formula. The system to solve is of order m also.

Another situation that requires an update of a solution occurs when the system is augmented with additional equations and more variables:

$$\begin{bmatrix} A & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x \\ x_+ \end{bmatrix} = \begin{bmatrix} b \\ b_+ \end{bmatrix}$$

A simple way of obtaining the solution to the augmented system is to use the solution x_0 to the original system in an iterative method. The starting point for a method based on Gauss-Seidel or a conjugate gradient method can be taken as $(x_0, 0)$, or as $(x_0, x_+^{(0)})$ if a better value of $x_+^{(0)}$ is known.

In many statistical applications the systems are overdetermined, with A being $n \times m$ and $n > m$. In the next section we consider the problem of updating a least squares solution to an overdetermined system.

3.7 Overdetermined Systems; Least Squares

An overdetermined system may be written as

$$Ax \approx b, \tag{3.21}$$

where A is $n \times m$ and $n > m$. The problem is to determine a value of x that makes the approximation close, in some sense. We sometimes refer to this as "fitting" the system, which is referred to as a "model". Although in general there is no x that will make the system an equation, the system can be written as the equation

$$Ax = b + e,$$

where e is an n-vector of possibly arbitrary "errors".

A *least squares* solution $\widehat{x}$ to the system in (3.21) is one such that the Euclidean norm of the vector $b - Ax$ is minimized. By differentiating, we see that the minimum of the square of this norm,

$$(b - Ax)^{\mathrm{T}}(b - Ax), \tag{3.22}$$

occurs at $\widehat{x}$ that satisfies the square system

$$A^{\mathrm{T}}A\widehat{x} = A^{\mathrm{T}}b. \tag{3.23}$$

The system (3.23) is called the *normal equations*. As we mentioned in Section 3.2, because the condition number of $A^{\mathrm{T}}A$ is the square of the condition number of A, it may be better to work directly on A in (3.21) rather than to use

the normal equations. (In fact, this was the reason that we introduced the QR factorization for nonsquare matrices, because the LU and Cholesky factorizations had been described only for square matrices.) The normal equations are useful expressions, however, whether or not they are used in the computations. (This is another case where a formula does not define an algorithm, as with other cases we have encountered many times.)

It is interesting to note from equation (3.23) that the residual vector, $b - A\widehat{x}$, is orthogonal to each column in A:

$$A^{\mathrm{T}}(b - A\widehat{x}) = 0.$$

Overdetermined systems abound in statistical applications. The linear regression model is an overdetermined system. We discuss least squares solutions to the regression problem in Section 6.2.

3.7.1 Full Rank Coefficient Matrix

If A is of full rank, the least squares solution, from (3.23), is $\widehat{x} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}b$ and is obviously unique. A good way to compute this is to form the QR factorization of A.

First we write $A = QR$, as in (3.4) on page 95, where R is as in (3.5):

$$R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

with R_1 an $m \times m$ upper triangular matrix. The residual norm (3.22) can be written as

$$\begin{aligned} (b - Ax)^{\mathrm{T}}(b - Ax) &= (b - QRx)^{\mathrm{T}}(b - QRx) \\ &= (Q^{\mathrm{T}}b - Rx)^{\mathrm{T}}(Q^{\mathrm{T}}b - Rx) \\ &= (c_1 - R_1x)^{\mathrm{T}}(c_1 - R_1x) + c_2^{\mathrm{T}}c_2, \end{aligned} \tag{3.24}$$

where c_1 is a vector of length m and c_2 is a vector of length $n - m$, such that

$$Q^{\mathrm{T}}b = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}.$$

Because quadratic forms are nonnegative, the minimum of the residual norm in (3.24) occurs when $(c_1 - R_1x)^{\mathrm{T}}(c_1 - R_1x) = 0$, that is, when $(c_1 - R_1x) = 0$, or

$$R_1x = c_1. \tag{3.25}$$

We could also use the same technique of differentiation to find the minimum of (3.24) that we did to find the minimum of (3.22).

Because R_1 is triangular, the system is easy to solve: $\widehat{x} = R_1^{-1}c_1$.

We also see from (3.24) that the minimum of the residual norm is $c_2^{\mathrm{T}}c_2$. This is called the *residual sum of squares* in the least squares fit.

In passing, we note from (3.6) that $\widehat{x} = A^{+}b$.

3.7.2 Coefficient Matrix Not of Full Rank

If A is not of full rank, that is, if A has rank $r < m$, the least squares solution is not unique, and in fact, a solution is any vector $\widehat{x} = A^{-}b$, where A^{-} is any generalized inverse. For any generalized inverse A^{-}, the set of all solutions is

$$A^{-}b + (I - A^{-}A)z,$$

for an arbitrary vector z.

The solution whose L_2-norm $\|x\|_2$ is minimum is unique, however. That solution is the one corresponding to the Moore-Penrose inverse.

To see that this solution has minimum norm, first factor A, as in equation (3.10), page 96,

$$A = QRU^{\mathrm{T}},$$

and form the Moore-Penrose inverse, as in equation (3.12):

$$A^{+} = U \begin{bmatrix} R_1^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^{\mathrm{T}}.$$

Then

$$\widehat{x} = A^{+}b \tag{3.26}$$

is a least squares solution, just as in the full rank case. Now, let

$$Q^{\mathrm{T}}b = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix},$$

as above, except c_1 is of length r and c_2 is of length $n - r$, and let

$$U^{\mathrm{T}}x = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix},$$

where z_1 is of length r. We proceed as in the equations (3.24) (except here we use the L_2 norm notation). We seek to minimize $\|b - Ax\|_2$; and because multiplication by an orthogonal matrix does not change the norm, we have

$$\begin{aligned} \|b - Ax\|_2 &= \|Q^{\mathrm{T}}(b - AUU^{\mathrm{T}}x)\|_2 \\ &= \left\| \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} - \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right\|_2 \\ &= \left\| \begin{pmatrix} c_1 - R_1 z_1 \\ c_2 \end{pmatrix} \right\|_2 . \end{aligned} \tag{3.27}$$

The residual norm is minimized for $z_1 = R_1^{-1}c_1$ and z_2 arbitrary. However, if $z_2 = 0$, then $\|z\|_2$ is also minimized. Because $U^{\mathrm{T}}x = z$ and U is orthogonal, $\|\widehat{x}\|_2 = \|z\|_2$, and so $\|\widehat{x}\|_2$ is minimized.

3.7.3 Updating a Solution to an Overdetermined System

In the last section we considered the problem of updating a given solution to be a solution to a perturbed consistent system. An overdetermined system is often perturbed by adding either some rows or some columns to the coefficient matrix A. This corresponds to including additional equations in the system,

$$\begin{bmatrix} A \\ A_+ \end{bmatrix} x \approx \begin{bmatrix} b \\ b_+ \end{bmatrix},$$

or to adding variables,

$$\begin{bmatrix} A & A_+ \end{bmatrix} \begin{bmatrix} x \\ x_+ \end{bmatrix} \approx b.$$

In either case, if the QR decomposition of A is available, the decomposition of the augmented system can be computed readily. Consider, for example, the addition of k equations to the original system $Ax \approx b$, which has n approximate equations. With the QR decomposition, for the original full rank system, putting $Q^{\mathrm{T}}A$ and $Q^{\mathrm{T}}b$ as partitions in a matrix, we have

$$\begin{bmatrix} R_1 & c_1 \\ 0 & c_2 \end{bmatrix} = Q^{\mathrm{T}} \begin{bmatrix} A & b \end{bmatrix}.$$

Augmenting this with the additional rows yields

$$\begin{bmatrix} R & c_1 \\ 0 & c_2 \\ A_+ & b_+ \end{bmatrix} = \begin{bmatrix} Q^{\mathrm{T}} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} A & b \\ A_+ & b_+ \end{bmatrix}. \tag{3.28}$$

All that is required now is to apply orthogonal transformations, such as Givens rotations, to the system (3.28) to produce

$$\begin{bmatrix} R_* & c_{1*} \\ 0 & c_{2*} \end{bmatrix},$$

where R_* is an $m \times m$ upper triangular matrix and c_{1*} is an m-vector as before, but c_{2*} is an $(n - m + k)$-vector.

The updating is accomplished by applying m rotations to (3.28) so as to zero out the $(n+q)^{\text{th}}$ row, for $q = 1, 2, \ldots, k$. These operations go through an outer loop with $p = 1, 2, \ldots, n$, and an inner loop with $q = 1, 2, \ldots, k$. The operations rotate R through a sequence $R^{(p,q)}$ into R_*, and they rotate A_+ through a sequence $A_+^{(p,q)}$ into 0. At the p, q step, the rotation matrix Q_{pq} corresponding to (3.14), page 101, has

$$\cos\theta = \frac{R_{pp}^{(p,q)}}{r}$$

and

$$\sin\theta = \frac{(A_+^{(p,q)})_{q,p}}{r},$$

where

$$r = \sqrt{(R_{pp}^{(p,q)})^2 + ((A_{+}^{(p,q)})_{q,p})^2}.$$

Gentleman (1974) and Miller (1992) give Fortran programs that implement this kind of updating. The software from *Applied Statistics* is available in `statlib` (see page 199).

3.8 Other Computations for Linear Systems

3.8.1 Rank Determination

It is often easy to determine that a matrix is of full rank. If the matrix is not of full rank, however, or if it is very ill-conditioned, it is difficult to determine its rank. This is because the computations to determine the rank eventually approximate 0. It is difficult to approximate 0; the relative error (if defined) would be either 0 or infinite. The rank revealing QR factorization (equation (3.10), page 96) is the preferred method to estimate the rank. When this decomposition is used to estimate the rank, it is recommended that complete pivoting be used in computing the decomposition. The LDU decomposition, described on page 92, can be modified the same way we used the modified QR to estimate the rank of a matrix. Again, it is recommended that complete pivoting be used in computing the decomposition.

3.8.2 Computing the Determinant

The determinant of a square matrix can be obtained easily as the product of the diagonal elements of the triangular matrix in any factorization that yields an orthogonal matrix times a triangular matrix. As we have stated before, it is not often that the determinant need be computed, however. One application in statistics is in optimal experimental designs. The D-optimal criterion, for example, chooses the design matrix, X, such that $|X^{\mathrm{T}}X|$ is maximized (see Section 6.2).

3.8.3 Computing the Condition Number

The computation of a condition number of a matrix can be quite involved. Various methods have been proposed to estimate the condition number using relatively simple computations. Cline et al. (1979) suggest a method that is easy to perform and is widely used. For a given matrix A and some vector v, solve

$$A^{\mathrm{T}}x = v,$$

and then

$$Ay = x.$$

By tracking the computations in the solution of these systems, Cline et al. conclude that

$$\frac{\|y\|}{\|x\|}$$

is approximately equal to, but less than, $\|A^{-1}\|$. This estimate is used with respect to the L_1 norm in LINPACK, but the approximation is valid for any norm. Solving the two systems above probably does not require much additional work because the original problem was likely to solve $Ax = b$, and solving a system with multiple right-hand sides can be done efficiently using the solution to one of the right-hand sides. The approximation is better if v is chosen so that $\|x\|$ is as large as possible relative to $\|v\|$.

Stewart (1980) and Cline and Rew (1983) investigated the validity of the approximation. The LINPACK estimator can underestimate the true condition number considerably, although generally not by an order of magnitude. Cline, Conn, and Van Loan (1982) give a method of estimating the L_2 condition number of a matrix that is a modification of the L_1 condition number used in LINPACK. This estimate generally performs better than the L_1 estimate, but the Cline/Conn/Van-Loan estimator still can have problems (see Bischof, 1990).

Hager (1984) gives another method for an L_1 condition number. Higham (1988) provides an improvement of Hager's method, given as Algorithm 3.4 below, which is used in LAPACK.

Algorithm 3.4 The Hager/Higham LAPACK Condition Number Estimator γ of the $n \times n$ Matrix A

Assume $n > 1$; else $\gamma = |A1|$. (All norms are L_1 unless specified otherwise.)

0. Set $k = 1$; $v^{(k)} = \frac{1}{n}A1$; $\gamma^{(k)} = \|v^{(k)}\|$; and $x^{(k)} = A^{\mathrm{T}}\mathrm{sign}(v^{(k)})$.
1. Set $j = \min\{i : |x_i^{(k)}| = \|x^{(k)}\|_\infty\}$.
2. Set $k = k + 1$.
3. Set $v^{(k)} = Ae_j$.
4. Set $\gamma^{(k)} = \|v^{(k)}\|$.
5. If $\mathrm{sign}(v^{(k)}) = \mathrm{sign}(v^{(k-1)})$ or $\gamma^{(k)} \leq \gamma^{(k-1)}$, then go to step 8.
6. Set $x^{(k)} = A^{\mathrm{T}}\mathrm{sign}(v^{(k)})$.
7. If $\|x^{(k)}\|_\infty \neq x_j^{(k)}$ and $k \leq k_{\max}$ then go to step 1.
8. For $i = 1, 2, \ldots, n$, set $x_i = (-1)^{i+1}\left(1 + \frac{i-1}{n-1}\right)$.
9. Set $x = Ax$.
10. If $\frac{2\|x\|}{(3n)} > \gamma^{(k)}$, set $\gamma^{(k)} = \frac{2\|x\|}{(3n)}$.
11. Set $\gamma = \gamma^{(k)}$. ∎

Higham (1987) compares Hager's condition number estimator with that of Cline et al. (1979) and finds that the Hager LAPACK estimator is generally more useful. Higham (1990) gives a survey and comparison of the various ways of estimating and computing condition numbers. You are asked to study the performance of the LAPACK estimate using Monte Carlo in Exercise 3.12c, page 121.

Exercises

3.1. Let $A = LU$ be the LU decomposition of the $n \times n$ matrix A.

(a) Suppose we multiply the j^{th} column of A by c_j, $j = 1, 2, \ldots n$, to form the matrix A_c. What is the LU decomposition of A_c? Try to express your answer in a compact form.

(b) Suppose we multiply the i^{th} row of A by c_i, $i = 1, 2, \ldots n$, to form the matrix A_r. What is the LU decomposition of A_r? Try to express your answer in a compact form.

(c) What application might these relationships have?

3.2. Show that if A is positive definite, there exists a unique upper triangular matrix T with positive diagonal elements such that

$$A = T^{\mathrm{T}}T.$$

Hint: Show that $a_{ii} > 0$; show that if A is partitioned into square submatrices A_{11} and A_{22},

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

that A_{11} and A_{22} are positive definite; use Algorithm 3.1 (page 94) to show the existence of a T; and finally show that T is unique.

3.3. Consider the system of linear equations:

$$\begin{array}{rcrcrcr} x_1 & + & 4x_2 & + & x_3 & = & 12 \\ 2x_1 & + & 5x_2 & + & 3x_3 & = & 19 \\ x_1 & + & 2x_2 & + & 2x_3 & = & 9 \end{array}$$

(a) Solve the system using Gaussian elimination with partial pivoting.

(b) Solve the system using Gaussian elimination with complete pivoting.

(c) Determine the D, L, and U matrices of the Gauss-Seidel method (equation 3.16, page 104) and determine the spectral radius of

$$(D + L)^{-1}U.$$

(d) Do two steps of the Gauss-Seidel method starting with $x^{(0)} = (1, 1, 1)$, and evaluate the L_2 norm of the difference of two successive approximate solutions.

(e) Do two steps of the Gauss-Seidel method with successive overrelaxation using $\omega = 0.1$, starting with $x^{(0)} = (1, 1, 1)$, and evaluate the L_2 norm of the difference of two successive approximate solutions.

(f) Do two steps of the conjugate gradient method starting with $x^{(0)} = (1, 1, 1)$, and evaluate the L_2 norm of the difference of two successive approximate solutions.

3.4. Given the $n \times k$ matrix A and the k-vector b (where n and k are large), consider the problem of evaluating $c = Ab$. As we have mentioned, there are two obvious ways of doing this: (1) compute each element of c, one at a time, as an inner product $c_i = a_i^{\mathrm{T}} b = \sum_j a_{ij} b_j$, or (2) update the computation of all of the elements of c in the inner loop.

(a) What is the order of computations of the two algorithms?

(b) Why would the relative efficiencies of these two algorithms be different for different programming languages, such as Fortran and C?

(c) Suppose there are p processors available and the fan-in algorithm on page 83 is used to evaluate Ax as a set of inner products. What is the order of time of the algorithm?

(d) Give a heuristic explanation of why the computation of the inner products by a fan-in algorithm is likely to have less roundoff error than computing the inner products by a standard serial algorithm. (This does not have anything to do with the parallelism.)

(e) Describe how the following approach could be parallelized. (This is the second general algorithm mentioned above.)

```
for i = 1, ..., n
{
  c_i = 0
  for j = 1, ..., k
  {
    c_i = c_i + a_ij b_j
  }
}
```

(f) What is the order of time of the algorithms you described?

3.5. Consider the problem of evaluating $C = AB$, where A is $n \times m$ and B is $m \times q$. Notice that this multiplication can be viewed as a set of matrix/vector multiplications, so either of the algorithms in Exercise 3.4d above would be applicable. There is, however, another way of performing this multiplication, in which all of the elements of C could be evaluated simultaneously.

(a) Write pseudo-code for an algorithm in which the nq elements of C could be evaluated simultaneously. Do not be concerned with the parallelization in this part of the question.

(b) Now suppose there are nmq processors available. Describe how the matrix multiplication could be accomplished in $O(m)$ steps (where a step may be a multiplication and an addition). *Hint:* Use a fan-in algorithm.

3.6. Let X_1, X_2, and X_3 be independent random variables identically distributed as standard normals.

(a) Determine a matrix A such that the random vector

$$A \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

has a multivariate normal distribution with variance-covariance matrix,

$$\begin{bmatrix} 4 & 2 & 8 \\ 2 & 10 & 7 \\ 8 & 7 & 21 \end{bmatrix}.$$

(b) Is your solution unique? (The answer is no.) Determine a different solution.

3.7. Generalized inverses.

(a) Prove equation (3.6), page 95 (Moore-Penrose inverse of a full-rank matrix).

(b) Prove equation (3.9), page 96 (generalized inverse of a non-full-rank matrix).

(c) Prove equation (3.12), page 96, (Moore-Penrose inverse of a non-full-rank matrix).

3.8. Determine the Givens transformation matrix that will rotate the matrix

$$A = \begin{bmatrix} 3 & 5 & 6 \\ 6 & 1 & 2 \\ 8 & 6 & 7 \\ 2 & 3 & 1 \end{bmatrix}$$

so that the second column becomes $(5, \tilde{a}_{22}, 6, 0)$. (See Exercise 5.3.)

3.9. Gram-Schmidt transformations.

(a) Use Gram-Schmidt transformations to determine an orthonormal basis for the space spanned by the vectors

$$\begin{aligned} v_1 &= (3, 6, 8, 2) \\ v_2 &= (5, 1, 6, 3) \\ v_3 &= (6, 2, 7, 1) \end{aligned}$$

(b) Write out a formal algorithm for computing the QR factorization of the $n \times m$ full-rank matrix A. Assume $n \geq m$.

(c) Write a Fortran or C subprogram to implement the algorithm you described.

3.10. The normal equations.

(a) For any matrix A with real elements, show $A^{\mathrm{T}}A$ is nonnegative definite.

(b) For any $n \times m$ matrix A with real elements, and with $n < m$, show $A^{\mathrm{T}}A$ is not positive definite.

(c) Let A be an $n \times m$ matrix of full column rank. Show that $A^{\mathrm{T}}A$ is positive definite.

3.11. Solving an overdetermined system $Ax = b$, where A is $n \times m$.

(a) Count how many floating-point multiplications and additions (flops) are required to form $A^{\mathrm{T}}A$.

(b) Count how many flops are required to form $A^{\mathrm{T}}b$.

(c) Count how many flops are required to solve $A^{\mathrm{T}}A = A^{\mathrm{T}}b$ using a Cholesky decomposition.

(d) Count how many flops are required to form a QR decomposition of A using reflectors.

(e) Count how many flops are required to form a $Q^{\mathrm{T}}b$.

(f) Count how many flops are required to solve $R_1 x = c_1$ (equation (3.25), page 112).

(g) If n is large relative to m, what is the ratio of the total number of flops required to form and solve the normal equations using the Cholesky method to the total number required to solve the system using a QR decomposition. Why is the QR method generally preferred?

3.12. A Monte Carlo study of condition number estimators.

(a) Write a Fortran or C program to generate $n \times n$ random orthogonal matrices (following Stewart, 1980):

1. Generate $n-1$ independent i-vectors, $x_2, x_3, \ldots, x_n$ from $N_i(0, I_i)$. (x_i is of length i.)
2. Let $r_i = \|x_i\|_2$, and let $\tilde{H}_i$ be the $i \times i$ reflection matrix that transforms x_i into the i-vector $(r_i, 0, 0, \ldots, 0)$.
3. Let H_i be the $n \times n$ matrix

$$\begin{bmatrix} I_{n-i} & 0 \\ 0 & \tilde{H}_i \end{bmatrix},$$

and form the diagonal matrix,

$$J = \text{diag}\big((-1)^{b_1}, (-1)^{b_2}, \ldots, (-1)^{b_n}\big),$$

where the b_i are independent realizations of a Bernoulli random variable.

4. Deliver the orthogonal matrix $JH_1H_2 \cdots H_n$.

(b) Write a Fortran or C program to compute an estimate of the L_1 LAPACK condition number of a matrix using Algorithm 3.4 (page 116).

(c) Design and conduct a Monte Carlo study to assess the performance of the condition number estimator in the previous part. Consider a few different sizes of matrices, say 5×5, 10×10, and 20×20; and consider a range of condition numbers, say 10, 10^4, and 10^8. Generate random matrices with known L_2 condition numbers. An easy way to do that is to form a diagonal matrix, D, with elements $0 < d_1 \leq d_2 \leq \cdots \leq d_n$, and then generate random orthogonal matrices as described above. The L_2 condition number of the diagonal matrix is d_n/d_1. That is also the condition number of the random matrix UDV, where U and V are random orthogonal matrices. (See Stewart, 1980, for a Monte Carlo study of the performance of the LINPACK condition number estimator.)

Chapter 4

Computation of Eigenvectors and Eigenvalues and the Singular Value Decomposition

Before we discuss methods for computing eigenvalues, we mention an interesting observation. Consider the polynomial, $f(\lambda)$,

$$\lambda^p + a_{p-1}\lambda^{p-1} + \cdots + a_1\lambda + a_0.$$

Now form the matrix, A,

$$\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & \cdots & 1 \\ -a_0 & -a_1 & -a_2 & \cdots & -a_{p-1} \end{bmatrix}.$$

The matrix A is called the *companion matrix* of the polynomial f. It is easy to see that the characteristic equation of A, equation (2.11) on page 68, is the polynomial $f(\lambda)$:

$$\det(A - \lambda I) = f(\lambda).$$

Thus, given a general polynomial f, we can form a matrix A whose eigenvalues are the roots of the polynomial. It is a well-known fact in the theory of equations that there is no general formula for the roots of a polynomial of degree

greater than 4. This means that we cannot expect to have a direct method for calculating eigenvalues; rather, we will have to use an iterative method.

In statistical applications, the matrices whose eigenvalues are of interest are almost always symmetric. Because the eigenvalues of a symmetric (real) matrix are real, the problem of determining the eigenvalues of a symmetric matrix is simpler than the corresponding problem for a general matrix.

We describe three methods for computing eigenvalues — the power method, the Jacobi method, and the QR method. Each method has some desirable property for particular applications. A QR-type method can also be used effectively to evaluate singular values.

If v is an eigenvector of A, the corresponding eigenvalue is easy to determine; it is the common ratio $(Av)_i/v_i$. Likewise, if the eigenvalue λ is known, the corresponding eigenvector is the solution to the system

$$(A - \lambda I)v = 0.$$

4.1 Power Method

Let A be a real $n \times n$ symmetric matrix with eigenvalues λ_i indexed so that $|\lambda_1| \leq |\lambda_2| \leq \cdots |\lambda_n|$, with corresponding unit eigenvectors v_i. We restrict our attention to simple matrices (see page 68), and assume that $\lambda_{n-1} < \lambda_n$ (i.e., λ_n and v_n are unique). In this case λ_n is called the *dominant eigenvalue* and v_n is called the *dominant eigenvector*. Now let $x^{(0)}$ be an n-vector that is not orthogonal to v_n.

Because A is assumed to be simple, $x^{(0)}$ can be represented as a linear combination of the eigenvectors:

$$x^{(0)} = c_1 v_1 + c_2 v_2 + \cdots + c_n v_n,$$

and because $x^{(0)}$ is not orthogonal to v_n, $c_n \neq 0$. The power method is based on a sequence that continues the finite Krylov space generating set:

$$x^{(0)}, Ax^{(0)}, A^2x^{(0)}, \ldots$$

From the relationships above and the definition of eigenvalues and eigenvectors, we have

$$\begin{aligned}
Ax^{(0)} &= c_1 Av_1 + c_2 Av_2 + \cdots + c_n Av_n \\
&= c_1\lambda_1 v_1 + c_2\lambda_2 v_2 + \cdots + c_n\lambda_n v_n \\
A^2x^{(0)} &= c_1\lambda_1^2 v_1 + c_2\lambda_2^2 v_2 + \cdots + c_n\lambda_n^2 v_n \\
\cdots &= \cdots \\
A^jx^{(0)} &= c_1\lambda_1^j v_1 + c_2\lambda_2^j v_2 + \cdots + c_n\lambda_n^j v_n \\
&= \lambda_n^j \left(c_1 \left(\frac{\lambda_1}{\lambda_n}\right)^j v_1 + c_2 \left(\frac{\lambda_2}{\lambda_n}\right)^j v_2 + \cdots + c_n v_n \right). \qquad (4.1)
\end{aligned}$$

To simplify the notation, let $u^{(j)} = A^j x^{(0)}/\lambda_n^j$ (or, equivalently, $u^{(j)} = Au^{(j-1)}/\lambda_n$). From (4.1) and the fact that $|\lambda_i| < |\lambda_n|$ for $i < n$, we see that $u^{(j)} \to c_n v_n$, which is the unnormalized dominant eigenvector.

We have the bound

$$\begin{aligned}
\|u^{(j)} - c_n v_n\| &= \|c_1 \left(\frac{\lambda_1}{\lambda_n}\right)^j v_1 + c_2 \left(\frac{\lambda_2}{\lambda_n}\right)^j v_2 + \cdots \\
&\quad + c_{n-1} \left(\frac{\lambda_{n-1}}{\lambda_n}\right)^j v_{n-1}\| \\
&\leq |c_1| \left|\frac{\lambda_1}{\lambda_n}\right|^j \|v_1\| + |c_2| \left|\frac{\lambda_2}{\lambda_n}\right|^j \|v_2\| + \cdots \\
&\quad + |c_{n-1}| \left|\frac{\lambda_{n-1}}{\lambda_n}\right|^j \|v_{n-1}\| \\
&\leq (|c_1| + |c_2| + \cdots + |c_{n-1}|) \left|\frac{\lambda_{n-1}}{\lambda_n}\right|^j . \qquad (4.2)
\end{aligned}$$

The last expression results from the facts that $|\lambda_i| \leq |\lambda_{n-1}|$ for $i < n-1$ and that the v_i are unit vectors.

From (4.2), we see that the norm of the difference of $u^{(j)}$ and $c_n v_n$ decreases by a factor of approximately $|\lambda_{n-1}/\lambda_n|$ with each iteration; hence, this ratio is an important indicator of the rate of convergence of $u^{(j)}$ to the dominant eigenvector.

If $|\lambda_{n-2}| < |\lambda_{n-1}| < |\lambda_n|$, $c_{n-1} \neq 0$, and $c_n \neq 0$ the power method converges linearly; that is,

$$0 < \lim_{j \to \infty} \frac{\|x^{(j+1)} - c_n v_n\|}{\|x^{(j)} - c_n v_n\|} < 1 \qquad (4.3)$$

(see Exercise 4.1c, page 134).

If an approximate value of the eigenvector v_n is available and $x^{(0)}$ is taken to be that approximate value, the convergence will be faster. If an approximate value of the dominant eigenvector, $\widehat{\lambda}_n$, is available, starting with any $y^{(0)}$, a few iterations on

$$(A - \widehat{\lambda}_n I) y^{(k)} = y^{(k-1)}$$

may yield a better starting value for $x^{(0)}$.

Once the dominant eigenvector is determined, the dominant eigenvalue λ_n can be easily determined.

In some applications, only the dominant eigenvalue is of interest. If other eigenvalues are needed, however, we can use the known dominant eigenvalue and eigenvector to compute the others. Whenever one eigenvalue, λ_i, and corresponding eigenvector, v_i, of a matrix A are available, another matrix can be formed that has all the same nonzero eigenvalues and corresponding eigenvectors as A, except for the i^{th} one. To see how to do this, let λ_j be an eigenvalue

of A such that $\lambda_j \neq \lambda_i$. Now, λ_j is also an eigenvalue of A^{T} (see the properties listed on page 68). Let w_j be the corresponding eigenvector of A^{T}. Now,

$$\langle Av_i, w_j \rangle = \langle \lambda_i v_i, w_j \rangle = \lambda_i \langle v_i, w_j \rangle.$$

But also,

$$\langle Av_i, w_j \rangle = \langle v_i, A^{\mathrm{T}} w_j \rangle = \langle v_i, \lambda_j w_j \rangle = \lambda_j \langle v_i, w_j \rangle.$$

But if

$$\lambda_i \langle v_i, w_j \rangle = \lambda_j \langle v_i, w_j \rangle,$$

and $\lambda_j \neq \lambda_i$, then $\langle v_i, w_j \rangle = 0$. Now let w_i be the eigenvector of A^{T} corresponding to λ_i, and consider the matrix

$$B = A - \lambda_i w_i w_i^{\mathrm{T}}.$$

We see that

$$\begin{aligned} Bw_j &= Aw_j - \lambda_i w_i w_i^{\mathrm{T}} w_j \\ &= Aw_j \\ &= \lambda_j w_j, \end{aligned}$$

so λ_j and w_j are respectively an eigenvalue and an eigenvector of B. This gives us a way to use the power method to find the second largest eigenvalue once the largest one is found.

If A is nonsingular, we can also use the power method on A^{-1} to determine the smallest eigenvalue of A.

4.2 Jacobi Method

The Jacobi method for determining the eigenvalues of a simple symmetric matrix A uses a sequence of orthogonal similarity transformations that eventually result in the transformation

$$A = P\Lambda P^{-1},$$

or

$$\Lambda = P^{-1} A P,$$

where Λ is diagonal. Recall that similar matrices have the same eigenvalues.

The matrices for the similarity transforms are the Givens rotation or Jacobi rotation matrices discussed on page 99. The general form of one of these orthogonal matrices, $Q_{pq}(\theta)$, given in (3.14) on page 101, is the identity matrix with $\cos\theta$ in the $(p,p)^{\text{th}}$ and $(q,q)^{\text{th}}$ positions, $\sin\theta$ in the $(p,q)^{\text{th}}$ position, and $-\sin\theta$ in the $(q,p)^{\text{th}}$ position:

$$Q_{pq}(\theta) = \begin{array}{c} \\ p \\ \\ q \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} & & p & & q & \\ \end{array} \\ \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & 0 & \sin\theta & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix} \end{array}.$$

The Jacobi iteration is

$$A^{(k)} = Q_{p_k q_k}^{\mathrm{T}}(\theta_k) A^{(k-1)} Q_{p_k q_k}(\theta_k),$$

where p_k, q_k, and θ_k are chosen so that the $A^{(k)}$ is "more diagonal" than $A^{(k-1)}$. Specifically, the iterations will be chosen so as to reduce the sum of the squares of the off-diagonal elements, which for any square matrix A is

$$\|A\|_F^2 - \sum_i a_{ii}^2.$$

The orthogonal similarity transformations preserve the Frobenius norm

$$\|A^{(k)}\|_F = \|A^{(k-1)}\|_F.$$

Because the rotation matrices change only the elements in the $(p,p)^{\text{th}}$, $(q,q)^{\text{th}}$, and $(p,q)^{\text{th}}$ positions (and also the $(q,p)^{\text{th}}$ position since both matrices are symmetric), we have

$$(a_{pp}^{(k)})^2 + (a_{qq}^{(k)})^2 + 2(a_{pq}^{(k)})^2 \;=\; (a_{pp}^{(k-1)})^2 + (a_{qq}^{(k-1)})^2 + 2(a_{pq}^{(k-1)})^2.$$

The off-diagonal sum of squares at the k^{th} stage in terms of that at the $(k-1)^{\text{th}}$ stage is

$$\begin{aligned} \|A^{(k)}\|_F^2 - \sum_i (a_{ii}^{(k)})^2 &= \|A^{(k-1)}\|_F^2 - \sum_{i \neq p,q} (a_{ii}^{(k-1)})^2 - \left((a_{pp}^{(k)})^2 + (a_{qq}^{(k)})^2 \right) \\ &= \|A^{(k-1)}\|_F^2 - \sum_{i \neq p,q} (a_{ii}^{(k-1)})^2 - (2a_{pq}^{(k-1)})^2 + (a_{pq}^{(k)})^2. \end{aligned} \tag{4.4}$$

Hence, for a given index pair, (p,q), at the k^{th} iteration the sum of the squares of the off-diagonal elements is minimized by choosing the rotation matrix so that

$$a_{pq}^{(k)} = 0. \tag{4.5}$$

As we saw on page 101, it is easy to determine the angle θ so as to introduce a zero in a single Givens rotation. Here, we are using the rotations in a similarity transformation, so it is a little more complicated.

The requirement that $a_{pq}^{(k)} = 0$ implies

$$a_{pq}^{(k-1)} \left(\cos^2\theta - \sin^2\theta \right) + \left(a_{pp}^{(k-1)} - a_{qq}^{(k-1)} \right) \cos\theta \sin\theta = 0. \tag{4.6}$$

Using the trigonometric identities

$$\begin{aligned} \cos(2\theta) &= \cos^2\theta - \sin^2\theta \\ \sin(2\theta) &= 2\cos\theta \sin\theta, \end{aligned}$$

in (4.6), we have

$$\tan(2\theta) = \frac{2a_{pq}^{(k-1)}}{a_{pp}^{(k-1)} - a_{qq}^{(k-1)}},$$

which yields a unique angle in $[-\pi/4, \pi/4]$. Of course, the quantities we need are $\cos\theta$ and $\sin\theta$, not the angle itself. First, using the identity

$$\tan\theta = \frac{\tan(2\theta)}{1 + \sqrt{1 + \tan^2(2\theta)}},$$

we get $\tan\theta$ from $\tan(2\theta)$; and then from $\tan\theta$, we can compute the quantities required for the rotation matrix $Q_{pq}(\theta)$:

$$\begin{aligned} \cos\theta &= \frac{1}{\sqrt{1 + \tan^2\theta}} \\ \sin\theta &= \cos\theta \tan\theta. \end{aligned}$$

Convergence occurs when the off-diagonal elements are sufficiently small. The quantity (4.4) using the Frobenius norm is the usual value to compare with a convergence criterion, ϵ.

From (4.5) we see that the best index pair, (p, q), is such that

$$|a_{pq}^{(k-1)}| = \max_{i<j} |a_{ij}^{(k-1)}|.$$

If this choice is made, the Jacobi method can be shown to converge. The method with this choice is called the *classical Jacobi* method.

For an $n \times n$ matrix, the number of operations to identify the maximum off-diagonal is $O(n^2)$. The computations for the similarity transform itself are only $O(n)$ because of the sparsity of the rotators. Of course the computations for the similarity transformations are more involved than those to identify the maximum off-diagonal, so for small n, the classical Jacobi method should be used. If n is large, however, it may be better not to spend time looking for the maximum off-diagonal. Various *cyclic Jacobi* methods have been proposed, in which the pairs (p, q) are chosen systematically without regard to the magnitude of the off-diagonal being zeroed. Depending on the nature of the cyclic Jacobi method, it may or may not be guaranteed to converge. For certain schemes, quadratic convergence has been proven; for at least one other scheme, an example showing failure of convergence has been given. See Watkins (1991) for a discussion of the convergence issues.

The Jacobi method is one of the oldest algorithms for computing eigenvalues, and has recently become important again because it lends itself to easy implementation on parallel processors.

Notice that at the k^{th} iteration, only two rows and two columns of $A^{(k)}$ are modified. This is what allows the Jacobi method to be performed in parallel. We can form $\lfloor n/2 \rfloor$ pairs, and do $\lfloor n/2 \rfloor$ rotations simultaneously. Thus, each parallel iteration consists of a choice of a set of index pairs and then a batch

of rotations. Although, as we have indicated, the convergence may depend on which rows are chosen for the rotations, if we are to achieve much efficiency by performing the operations in parallel, we cannot spend much time in deciding how to form the pairs for the rotations. Various schemes have been suggested for forming the pairs for a parallel iteration. Luk and Park (1989) have analyzed some of the proposed schemes. A simple scheme, called "mobile Jacobi" (see Watkins, 1991), is

1. perform $\lfloor n/2 \rfloor$ rotations using the pairs

$$(1,2),\ (3,4),\ (5,6),\ldots$$

2. interchange all rows and columns that were rotated
3. perform $\lfloor (n-1)/2 \rfloor$ rotations using the pairs

$$(2,3),\ (4,5),\ (6,7),\ldots$$

4. interchange all rows and columns that were rotated
5. if convergence has not been achieved, go to 1.

The notation above that specifies the pairs refers to the rows and columns at the current state; that is, after the interchanges up to that point. The interchange operation is a similarity transformation using an elementary permutation matrix (see page 65), hence the eigenvalues are left unchanged by this operation.

4.3 QR Method for Eigenanalysis

The most common algorithm for extracting eigenvalues is the QR method. The method devised by Francis (1961a, 1961b) can be used for nonsymmetric matrices. It is simpler for symmetric matrices, of course, because the eigenvalues are real. Also for symmetric matrices the computer storage is less, the computations are fewer, and some transformations are particularly simple.

The QR method requires that the matrix first be transformed into *upper Hessenberg form*. A matrix is in upper Hessenberg form, and is called a *Hessenberg matrix*, if it is upper triangular except for the first subdiagonal, which may be nonzero. That is, $a_{ij} = 0$ for $i > j + 1$:

$$\begin{bmatrix} \mathtt{X} & \mathtt{X} & \mathtt{X} & \cdots & \mathtt{X} & \mathtt{X} \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \cdots & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{X} & \mathtt{X} & \cdots & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{X} & \cdots & \mathtt{X} & \mathtt{X} \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \cdots & \mathtt{X} & \mathtt{X} \end{bmatrix}.$$

A matrix can be reduced to Hessenberg form in a finite number of similarity transformations, using either Householder reflections or Givens rotations.

The Hessenberg form for a symmetric matrix is tridiagonal. The Hessenberg form allows a large savings in the subsequent computations, even for nonsymmetric matrices.

The QR method for determining the eigenvalues is iterative and produces a sequence of Hessenberg matrices $A^{(0)}, A^{(1)}, A^{(2)}, \ldots$, which converges to a triangular matrix.

The eigenvalues of a matrix in upper Hessenberg form are extracted by a process called "chasing", which consists of steps that alternate between creating nonzero entries in positions $(i+2,i)$, $(i+3,i)$, and $(i+3,i+1)$ and restoring these entries to zero, as the nonzero entries are moved farther down the matrix. For example,

$$\begin{bmatrix} \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{Y} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{Y} & \mathtt{Y} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} \end{bmatrix} \rightarrow \begin{bmatrix} \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{Y} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{Y} & \mathtt{Y} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} \end{bmatrix}.$$

In the j^{th} step of the QR method, a bulge is created and is chased down the matrix by similarity transformations, usually Givens transformations,

$$G_k^{-1} A^{(j-1,k)} G_k.$$

The transformations are based on the eigenvalues of 2×2 matrices in the lower right-hand part of the matrix.

There are some variations on the way the chasing occurs. Haag and Watkins (1993) describe an efficient modified QR algorithm that uses both Givens transformations and Gaussian elimination transformations, with or without pivoting. For the $n \times n$ Hessenberg matrix $A^{(0,0)}$, the first step of the Haag-Watkins procedure begins with a 3×3 Householder reflection matrix, $\tilde{G}_0$, whose first column is

$$(A^{(0,0)} - \sigma_1 I)(A^{(0,0)} - \sigma_2 I)e_1,$$

where σ_1 and σ_2 are the eigenvalues of the 2×2 matrix

$$\begin{bmatrix} a_{n-1,n-1} & a_{n-1,n} \\ a_{n-1,n} & a_{n,n} \end{bmatrix},$$

and e_1 is the first unit vector of length n. The $n \times n$ matrix G_0 is $\text{diag}(\tilde{G}_0, I)$. The initial transformation $G_0^{-1} A^{(0,0)} G_0$ creates a bulge with nonzero elements $a_{31}^{(0,1)}$, $a_{41}^{(0,1)}$, and $a_{42}^{(0,1)}$.

After the initial transformation, the Haag-Watkins procedure makes $n-3$ transformations

$$A^{(0,k+1)} = G_k^{-1} A^{(0,k)} G_k,$$

for $k = 1, 2, \ldots, n-3$, that chase the bulge diagonally down the matrix, so that $A^{(0,k+1)}$ differs from Hessenberg form only by the nonzero elements $a^{(0,k+1)}_{k+3,k+1}$, $a^{(0,k+1)}_{k+4,k+1}$, and $a^{(0,k+1)}_{k+4,k+2}$. To accomplish this, the matrix G_k differs from the identity only in rows and columns $k+1$, $k+2$, and $k+3$. The transformation

$$G_k^{-1} A^{(0,k)}$$

annihilates the entries $a^{(0,k)}_{k+2,k}$ and $a^{(0,k)}_{k+3,k}$, and the transformation

$$(G_k^{-1} A^{(0,k)}) G_k$$

produces $A^{(0,k+1)}$ with two new nonzero elements $a^{(0,k+1)}_{k+4,k+1}$ and $a^{(0,k+1)}_{k+4,k+2}$. The final transformation in the first step, for $k = n-2$, annihilates $a^{(0,k)}_{n,n-2}$. The transformation matrix G_{n-2} differs from the identity only in rows and columns $n-1$ and n. These steps are iterated until the matrix becomes triangular. As the subdiagonal elements converge to zero, the shifts for use in the first transformation of a step (corresponding to σ_1 and σ_2) are determined by 2×2 submatrices higher on the diagonal. Special consideration must be given to situations in which these submatrices contain zero elements. For this, the reader is referred to Watkins (1991) or Golub and Van Loan (1996).

This description has just indicated the general flavor of the QR method. There are different variations on the overall procedure, and then many computational details that must be observed. In the Haag-Watkins procedure, for example, the G_k's are not unique; and their form can affect the efficiency and the stability of the algorithm. Haag and Watkins (1993) describe criteria for the selection of the G_k's. They also discuss some of the details of programming the algorithm.

4.4 Singular Value Decomposition

The standard algorithm for computing the singular value decomposition

$$A = U \Sigma V^{\mathrm{T}}$$

is due to Golub and Reinsch (1970), and is built on ideas of Golub and Kahan (1965). The first step in the Golub-Reinsch algorithm for the singular value

decomposition of the $n \times m$ matrix A is to reduce A to upper bidiagonal form:

$$A^{(0)} = \begin{bmatrix} \mathtt{X} & \mathtt{X} & \mathtt{0} & \cdots & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{X} & \mathtt{X} & \cdots & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{0} & \mathtt{X} & \cdots & \mathtt{0} & \mathtt{0} \\ & & & \ddots & \ddots & \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \cdots & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \cdots & \mathtt{0} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \cdots & \mathtt{0} & \mathtt{0} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \cdots & \mathtt{0} & \mathtt{0} \end{bmatrix}.$$

We assume $n \geq m$. (If this is not the case, we merely use A^{T}.) This algorithm is basically a factored form of the QR algorithm for the eigenvalues of $A^{(0)\mathrm{T}}A^{(0)}$, which would be symmetric and tridiagonal.

The Golub-Reinsch method produces a sequence of upper bidiagonal matrices, $A^{(0)}, A^{(1)}, A^{(2)}, \ldots$, which converges to the diagonal matrix Σ. (Each of these has a zero submatrix below the square submatrix.) Similarly to the QR method for eigenvalues, the transformation from $A^{(j)}$ to $A^{(j+1)}$ is effected by a sequence of orthogonal transformations,

$$\begin{aligned} A^{(j+1)} &= R_{m-2}^{\mathrm{T}} R_{m-3}^{\mathrm{T}} \cdots R_0^{\mathrm{T}} A^{(j)} T_0 T_1 \cdots T_{m-2} \\ &= R^{\mathrm{T}} A^{(j)} T, \end{aligned}$$

which first introduces a nonzero entry below the diagonal (T_0 does this) and then chases it down the diagonal. After T_0 introduces a nonzero entry in the $(2,1)$ position, R_0^{T} annihilates it and produces a nonzero entry in the $(1,3)$ position; T_1 annihilates the $(1,3)$ entry and produces a nonzero entry in the $(3,2)$ position, which R_1^{T} annihilates, and so on. Each of the R_k's and T_k's are Givens transformations, and, except for T_0, it should be clear how to form them.

If none of the elements along the main diagonal or the diagonal above the main diagonal is zero, then T_0 is chosen as the Givens transformation such that T_0^{T} will annihilate the second element in the vector

$$(a_{11}^2 - \sigma_1, a_{11}a_{12}, 0, \cdots, 0),$$

where σ_1 is the eigenvalue of the lower right-hand 2×2 submatrix of $A^{(0)\mathrm{T}}A^{(0)}$ that is closest in value to the (m,m) element of $A^{(0)\mathrm{T}}A^{(0)}$. This is easy to compute (see Exercise 4.6).

If an element along the main diagonal or the diagonal above the main diagonal is zero, we must proceed slightly differently. (Remember that for purposes of computations, "zero" generally means "near zero", that is, to within some set tolerance.)

If an element above the main diagonal is zero, the bidiagonal matrix is separated at that value into a block diagonal matrix, and each block (which is bidiagonal) is treated separately.

If an element on the main diagonal, say a_{kk}, is zero, then a singular value is zero. In this case we apply a set of Givens transformations from the left. We first use G_1, which differs from the identity only in rows and columns k and $k+1$, to annihilate the $(k, k+1)$ entry and introduce a nonzero in the $(k, k+2)$ position. We then use G_2, which differs from the identity only in rows and columns k and $k+2$, to annihilate the $(k, k+2)$ entry and introduce a nonzero in the $(k, k+3)$ position. Continuing this process, we form a matrix of the form

$$\begin{bmatrix} \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{Y} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{0} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} \end{bmatrix}.$$

The Y in this matrix (in position $(k-1, k)$) is then chased up the upper block consisting of the first k rows and columns of the original matrix by using Givens transformations applied from the right. This then yields two block bidiagonal matrices (and a 1×1 0 matrix). We operate on the individual blocks as before.

After the steps have converged to yield a diagonal matrix, $\tilde{\Sigma}$, all of the Givens matrices applied from the left are accumulated into a single matrix, and all from the right are accumulated into a single matrix, to yield a decomposition

$$A = \tilde{U}\tilde{\Sigma}\tilde{V}^{\mathrm{T}}.$$

There is one last thing to do. The elements of $\tilde{\Sigma}$ may not be nonnegative. This is easily remedied by postmultiplying by a diagonal matrix D that is the same as the identity except for having a -1 in any position corresponding to a negative value in $\tilde{\Sigma}$. In addition, we generally form the singular value decomposition is such a way that the elements in Σ are nonincreasing. The entries in $\tilde{\Sigma}$ can be rearranged by a permutation matrix P so they are in nonincreasing order. So we have

$$\Sigma = P^{\mathrm{T}}\tilde{\Sigma}DP,$$

and the final decomposition is

$$\begin{aligned} A &= \tilde{U}PD\Sigma P^{\mathrm{T}}\tilde{V}^{\mathrm{T}} \\ &= U\Sigma V^{\mathrm{T}}. \end{aligned}$$

If $n \geq \frac{5}{3}m$, a modification of this algorithm by Chan (1982a, 1982b) is more efficient than the standard Golub-Reinsch method.

Exercises

4.1. Simple matrices and the power method.

(a) Let A be an $n \times n$ matrix whose elements are generated independently (but not necessarily identically) from continuous distributions. What is the probability that A is simple?

(b) Under the same conditions as in Exercise 4.1a, and with $n \geq 3$, what is the probability that $|\lambda_{n-2}| < |\lambda_{n-1}| < |\lambda_n|$, where λ_{n-2}, λ_{n-1}, and λ_n are the three eigenvalues with largest absolute values?

(c) Prove that the power method converges linearly if $|\lambda_{n-2}| < |\lambda_{n-1}| < |\lambda_n|$, $c_{n-1} \neq 0$, and $c_n \neq 0$. (The c's are the coefficients in the expansion of $x^{(0)}$.) *Hint:* Substitute the expansion in equation (4.2), page 125, into the expression for the convergence ratio in (4.3).

(d) Suppose A is simple, and the elements of $x^{(0)}$ are generated independently (but not necessarily identically) from continuous distributions. What is the probability that the power method will converge linearly?

4.2. Consider the matrix

$$\begin{bmatrix} 4 & 1 & 2 & 3 \\ 1 & 5 & 3 & 2 \\ 2 & 3 & 6 & 1 \\ 3 & 2 & 1 & 7 \end{bmatrix}.$$

(a) Use the power method to determine the largest eigenvalue of this matrix.

(b) Using Givens transformations, reduce the matrix to upper Hessenberg form.

4.3. In the matrix

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 5 & 2 & 0 \\ 3 & 2 & 6 & 1 \\ 0 & 0 & 1 & 8 \end{bmatrix}$$

determine the Givens transformations to chase the 3 in the $(3, 1)$ position out of the matrix.

4.4. In the matrix

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 3 & 5 & 2 & 0 \\ 0 & 0 & 6 & 1 \\ 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

determine the Givens transformations to chase the 3 in the $(2, 1)$ position out of the matrix.

4.5. In the QR methods for eigenvectors and singular values, why can we not just use additional orthogonal transformations to triangularize the given matrix (instead of just forming a similar Hessenberg matrix, as in Section 4.3), or to diagonalize the given matrix (instead of just forming the bidiagonal matrix, as in Section 4.4)?

4.6. Determine the eigenvalue σ_1 (on page 132) used in forming the matrix T_0 for initiating the chase in the algorithm for the singular value decomposition. Express it in terms of $a_{m,m}$, $a_{m-1,m-1}$, $a_{m-1,m}$, and $a_{m-1,m-2}$.

Chapter 5

Software for Numerical Linear Algebra

Because of the importance of linear algebraic computations, there is a wide range of software for these computations. The Guide to Available Mathematical Software (GAMS) (see the bibliography) is a good source of information about software.

For some types of software, it is important to be aware of the way the data are stored in the computer, as we discussed in Section 2.2, beginning on page 81. This may include such things as whether the storage is row-major or column-major, which will determine the stride, and may determine the details of an algorithm so as to enhance the efficiency. Software written in a language such as Fortran or C often requires the specification of the number of rows (in Fortran) or columns (in C) that have been allocated for the storage of a matrix. As we have indicated before, the amount of space allocated for the storage of a matrix may not correspond exactly to the size of the matrix.

There are many issues to consider in evaluating software or to be aware of when developing software. The portability of the software is an important consideration because a user's programs are often moved from one computing environment to another.

Some situations require special software that is more efficient than general-purpose software would be. Software for sparse matrices, for example, is specialized to take advantage of the zero entries. For sparse matrices it is necessary to have a scheme for identifying the locations of the nonzeros, and for specifying their values. The nature of storage schemes varies from one software package to another. The reader is referred to GAMS as a resource for information about software for sparse matrices.

Occasionally we need to operate on vectors or matrices whose elements are variables. Software for symbolic manipulation, such as Maple, can perform vector/matrix operations on variables. See Exercise 5.6, page 159.

5.1 Fortran and C

Fortran and C are the most commonly used procedural languages for scientific computation. Fortran has evolved over many years of usage by scientists and engineers. A common version of Fortran, called Fortran 77, was defined in its current form in 1978. A newer version, called Fortran 90 (see Kerrigan, 1993, Metcalf and Reid, 1990, or Press et al., 1996), provides additional facilities for working directly with arrays, for optional arguments in subprograms, for pointers, and for dynamic memory allocation. Fortran 90 did not replace Fortran 77; the latter is still a supported standard.

C began as a low-level language that provided many of the capabilities of a higher-level language together with more direct access to the operating system. It lacks some of the facilities that are very useful in scientific computation, such as complex data types, an exponentiation operator, and direct manipulation of arrays as vectors or matrices. An advantage of C, however, is that it provides for easier communication between program units, so it is often used when larger program systems are being put together.

Several libraries of program modules for numerical linear algebra are available both in Fortran and in C.

Indexing Arrays

Neither Fortran 77 nor C allows vectors and matrices to be treated as atomic units. Numerical operations on vectors and matrices are performed either within loops of operations on the individual elements or by invocation of a separate program module.

The natural way of representing vectors and matrices in Fortran and C is as array variables with indexes. Fortran handles arrays as multiply indexed memory locations, consistent with the nature of the object. Indexes start at 1, just as in the mathematical notation used throughout this book. The storage of two-dimensional arrays in Fortran is column-major, that is, the array A is stored as $\mathrm{vec}(A)$. To reference the contiguous memory locations, the first subscript varies fastest. In general-purpose software consisting of Fortran subprograms, it is often necessary to specify the lengths of all dimensions of a Fortran array except the last one.

An array in C is an ordered set of memory locations referenced by a pointer or by a name and an index. Indexes start at 0. The indexes are enclosed in rectangular brackets following the variable name. An element of a multidimensional array in C is indexed by multiple indexes, each within rectangular brackets. If the 3×4 matrix A is as stored in the C array `A`, the $(2,3)$ element, $A_{2,3}$ is referenced as `A[1][2]`.

Multidimensional arrays in C are arrays of arrays, in which the array constructors operate from right to left. This results in two-dimensional C arrays being stored in row-major order, that is, the array A is stored as $\mathrm{vec}(A^{\mathrm{T}})$. To reference the contiguous memory locations, the last subscript varies fastest.

In general-purpose software consisting of C functions, it is often necessary to specify the lengths of all dimensions of a C array except the first one.

Computational Efficiency

Two seemingly trivial things can have major effects on computational efficiency. One is movement of data from the computer's memory into the computational unit. How quickly this movement occurs depends, among other things, on the organization of the data in the computer. Multiple elements of an array can be retrieved from memory more quickly if they are in contiguous memory locations. (Location in computer memory does not necessarily refer to a physical place; in fact, memory is often divided into banks, and adjacent "locations" are in alternate banks. Memory is organized to optimize access.) The main reason that storage of data in contiguous memory locations affects efficiency involves the different levels of computer memory. A computer often has three levels of randomly accessible memory, ranging from "cache" memory, which is very fast, to "disk" memory, which is relatively slower. When data are used in computations they may be moved in blocks, or pages, from contiguous locations in one level of memory to a higher level. This allows faster subsequent access to other data in the same page. When one block of data is moved into the higher level of memory, another block is moved out. The movement of data (or program segments, which are also data) from one level of memory to another is called "paging".

In Fortran a column of a matrix occupies contiguous locations, so when paging occurs, elements in the same column are moved. Hence, a column of a matrix can often be operated on more quickly in Fortran than a row of a matrix. In C a row can be operated on more quickly.

Some computers have array processors that provide basic arithmetic operations for vectors. The processing units are called vector registers, and typically hold 128 or 256 full-precision floating-point numbers (see Section 1.1). For software to achieve high levels of efficiency, computations must be organized to match the length of the vector processors as often as possible. See Dongarra et al. (1991) for descriptions of methods for matrix computations on array processors.

Another thing that affects the performance of software is the execution of loops. In the simple loop

```
do i = 1, n
   sx(i) = sin(x(i))
end do
```

it may appear that the only computing is just the evaluation of the sine of the elements in the vector `x`. In fact, a nonnegligible amount of time may be spent in keeping track of the loop index and in accessing memory. A compiler on a vector computer may organize the computations so that they are done in groups corresponding to the length of the vector registers. On a computer

that does not have vector processors, a technique called "unrolling do-loops" is sometimes used. For the code segment above, unrolling the do-loop to a depth of 7, for example, would yield the following code:

```
do i = 1, n, 7
   sx(i)   = sin(x(i))
   sx(i+1) = sin(x(i+1))
   sx(i+2) = sin(x(i+2))
   sx(i+3) = sin(x(i+3))
   sx(i+4) = sin(x(i+4))
   sx(i+5) = sin(x(i+5))
   sx(i+6) = sin(x(i+6))
end do
```

plus a short loop for any additional elements in x beyond $7\lfloor n/7 \rfloor$. Obviously, this kind of programming effort is warranted only when n is large and when the code segment is expected to be executed many times. For widely distributed programs, such as the BLAS discussed in the next section, the extra programming is worthwhile.

5.1.1 BLAS

There are several basic computations for vectors and matrices that are very common across a wide range of scientific applications. Computing the dot product of two vectors, for example, is a task that may occur in such diverse areas as fitting a linear model to data or determining the maximum value of a function. The sets of routines called "basic linear algebra subprograms" (BLAS) implement many of the standard operations for vectors and matrices. The BLAS represent a very significant step toward software standardization, because the definitions of the tasks and the user interface are the same on all computing platforms. The actual coding, however, may be quite different, to take advantage of special features of the hardware or underlying software, such as compilers.

The level 1 BLAS or BLAS-1, the original set of the BLAS, are for vector operations. They were defined by Lawson et al. (1979). Matrix operations, such as multiplying two matrices were built using the BLAS-1. Later, a set of the BLAS, called level 2 or the BLAS-2, for operations involving a matrix and a vector, was defined by Dongarra et al. (1988), a set called the level 3 BLAS or the BLAS-3, for operations involving two dense matrices, was defined by Dongarra et al. (1990), and a set of the level 3 BLAS for sparse matrices was proposed by Duff et al. (1997).

The operations performed by the BLAS often cause an input variable to be updated. For example, in a Givens rotation, two input vectors are rotated into two new vectors. In this case, it is natural and efficient just to replace the input values with the output values (see below). A natural implementation of such an operation is to use an argument that is both input and output. In some

programming paradigms, such a "side effect" can be somewhat confusing, but the value of this implementation outweighs the undesirable properties.

There is a consistency of the interface among the BLAS routines. The nature of the arguments and their order in the reference are similar from one routine to the next. The general order of the arguments is:

1. the size or shape of the vector or matrix,
2. the array itself, which may be either input or output,
3. the stride, and
4. other input arguments.

The first and second types of arguments are repeated as necessary for each of the operand arrays and the resultant array.

A BLAS routine is identified by a root character string that indicates the operation, for example, `dot` or `axpy`. The name of the BLAS program module may depend on the programming language. In Fortran, the root may be prefixed by `s` to indicate single precision, by `d` to indicate double precision, or by `c` to indicate complex, for example. If the language allows generic function and subroutine references, just the root of the name is used.

The axpy operation we referred to on page 49 multiplies one vector by a constant and then adds another vector ($ax + y$). The BLAS routine `axpy` performs this operation. The interface is

```
axpy(n, a, x, incx, y, incy)
```

where

`n` – the number of elements in each vector

`a` – the scalar constant

`x` – the input/output one-dimensional array that contains the elements of the vector x

`incx` – the stride in the array `x` that defines the vector

`y` – the input/output one-dimensional array that contains the elements of the vector y

`incy` – the stride in the array `y` that defines the vector

Another example, the routine `rot` to apply a Givens rotation (similar to the routine `rotm` that we referred to in Section 3.2), has the interface:

```
rot(n, x, incx, y, incy, c, s)
```

where

`n` – the number of elements in each vector

`x` – the input/output one-dimensional array that contains the elements of the vector x

`incx` – the stride in the array `x` that defines the vector

`y` – the input/output one-dimensional array that contains the elements of the vector y

`incy` – the stride in the array `y` that defines the vector

`c` – the cosine of the rotation

`s` – the sine of the rotation

This routine is invoked after `rotg` has been called to determine the cosine and the sine of the rotation. (See Exercise 5.3, page 158.)

5.1.2 Fortran and C Libraries

When work was being done on the BLAS-1 in the 1970s, those lower-level routines were being incorporated into a higher-level set of Fortran routines for matrix eigensystem analysis, called EISPACK (Smith et al., 1976), and into a higher-level set of Fortran routines for solutions of linear systems, called LINPACK (Dongarra et al., 1979). As work progressed on the BLAS-2 and BLAS-3 in the 1980s and later, a unified set of Fortran routines for both eigenvalue problems and solutions of linear systems was developed, called LAPACK (Anderson et al., 1995).

Another standard set of routines, called the BLACS (Basic Linear Algebra Communication Subroutines), provides a portable message-passing interface primarily for linear algebra computations with a user interface similar to that of the BLAS. A slightly higher-level set of routines, the PBLAS, combine both the data communication and computation into one routine, also with a user interface similar to that of the BLAS. A distributed memory version of LAPACK, called ScaLAPACK, has been built on the BLACS and the PBLAS modules.

Standards for message passing in a distributed-memory parallel processing environment are evolving. The MPI (message passing interface) standard is being developed, primarily at Argonne National Labs. This allows for standardized message passing across languages and systems. IBM has built the Message Passing Library (MPL) in both Fortran and C that provides message passing kernels. PLAPACK is a package for linear algebra built on MPI. (See Van de Geijn, 1997.)

All of these packages are available on a range of platforms, especially on high-performance computers.

Two of the most widely used Fortran and C libraries are the IMSL Libraries and the NAG Library. They provide a large number of routines for numerical linear algebra, ranging from very basic computations as provided in the BLAS through complete routines for solving various types of systems of equations and for performing eigenanalysis. Both libraries are available in both Fortran and C versions.

Matrix Storage Modes

Matrices that have multiple elements with the same value can often be stored in the computer in such a way that the individual elements do not all have separate locations. Symmetric matrices and matrices with many zeros, such as the upper or lower triangular matrices of the various factorizations we have discussed, are examples of matrices that do not require full rectangular arrays for their storage.

A special storage mode for symmetric matrices uses a linear array to store only the unique elements. The symmetric matrix A is stored as $\text{vech}(A)$. For example, the symmetric matrix

$$\begin{bmatrix} 1 & 2 & 4 & \cdots \\ 2 & 3 & 5 & \cdots \\ 4 & 5 & 6 & \cdots \\ & \cdots & & \end{bmatrix}$$

is represented by the array

$$(1, 2, 3, 4, 5, 6, \cdots).$$

For an $n \times n$ symmetric matrix A, the correspondence with the $n(n+1)/2$-vector v is $v_{i(i-1)/2+j} = a_{i,j}$, for $i \geq j$. Notice that the relationship does not involve n. For $i \geq j$, in Fortran, it is

```
v(i*(i-1)/2+j) = a(i,j)
```

and in C it is

```
v[i*(i+1)/2+j] = a[i][j]
```

Although the amount of space saved by not storing the full symmetric matrix is only about one half of the amount of space required, the use of rank 1 arrays rather than rank 2 arrays can yield some reference efficiencies. (Recall that in discussions of computer software objects, "rank" usually means the number of dimensions.) For band matrices and for other sparse matrices, the savings in storage can be much larger.

The BLAS and the IMSL Libraries implement a wide range of matrix storage modes:

Symmetric mode. A full matrix is used for storage, but only the upper or lower triangular portion of the matrix is used. Some library routines allow the user to specify which portion is to be used, and others require that it be the upper portion.

Hermitian mode. This is the same as the symmetric mode, except for the obvious changes for the Hermitian transpose.

Triangular mode. This is the same as the symmetric mode (with the obvious changes in the meanings).

Band mode. For the $n \times m$ band matrix A with lower band width w_l and upper band width w_u, an $w_l + w_u \times m$ array is used to store the elements. The elements are stored in the same column of the array, say `aa`, as they are in the matrix; that is,

$$\texttt{aa}(i - j + w_u + 1, j) = a_{i,j},$$

for $i = 1, 2, \ldots, w_l + w_u + 1$.
Band symmetric, band Hermitian, and **band triangular** modes are all defined similarly. In each case, only the upper or lower bands are referenced.

Sparse storage mode. There are several different schemes for representing sparse matrices. The IMSL Libraries use three arrays, each of rank 1 and with length equal to the number of nonzero elements. The integer array `i` contains the row indicator, the integer array `j` contains the column indicator, and the floating-point array `a` contains the corresponding values; that is, the (`i`(k), `j`(k)) element of the matrix is stored in `a`(k). The level 3 BLAS for sparse matrices proposed by Duff et al. (1997) have an argument to allow the user to specify the type of storage mode.

Examples of Use of the IMSL Libraries

Consider the problem of solving the system of linear equations:

$$\begin{aligned} x_1 + 4x_2 + 7x_3 &= 10 \\ 2x_1 + 5x_2 + 8x_3 &= 11 \\ 3x_1 + 6x_2 + 9x_3 &= 12. \end{aligned}$$

Write the system as $Ax = b$. The coefficient matrix A is real (not necessarily `REAL`) and square. We can use various IMSL subroutines to solve this problem. The two simplest basic routines are `LSLRG/DLSLRG` and `LSARG/DLSARG`. Both have the same set of arguments:

`N`, the problem size,

`A`, the coefficient matrix,

LDA, the leading dimension of A (A can be defined to be bigger than it actually is in the given problem),

B, the right-hand sides,

IPATH, an indicator of whether $Ax = b$ or $A^{\mathrm{T}}x = b$ is to be solved, and

X, the solution.

The difference in the two routines is whether or not they do iterative refinement. A program to solve the system without iterative refinement is shown in Figure 5.1.

```
C  Fortran 77 program
      parameter (ida=3)
      integer   n, ipath
      real      a(ida, ida), b(ida), x(ida)
C  Storage is by column;
C  nonblank character in column 6 indicates continuation
      data      a/1.0,   2.0,  3.0,
     +            4.0,   5.0,  6.0,
     +            7.0,   8.0,  9.0/
      data      b/10.0, 11.0, 12.0/
      n     = 3
      ipath = 1
      call lsarg (n, a, lda, b, ipath, x)
      print *, 'The solution is', x
      end
```

Figure 5.1: IMSL Fortran Program to Solve the System of Linear Equations

The IMSL C function to solve this problem is lin_sol_gen, which is available as *float* *imsl_f_lin_sol_gen or *double* *imsl_d_lin_sol_gen. The only required arguments for *imsl_f_lin_sol_gen are:

int n, the problem size,

float a[], the coefficient matrix,

float b[], the right-hand sides.

Either function will allow the array a to be larger than n, in which case the number of columns in a must be supplied in an optional argument. Other optional arguments allow the specification of whether $Ax = b$ or $A^{\mathrm{T}}x = b$ is to be solved (corresponding to the argument IPATH in the Fortran subroutines LSLRG/DLSLRG and LSARG/DLSARG), the storage of the LU factorization, the storage of the inverse, and so on. A program to solve the system is shown in Figure 5.2. Note the difference in the column orientation of Fortran and the row orientation of C.

The argument IMSL_A_COL_DIM is optional, taking the value of n, the number of equations, if it is not specified. It is used in Figure 5.2 only for illustration.

```
\*  C program *\
#include <imsl.h>
#include <stdio.h>
main()
{
    int   n = 3;
    float *x;
\* Storage is by row;
   statements are delimited by ';', so statements continue automatically. */
    float a[] = {1.0,  4.0,  7.0,
                 2.0,  5.0,  8.0,
                 3.0,  6.0,  9.0};
    float b[] = {10.0, 11.0, 12.0};
    x = imsl_f_lin_sol_gen (n, a, IMSL_A_COL_DIM, 3, b, 0);
    printf ("The solution is %10.4f%10.4f%10.4f\n",
                             x[0], x[1], x[2]);
}
```

Figure 5.2: IMSL C Program to Solve the System of Linear Equations

5.1.3 Fortran 90 and 95

One of the most useful of the features of Fortran 90 for the scientific programmer is the provision of primitive constructs for vectors and matrices. Whereas all of the Fortran 77 intrinsics are scalar-valued functions, Fortran 90 provides matrix-valued functions. For example, if `aa` and `bb` represent matrices conformable for multiplication, the statement

```
cc = matmul(aa, bb)
```

yields the Cayley product in `cc`. The `matmul` function also allows multiplication of vectors and matrices.

Indexing of arrays starts at 1 by default (any starting value can be specified, however), and storage is column-major.

Space must be allocated for arrays in Fortran 90, but this can be done at run time. An array can be initialized either in the statement allocating the space or in a regular assignment statement. A vector can be initialized by listing the elements between "(/" and "/)". This list can be generated in various ways. The `reshape` function can be used to initialize matrices.

For example, a Fortran 90 statement to declare that the variable `aa` is to be used as a 3×4 array, and to allocate the necessary space is

```
real, dimension(3,4) :: aa
```

A Fortran 90 statement to initialize `aa` with the matrix

$$\begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

is

```
aa = reshape( (/ 1., 2., 3., &
                 4., 5., 6., &
                 7., 8., 9., &
                10.,11.,12./), &
               (/3,4/) )
```

Fortran 90 has an intuitive syntax for referencing subarrays, shown in Table 5.1.

`aa(2:3,1:3)`	the 2×3 submatrix in rows 2 and 3 and columns 1 to 3 of `aa`
`aa(:,1:4:2)`	refers to the submatrix with all 3 rows and the 1^{st} and 3^{rd} columns of `aa`
`aa(:,4)`	refers to the column vector that is the 4^{th} column of `aa`

Table 5.1: Subarrays in Fortran 90

Notice that because the indexing starts with 1 (instead of with 0) the correspondence between the computer objects and the mathematical objects is a natural one. The subarrays can be used directly in functions. For example, if `bb` is the matrix

$$\begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}$$

the Fortran 90 function reference

```
matmul(aa(1:2,2:3), bb(3:4,:))
```

yields the Cayley product

$$\begin{bmatrix} 4 & 7 \\ 5 & 8 \end{bmatrix} \begin{bmatrix} 3 & 7 \\ 4 & 8 \end{bmatrix}. \tag{5.1}$$

Libraries built on Fortran 90 allow some of the basic operations of linear algebra to be implemented as operators whose operands are vectors or matrices.

Fortran 95 is a revision of and replacement for Fortran 90. It is a relatively minor revision, with primary emphasis on clarifications, corrections, and interpretations of Fortran 90. It also contains some of the constructs, such as `forall`, that have evolved to support parallel processing.

Current plans are to produce a more extensive revision called Fortran 2000 that will include such features as exception handling, interoperability with C, allocatable components, parameterized derived types, and object-oriented programming.

5.2 Interactive Systems for Array Manipulation

Many of the computations for linear algebra are implemented as simple operators on arrays in some interactive systems. Some of the more common interactive systems that provide for direct array manipulation are Matlab, S-Plus, SAS IML, APL, Lisp-Stat, Gauss, IDL, and PV-Wave. There is no need to allocate space for the arrays in these systems, as there is for arrays in Fortran and C.

In the next two sections we briefly describe the facilities for linear algebra in Matlab and S-Plus. The purpose is to give a very quick comparative introduction.

5.2.1 Matlab

Matlab is an interactive, interpretive, expression language that works with only one type of object: a rectangular array of numbers (possibly complex). The array is a matrix. Scalars (even indices) are 1×1 matrices. The indexing of arrays starts with 1.

If an assignment statement in Matlab is not terminated with a semicolon, the matrix on the left-hand side of the assignment is printed. If a statement consists only of the name of a matrix, the object is printed.

A comment statement in Matlab begins with a percent sign, "%".

In Matlab a matrix is initialized by listing the elements row-wise within brackets and with semicolons marking the end of rows. (Matlab also has a `reshape` function similar to that of Fortran 90 that treats the matrix in a column-major fashion.)

In general, the operators in Matlab refer to the common vector/matrix operations. For example, Cayley multiplication is indicated by the usual multiplication symbol, "*". The meaning of an operator can often be changed to become the corresponding element-by-element operation by preceding the operator with a period; for example, the symbol ".*" indicates the Hadamard product of two matrices. The expression

```
aa * bb
```

indicates the Cayley product of the matrices, where the number of columns of `aa` must be the same as the number of rows of `bb`; and the expression

```
aa .* bb
```

indicates the Hadamard product of the matrices, where the numbers of rows and columns of `aa` must be the same as the numbers of rows and columns of `bb`. The transpose of a vector or matrix is obtained by use of a postfix operator "T", which is the same ASCII character as the apostrophe:

```
aa'
```

Figure 5.3 below shows Matlab code that initializes the same matrix `aa` that we used as an example for Fortran 90 above. The code in Figure 5.3 also initializes a vector `xx` and a 4×2 matrix `bb`, and then forms and prints some products.

```
%  Matlab program fragment
xx = [1 2 3 4];
%  Storage is by rows; continuation is indicated by '...'
aa = [1 4 7 10; ...
      2 5 8 11; ...
      3 6 9 12];
bb = [1 5; 2 6; 3 7; 4 8];
%  Printing occurs automatically unless ';' is used to suppress it.
yy = a*xx'
yy = xx(1:3)*aa
cc = aa*bb
```

Figure 5.3: Matlab Code to Define and Initialize Two Matrices and a Vector, and Then Form and Print Their Product

Matlab distinguishes between row vectors and column vectors. A row vector is a matrix whose first dimension is 1, and a column vector is a matrix whose second dimension is 1. In either case, an element of the vector is referenced by a single index.

Subarrays in Matlab are defined in much the same way as in Fortran 90, except for one major difference: the upper limit and the stride are reversed in the triplet used in identifying the row or column indices. Examples of subarray references in Matlab are shown in Table 5.2. Compare these with the Fortran 90 references shown in Table 5.1.

`aa(2:3,1:3)`	the 2×3 submatrix in rows 2 and 3 and columns 1 to 3 of `aa`
`aa(:,1:2:4)`	the submatrix with all 3 rows and the 1^{st} and 3^{rd} columns of `aa`
`aa(:,4)`	the column vector that is the 4^{th} column of `aa`

Table 5.2: Subarrays in Matlab

The subarrays can be used directly in expressions. For example, the expression

```
aa(1:2,2:3) * bb(3:4,:)
```

yields the product

$$\begin{bmatrix} 4 & 7 \\ 5 & 8 \end{bmatrix} \begin{bmatrix} 3 & 7 \\ 4 & 8 \end{bmatrix}$$

as on page 147.

Matlab has functions for many of the basic operations we have discussed, some of which are shown in Table 5.3

`norm`	Matrix or vector norm.
	For vectors all L_p norms are available.
	For matrices the L_1, L_2, L_∞, and Frobenius norms are available.
`rank`	Number of linearly independent rows or columns.
`det`	Determinant.
`trace`	Trace.
`cond`	Matrix condition number.
`null`	Null space.
`orth`	Orthogonalization.
`inv`	Matrix inverse.
`pinv`	Pseudoinverse.
`lu`	LU decomposition.
`qr`	QR decomposition.
`chol`	Cholesky factorization.
`svd`	Singular value decomposition.
`linsolve`	Solve system of linear equations.
`lscov`	Weighted least squares.
	The operator "\" can be used for ordinary least squares.
`nnls`	Nonnegative least-squares.
`eig`	Eigenvalues and eigenvectors.
`poly`	Characteristic polynomial.
`hess`	Hessenberg form.
`schur`	Schur decomposition.
`balance`	Diagonal scaling to improve eigenvalue accuracy.
`expm`	Matrix exponential.
`logm`	Matrix logarithm.
`sqrtm`	Matrix square root. (More general than `chol`.)
`funm`	Evaluate general matrix function.

Table 5.3: Some Matlab Functions for Vector/Matrix Computations

In addition to these functions, Matlab has special operators "\" and "/" for solving linear systems or for multiplying one matrix by the inverse of another.

Matlab is a proprietary package distributed by The Mathworks, Inc. There is a freely available package, called Octave, that provides the same functionality in the same language. There are a number of books on Matlab, including, for

example, Hanselman and Littlefield (1996) and Etter (1996). The book by Coleman and Van Loan (1988) is not specifically on Matlab, but shows how to perform matrix computations in Matlab.

5.2.2 S, S-Plus

The software system called S was developed at Bell Laboratories in the mid-1970s. Work on S has continued at Bell Labs and the system has evolved considerably since the early versions. S-Plus is an enhancement of S, developed by StatSci, Inc. (now a part of MathSoft, Inc.). The enhancements include graphical interfaces, more statistical analysis functionality, and support. S and S-Plus are both data analysis systems and object-oriented programming languages. In the following, rather than continuing to refer to both S and S-Plus, we will refer only to S-Plus, but most of the discussion applies to both systems.

The most important S-Plus entity is the function. In S-Plus all actions are "functions", and S-Plus has an extensive set of functions, that is, verbs. Assignment is made by "`<-`" or by "`_`". (The symbol "`_`" should *never* be used for assignment, in my opinion. It is not mnemonic, and it is often used as a connective. I have seen students use a variable `L_p`, with "`_`" being used as a connective, and then use a statement like `norm_L_p`, in which the first "`_`" is an assignment. Use of this symbol instead of `<-` saves exactly one unshifted keystroke!)

A comment statement in S-Plus begins with a pound sign, "`#`".

S-Plus has a natural syntax and powerful functions for dealing with vectors and matrices, which are objects in the base language. S-Plus has functions for printing, but if a statement consists of just the name of an object, the object is printed.

Indexing of arrays starts at 1, and storage is column-major. Indexes are indicated by "`[ ]`"; for example, `aa[1]` refers to the first element of the one-dimensional array `aa`.

A list is constructed by the `c` function. A list can be treated as a vector without modification. A matrix is constructed from a list by the `matrix` function. Cayley multiplication is indicated by the symbol, "`%*%`". Most operators with array operands are applied elementwise; for example, the symbol "`*`" indicates the Hadamard product of two matrices. The expression

```
aa %*% bb
```

indicates the Cayley product of the matrices, where the number of columns of `aa` must be the same as the number of rows of `bb`; and the expression

```
aa * bb
```

indicates the Hadamard product of the matrices, where the numbers of rows and columns of `aa` must be the same as the numbers of rows and columns of `bb`. The transpose of a vector or matrix is obtained by use of the function "`t`":

```
t(aa)
```

An important comment about S-Plus usage is in order here. The two functions `c` and `t` essentially preempt these two letters as names of variables. Although it is possible to redefine functions, it is probably better just to remember never to name a variable "c" or "t."

Figure 5.4 below shows S-Plus code that does the same thing as the Matlab code in Figure 5.3, that is, initialize two matrices and a vector, and then form and print their products.

```
#   S-Plus program fragment
xx <- c(1 2 3 4) #
#   Storage is by column, but a matrix can be constructed by rows;
#   the form of a statement indicates when it is complete, so
#   statements continue automatically.
aa <- matrix(c( 1, 4, 7, 10,
                2, 5, 8, 11,
                3, 6, 9, 12),
             nrow=3, byrow=T)
bb <- matrix(seq(1,8), nrow=4)
yy <- aa %*% xx
#   Printing is performed by entering the name of the object
yy
yy <- xx[c(1,2,3)] %*% aa
yy
cc <- aa %*% bb
cc
```

Figure 5.4: S-Plus Code to Define and Initialize Two Matrices and a Vector, and Then Form and Print Their Product

S-Plus considers vectors to be column vectors; hence, if `xx` is a vector,

```
t(xx) %*% xx
```

is the dot product of `xx` with itself, and

```
xx %*% t(xx)
```

is a matrix. In expressions involving matrix/vector multiplication, S-Plus does not distinguish between row vectors and column vectors. In the expressions

```
yy <- aa %*% xx
```

and

```
yy <- xx[c(1,2,3)] %*% aa
```

in Figure 5.4, the vector is interpreted as a row or column as appropriate for the multiplication to be defined. Compare the similar expressions in the Matlab code in Figure 5.3.

Subarrays in S-Plus are defined in a similar way as in Fortran 90. A missing index indicates that the entire corresponding dimension is to be used. Groups of indices can be formed by the `c` function or the `seq` function, which is similar to the `i:j:k` notation of Fortran 90.

`aa[c(2,3),c(1,3)]`	the 2×3 submatrix in rows 2 and 3 and columns 1 to 3 of `aa`
`aa[,seq(1,4,2)]`	the submatrix with all 3 rows and the 1^{st} and 3^{rd} columns of `aa`
`aa[,4]`	the column vector that is the 4^{th} column of `aa`

Table 5.4: Subarrays in S-Plus

The subarrays can be used directly in expressions. For example, the expression

```
aa[c(1,2),c(2,3)] %*% bb[c(3,4),]
```

yields the product

$$\begin{bmatrix} 4 & 7 \\ 5 & 8 \end{bmatrix} \begin{bmatrix} 3 & 7 \\ 4 & 8 \end{bmatrix}$$

as on page 147.

S-Plus has functions for many of the basic operations we have discussed, some of which are shown in Table 5.5

S and S-Plus are proprietary systems, but they have a very broad community of users and supporters. There are independently developed guides to the software, such as Everitt (1994), Spector (1994), and Krause and Olson (1997). There are also texts that address the statistical methods in the context of the S or S-Plus system, such as Chambers and Hastie (1992) (John Chambers was the principal designer of S), Härdle (1991), Venables and Ripley (1997), and Bruce and Gao (1996), who describe wavelet analysis in S-Plus.

There is a freely available package, called R, that provides the same functionality in the same language as S-Plus. This is similar to Octave and Matlab.

5.3 High-Performance Software

Because computations for linear algebra are so pervasive in scientific applications, it is important to have very efficient software for carrying out these

`norm`	Matrix norm. The L_1, L_2, L_∞, and Frobenius norms are available.
`vecnorm`	Vector L_p norm.
`det`	Determinant.
`rcond.Matrix`	Matrix condition number.
`solve.Matrix`	Matrix inverse or seudoinverse.
`lu`	LU decomposition.
`qr`	QR decomposition.
`chol`	Cholesky factorization.
`svd`	Singular value decomposition.
`solve.Matrix`	Solve system of linear equations.
`lsfit`	Ordinary or weighted least squares.
`nnls.fit`	Nonnegative least-squares.
`eigen`	Eigenvalues and eigenvectors.

Table 5.5: Some S-Plus Functions for Vector/Matrix Computations

computations. Surveys of specialized software for vector architectures and parallel processors are available in Dongarra et al. (1991); Dongarra and Walker (1995); Gallivan et al. (1990); and Gallivan, Plemmons, and Sameh (1990). Pissanetzky (1984) discusses special software for sparse matrices. Dodson, Grimes, and Lewis (1991a, 1991b) provide a set of Fortran BLAS for sparse matrices. Freund, Golub, and Nachtigal (1992) describe algorithms for iterative matrix computations and identify some available software that implements them.

Quinn (1994, Chapters 7 and 9) describes various algorithms for basic vector/matrix operations, including solution of linear systems, that perform parallel processing.

The constructs of Fortran 90 are helpful in thinking of operations in such a way that they are naturally parallelized. While addition of arrays in Fortran 77 or C is an operation that leads to loops of sequential scalar operations, in Fortran 90 it is thought of as a single higher-level operation. How to perform operations in parallel efficiently is still not a natural activity, however. For example, the two Fortran 90 statements to add the arrays `aa` and `bb` and then to add `aa` and `cc`:

```
dd = aa + bb
ee = aa + cc
```

may be less efficient than loops because the array `aa` may be accessed twice.

The software package PVM, or Parallel Virtual Machine, which was developed at Oak Ridge National Lab, University of Tennessee, and Emory University, provides a set of C functions or Fortran subroutines that allow a heterogeneous collection of Unix computers to operate smoothly as a multicomputer. See Geist et al. (1994).

5.4 Test Data

Testbeds for software consist of test datasets that vary in condition, but that have known solutions or for which there is an easy way of verifying the solution. For testing software for matrix computations, a very common matrix is the *Hilbert matrix*, which has elements

$$h_{ij} = \frac{1}{i+j-1}.$$

Hilbert matrices have large condition numbers; for example, the 10×10 Hilbert matrix has condition number of order 10^{13}. The Matlab function `hilb(n)` generates an $n \times n$ Hilbert matrix.

Ericksen (1985) describes how to generate matrices with known inverses in such a way that the condition numbers vary widely. To generate an $n \times n$ matrix A, choose $x_1, x_2, \ldots, x_n$ arbitrarily, except such that $x_1 \neq 0$, and take

$$\begin{aligned} a_{1j} &= x_1 && \text{for } j = 1, \ldots, n, \\ a_{i1} &= x_i && \text{for } i = 2, \ldots, n, \\ a_{ij} &= a_{i,j-1} + a_{i-1,j-1} && \text{for } i, j = 2, \ldots, n. \end{aligned}$$

To represent the elements of the inverse, first define $y_1 = x_1^{-1}$, and for $i = 2, \ldots, n$,

$$y_i = -y_1 \sum_{k=0}^{i-1} x_{i-k} y_k.$$

Then the elements of the inverse of A, $B = (b_{ij})$, are given by

$$b_{in} = (-1)^{i+k} \binom{n-1}{i-1} y_1 \qquad \text{for } i = 1, \ldots, n,$$

$$b_{nj} = y_{n+1-j} \qquad \text{for } j = 1, \ldots, n-1,$$

$$b_{ij} = x_1 b_{in} b_{nj} + \textstyle\sum_{k=i+1}^{n} b_{k,j+1} \qquad \text{for } i, j = 1, \ldots, n-1,$$

where the binomial coefficient, $\binom{k}{m}$, is defined to be 0 if $k < m$ or $m < 0$.

The nonzero elements of L and U in the LU decomposition of A are easily seen to be $l_{ij} = x_{i+1-j}$ and $u_{ij} = \binom{j-1}{i-1}$. The nonzero elements of the inverses of L and U are then seen to have (i,j) elements y_{i+1-j} and $(-1)^{i-j}\binom{j-1}{i-1}$. The determinant of A is x_1^n. For some choices of $x_1, \ldots, x_n$, it is easy to determine the condition numbers, especially with respect to the L_1 norm, of the matrices A generated in this way. Ericksen (1985) suggests that the x's be chosen as

$$x_1 = 2^m \quad \text{for } m \leq 0$$

and

$$x_i = \begin{pmatrix} k \\ i-1 \end{pmatrix}, \quad \text{for } i = 2, \ldots, n \quad \text{and } k \geq 2,$$

in which case the L_1 condition number of 10×10 matrices will range from about 10^7 to 10^{17} as n ranges from 2 to 20 for $m = 0$, and will range from about 10^{11} to 10^{23} as n ranges from 2 to 20 for $m = -1$.

For testing algorithms for computing eigenvalues, a useful matrix is a *Wilkinson matrix*, which is a symmetric, tridiagonal matrix with 1's on the off-diagonals. For an $n \times n$ Wilkinson matrix, the diagonal elements are

$$\frac{n-1}{2}, \frac{n-5}{2}, \frac{n-5}{2}, \ldots, \frac{n-5}{2}, \frac{n-3}{2}, \frac{n-1}{2}.$$

If n is odd, the diagonal includes 0, otherwise all of the diagonal elements are positive. The 5×5 Wilkinson matrix, for example, is

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{bmatrix}.$$

The two largest eigenvalues of a Wilkinson matrix are very nearly equal. Other pairs are likewise almost equal to each other: the third and fourth largest eigenvalues are also close in size, the fifth and sixth largest, and so on. The largest pair is closest in size, and each smaller pair is less close in size.

The Matlab function `wilkinson(n)` generates an $n \times n$ Wilkinson matrix. Another test matrix available in Matlab is the Rosser test matrix, which is an 8×8 matrix with an eigenvalue of multiplicity two and three nearly equal eigenvalues. It is constructed by the Matlab function `rosser`.

A well-known, large, and wide-ranging set of test matrices for computational algorithms for various problems in linear algebra was compiled and described by Gregory and Karney (1969). Higham (1991) describes a set of test matrices and provides Matlab programs to generate the matrices.

Another set of test matrices is available through the "Matrix Market", designed and developed by R. Boisvert, R. Pozo, and K. Remington of the National Institute of Standards and Technology, with contributions by various other people. The test matrices are accessed over the World Wide Web at

`http://math.nist.gov/MatrixMarket`

The database can be searched by specifying characteristics of the test matrix, such as size, symmetry and so on. Once a particular matrix is found, its sparsity pattern can be viewed at various levels of detail, and other pertinent data can be reviewed. If the matrix seems to be what the user wants, it can be downloaded.

The initial database for the Matrix Market is the approximately 300 problems from the Harwell-Boeing Sparse Matrix Collection.

Assessing the Accuracy of a Computed Result

In real-life applications, the correct solution is not known. (This is also often the case for randomly generated test datasets.) We would like to have some way of assessing the accuracy using the data themselves.

Sometimes a convenient way of assessing the accuracy of the computations in a given problem is to perform internal consistency tests. An internal consistency test may be an assessment of the agreement of various parts of the output. Relationships among the output are exploited to insure that the individually computed quantities satisfy these relationships. Other internal consistency tests may be performed by comparing the results of the solutions of two problems with a known relationship.

The solution to the linear system $Ax = b$ has a simple relationship to the solution to the linear system $Ax = b + ca_j$, where a_j is the j^{th} column of A and c is a constant. A useful check on accuracy of a computed solution to $Ax = b$ is to compare it with a computed solution to the modified system. Of course, if the expected relationship does not hold, we do not know which solution is incorrect, but it is probably not a good idea to trust either. Mullet and Murray (1971) describe this kind of consistency test for regression software. To test the accuracy of the computed regression coefficients for regressing y on $x_1, \ldots, x_m$, they suggest comparing them to the computed regression coefficients for regressing $y+dx_j$ on $x_1, \ldots, x_m$. If the expected relationships do not obtain, the analyst has strong reason to doubt the accuracy of the computations.

Another simple modification of the problem of solving a linear system with a known exact effect is the permutation of the rows or columns.

Another simple internal consistency test that is applicable to many problems is to use two levels of precision in the computations. In using this test, one must be careful to make sure that the input data are the same. Rounding of the input data may cause incorrect output to result, but that is not the fault of the computational algorithm.

Internal consistency tests cannot confirm that the results are correct; they can only give an indication that the results are incorrect.

Exercises

5.1. Write a recursive function in Fortran, C, Matlab, S-Plus, or PV-Wave to multiply two square matrices using the Strassen algorithm (page 83). Write the function so that it uses an ordinary multiplication method if the size of the matrices is below a threshold that is supplied by the user.

5.2. There are various ways to evaluate the efficiency of a program: counting operations, checking the "wall time", using a shell level timer, and using a call within the program. In C the timing routine is `ctime`, and in Fortran 90 it is the subroutine `system_clock`. Fortran 77 does not have a built-in timing routine, but the IMSL Fortran Library provides one. For

this assignment you are to write six short C programs and six short Fortran programs. The programs in all cases are to initialize an $n \times m$ matrix so that the entries are equal to the column numbers, that is, all elements in the first column are 1's, all in the second column are 2's, etc. The six programs arise from three matrices of different sizes: 10000×10000, 100×1000000, and 1000000×100; and from two different ways of nesting the loops: for each size matrix, first nest the row loop within the column loop and then reverse the loops. The number of operations is the same for all programs. For each program, use both a shell level timer (e.g., in Unix, use `time`) and a timer called from within your program. Make a table of the times:

		10000 × 10000	100 × 1000000	1000000 × 100
Fortran	column-in-row	—	—	—
	row-in-column	—	—	—
C	column-in-row	—	—	—
	row-in-column	—	—	—

5.3. Obtain the BLAS routines `rotg` and `rot` for constructing and applying a Givens rotation. These routines exists in both Fortran and C; they are available in the IMSL Libraries or from *CALGO* (*Collected Algorithms of the ACM*; see the bibliography).

(a) Using these two routines, apply a Givens rotation to the matrix used in Exercise 3.8, in Chapter 3,

$$A = \begin{bmatrix} 3 & 5 & 6 \\ 6 & 1 & 2 \\ 8 & 6 & 7 \\ 2 & 3 & 1 \end{bmatrix},$$

so that the second column becomes $(5, \tilde{a}_{22}, 6, 0)$.

(b) Write a routine in Fortran or C that accepts as input a matrix and its dimensions, and uses the BLAS routines `rotg` and `rot` to produce its QR decomposition. There are several design issues you should address: how the output is returned (for purposes of this exercise, just return two arrays or pointers to the arrays in full storage mode); how to handle non-full rank matrices (for this exercise, assume that the matrix is full rank, so return an error message in this case); how to handle other input errors (what do you do if the user inputs a negative number for a dimension?); etc.

5.4. Using the BLAS routines `rotg` and `rot` for constructing and applying a Givens rotation and the program you wrote in Exercise 5.3, write a Fortran or C routine that accepts a simple symmetric matrix and computes its eigenvalues using the mobile Jacobi scheme The outer loop of your routine consists of the steps shown on page 129, and the multiple actions

of each of those steps can be implemented in a loop in serial mode. The importance of this algorithm, however, is realized when the actions in the individual steps on page 129 are performed in parallel.

5.5. Compute the two largest eigenvalues of the 21×21 Wilkinson matrix to 15 digits.

5.6. Use a symbolic manipulation software package such as Maple to determine the inverse of the matrix:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

Determine conditions for which the matrix would be singular. (You can use the `solve()` function in Maple on certain expressions in the symbolic solution you obtained.)

5.7. Consider the 3×3 symmetric Toeplitz matrix with elements a, b, and c, that is, the matrix that looks like this:

$$\begin{bmatrix} a & b & c \\ b & a & b \\ c & b & a \end{bmatrix}.$$

Invert this matrix.

Determine conditions for which the matrix would be singular.

Chapter 6

Applications in Statistics

One of the most common structures for statistical datasets is a two-dimensional array. A matrix is often a convenient object for representing numeric data structured this way; the variables on the dataset generally correspond to the columns, and the observations correspond to the rows. If the data are in the matrix X, a useful statistic is the sums of squares and cross-products matrix, $X^{\mathrm{T}}X$, or the "adjusted" squares and cross-products matrix, $X_a^{\mathrm{T}}X_a$, where X_a is the matrix formed by subtracting from each element of X the mean of the column containing that element. The matrix

$$\frac{1}{n-1}X_a^{\mathrm{T}}X_a,$$

where n is the number of observations (the number of rows in X), is the sample variance-covariance matrix. This matrix is nonnegative definite (see Exercise 6.1a, page 176). Estimates of the variance-covariance matrix or the correlation matrix of the underlying distribution may not be positive definite, however, and in Exercise 6.1d we describe a possible way of adjusting a matrix to be positive definite.

Some of the most important applications of statistics involve the study of the relationship of one variable, often called a "response variable", to other variables. A general model for the relationship of one variable, y, to others, x (a vector), is

$$y \approx f(x). \tag{6.1}$$

A specific form of (6.1) is

$$y = \beta^{\mathrm{T}}x + \epsilon, \tag{6.2}$$

which expresses y as a linear combination of the x's plus some adjustment term. The adjustment term may be modeled as a random variable, in which case we write the equation as

$$Y = \beta^{\mathrm{T}}x + \mathrm{E}.$$

(In this expression "E" is an uppercase epsilon. In the notation we attempt to use consistently, "E" represents a random variable, and "ϵ" represents a realization of the random variable.)

Models that express an asymmetric relationship between some variables ("dependent variables") and other variables ("independent variables") are called regression models. A model such as (6.2) is called a linear regression model. There are many useful variations of the model (6.1) that express other kinds of relationships between the response variable and the other variables.

6.1 Fitting Linear Models with Data

Let us assume a linear relationship between a "response" variable and some "regressor" variables. Suppose we have an n-vector of observations on the response variable, and an $n \times m$ observed matrix, X, of values of the regressor variables. We are interested in the value of β that makes a good fit, that is, so that

$$y \approx X\beta.$$

The regression model may also be written as

$$y = X\beta + \epsilon, \tag{6.3}$$

where ϵ is a vector of deviations ("errors") of the observations from the functional model. The ϵ vector contains the distances of the observations in y from the hyperplane $\beta^{\mathrm{T}}x$ *measured in the direction of the* y *axis.* The objective is to determine a value of β that minimizes some norm of ϵ. The use of the L_2 norm is called "least squares".

Another value to use for β is one that would minimize the distances of the observed values of y from the vector $X\beta$. This use of the L_2 norm is sometimes called "total least squares" (see Golub and Van Loan, 1980). This is a reasonable approach when it is assumed that the observations in X are realizations of some random variable, that is, an "errors-in-variables" model is appropriate (see Fuller, 1987). Golub and Van Loan (1980) give an algorithm for determining a value of β that yields the minimum distances, and the interested reader is referred to that reference.

The statistical problem is to *estimate* β. (Notice the distinction in the phrase "to *estimate* β" and the phrase "to determine a value of β that minimizes ...". The mechanical aspects of the two problems may be the same, of course.) The statistician uses the model and the given observations to explore relationships between the response and the regressors. Considering ϵ to be a realization of a random variable E (a vector) and assumptions about a distribution of the random variable E allow us to make statistical inferences about a "true" β.

6.2 Linear Models and Least Squares

The most common estimator of β is one that minimizes the L_2 norm of the vertical distances in (6.3), that is, to form a least squares fit. The estimator is the b that minimizes the dot product

$$(y - Xb)^{\mathrm{T}}(y - Xb) = \sum (y_i - x_i^{\mathrm{T}} b)^2, \tag{6.4}$$

where x_i^{T} is the i^{th} row of X.

As we saw in Section 3.7 (where we used a slightly different notation), use of elementary calculus to determine the minimum of (6.4) yields the "normal equations",

$$X^{\mathrm{T}} X \widehat{\beta} = X^{\mathrm{T}} y,$$

whose solution is

$$\widehat{\beta} = (X^{\mathrm{T}} X)^{-} X^{\mathrm{T}} y. \tag{6.5}$$

If X is of full rank, the generalized inverse in equation (6.5) is, of course, the inverse, and $\widehat{\beta}$ is the unique least squares estimator. If X is not of full rank, we generally use the Moore-Penrose inverse, $(X^{\mathrm{T}} X)^{+}$, in equation (6.5).

As we saw in equation (3.26), we also have

$$\widehat{\beta} = X^{+} y. \tag{6.6}$$

Equation (6.6) indicates the appropriate way to compute $\widehat{\beta}$. As we have seen many times before, however, we often use an expression without computing the individual terms. Instead of computing X^{+} in equation (6.6) explicitly, we use either Householder or Givens transformations to obtain the orthogonal decomposition

$$X = QR,$$

or

$$X = QRU^{\mathrm{T}},$$

if X is not of full rank. As we have seen, the QR decomposition of X can be performed row-wise using Givens transformations if the data are available only one observation at a time. The equation used for computing $\widehat{\beta}$ is

$$R\widehat{\beta} = Q^{\mathrm{T}} y,$$

which can be solved by back substitution in the triangular matrix R.

Because

$$X^{\mathrm{T}} X = R^{\mathrm{T}} R,$$

the quantities in $X^{\mathrm{T}} X$ or its inverse, which are useful for making inferences using the regression model, can be obtained from the QR decomposition.

If X is not of full rank, the expression (6.6) not only is a least squares solution, it is the one with minimum length (minimum Euclidean norm), as we saw in equations (3.26) and (3.27).

The vector $\widehat{y} = X\widehat{\beta}$ is the projection of the n-vector y onto a space of dimension m. The vector of the model, $y = X\beta$, is also in the m-dimensional space span(X). The projection matrix $I - X(X^{\mathrm{T}}X)^{+}X^{\mathrm{T}}$ projects y onto a $(n-m)$-dimensional residual space that is orthogonal to span(X). Figure 6.1 represents these subspaces and the vectors in them.

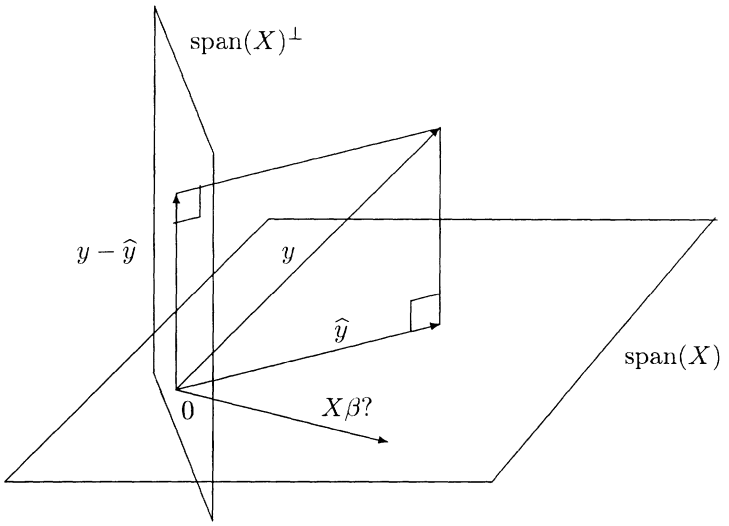


Figure 6.1: The Linear Least-Squares Fit of y with X

In the $(m+1)$-order vector space of the variables, the hyperplane $\widehat{\beta}^{\mathrm{T}}x$ is the estimated model.

The projection matrix $H = X(X^{\mathrm{T}}X)^{+}X^{\mathrm{T}}$ is sometimes called the "hat matrix" because

$$\begin{aligned} \widehat{y} &= X\widehat{\beta} \\ &= X(X^{\mathrm{T}}X)^{+}X^{\mathrm{T}}y \\ &= Hy, \end{aligned} \tag{6.7}$$

that is, it projects y onto $\widehat{y}$ in the span of X. Notice that the hat matrix can be computed without knowledge of the observations in y. The elements of H are useful in assessing the effect of the particular pattern of the regressors on the predicted values of the response. The extent to which a given point in the row space of X affects the regression fit is called its "leverage". The leverage of the i^{th} observation is

$$h_{ii} = x_i^{\mathrm{T}}(X^{\mathrm{T}}X)^{+}x_i.$$

A relatively large value of h_{ii}, compared with the other diagonal elements of the hat matrix, means that the i^{th} observed response, y_i, has a correspondingly relatively large effect on the regression fit.

6.2.1 The Normal Equations and the Sweep Operator

The coefficient matrix in the normal equations, $X^{\mathrm{T}}X$, or the adjusted version, $X_a^{\mathrm{T}}X_a$, is often of interest for reasons other than just to compute the least squares estimators. The adjusted or centered matrix X_a is $X - N^{\mathrm{T}}X$, where N is an $n \times m$ matrix all of whose elements are $1/n$. The condition number of $X^{\mathrm{T}}X$ is the square of the condition number of X, however, and so any ill-conditioning is exacerbated by formation of the cross-products matrix. The adjusted cross-products, $X_a^{\mathrm{T}}X_a$, tends to be better conditioned, so it is usually the one used in the normal equations.

A useful matrix can be formed from the normal equations:

$$\begin{bmatrix} X^{\mathrm{T}}X & X^{\mathrm{T}}y \\ y^{\mathrm{T}}X & y^{\mathrm{T}}y \end{bmatrix}. \tag{6.8}$$

Applying m elementary operations on this matrix, we can get

$$\begin{bmatrix} (X^{\mathrm{T}}X)^{+} & X^{+}y \\ y^{\mathrm{T}}X^{+\mathrm{T}} & y^{\mathrm{T}}y - y^{\mathrm{T}}X(X^{\mathrm{T}}X)^{+}X^{\mathrm{T}}y \end{bmatrix}.$$

(If X is not of full rank, in order to get the Moore-Penrose inverse in this expression, the elementary operations must be applied in a fixed manner.) The matrix in the upper left of the partition is related to the estimated variance-covariance matrix of the particular solution of the normal equations, and it can be used to get an estimate of the variance-covariance matrix of estimates of any independent set of linearly estimable functions of β. The vector in the upper right of the partition is the unique minimum-length solution to the normal equations, $\widehat{\beta}$. The scalar in the lower right partition, which is the Schur complement of the full inverse (see equations (2.6) and (2.8)), is the square of the residual norm. The squared residual norm provides an estimate of the variance of the residuals in (6.3), after proper scaling.

The elementary operations can be grouped into a larger operation, called the "sweep operation", which is performed for a given row. The sweep operation on row i, S_i, of the nonnegative definite matrix A to yield the matrix B, which we denote by

$$\mathrm{S}_i(A) = B,$$

is defined in Algorithm 6.1.

Algorithm 6.1 Sweep of the i^{th} Row

1. If $a_{ii} = 0$ skip the following operations.
2. Set $b_{ii} = a_{ii}^{-1}$.
3. For $j \neq i$, set $b_{ij} = a_{ii}^{-1}a_{ij}$.
4. For $k \neq i$, set $b_{kj} = a_{kj} - a_{ki}a_{ii}^{-1}a_{ij}$. ∎

Skipping the operations if $a_{ii} = 0$ allows the sweep operator to handle non-full rank problems. The sweep operator is its own inverse:

$$\mathrm{S}_i(\mathrm{S}_i(A)) = A.$$

The sweep operator applied to the matrix (6.8) corresponds to adding or removing the i^{th} variable (column) of the X matrix to the regression equation.

6.2.2 Linear Least Squares Subject to Linear Equality Constraints

In the regression model (6.3), it may be known that β satisfies certain constraints, such as all the elements are nonnegative. For constraints of the form $g(\beta) \in C$, where C is some m-dimensional space, we may estimate β by the *constrained least squares estimator*; that is, the vector $\widehat{\beta}_C$ that minimizes the dot product (6.4), among all b that satisfy $g(b) \in C$.

The nature of the constraints may or may not make drastic changes to the computational problem. (The constraints also change the statistical inference problem in various ways, but we do not address that here.) If the constraints are nonlinear, or if the constraints are inequality constraints (such as all the elements are nonnegative), there is no general closed form solution.

It is easy to handle linear equality constraints of the form

$$\begin{aligned} g(\beta) &= L\beta \\ &= c, \end{aligned}$$

where L is a $q \times m$ matrix of full rank. The solution is, analogous to (6.5),

$$\widehat{\beta}_C = (X^{\mathrm{T}}X)^+X^{\mathrm{T}}y + (X^{\mathrm{T}}X)^+L^{\mathrm{T}}(L(X^{\mathrm{T}}X)^+L^{\mathrm{T}})^+(c - L(X^{\mathrm{T}}X)^+X^{\mathrm{T}}y). \quad (6.9)$$

When X is full rank, this result can be derived by using Lagrange multipliers and the derivative of the norm (6.4) (see Exercise 6.4, page 179). When X is not of full rank, it is slightly more difficult to show this, but it is still true. (See a text on linear regression, such as Draper and Smith, 1981.)

The restricted least squares estimate, $\widehat{\beta}_C$, can be obtained (in the (1,2) block) by performing $m + q$ sweep operations on the matrix,

$$\begin{bmatrix} X^{\mathrm{T}}X & X^{\mathrm{T}}y & L^{\mathrm{T}} \\ y^{\mathrm{T}}X & y^{\mathrm{T}}y & c^{\mathrm{T}} \\ L & c & 0 \end{bmatrix}, \quad (6.10)$$

analogous to (6.8).

6.2.3 Weighted Least Squares

In fitting the regression model $y \approx X\beta$ it is often desirable to weight the observations differently, and so instead of minimizing (6.4), we minimize

$$\sum w_i(y_i - x_i^{\mathrm{T}}b)^2,$$

where w_i represents a nonnegative weight to be applied to the i^{th} observation. The purpose of the weight is to control the effect of a given observation on the overall fit. If a model of the form of (6.3)

$$y = X\beta + \epsilon$$

is assumed, and ϵ is taken to be a random variable, such that ϵ_i has variance σ_i, an appropriate value of w_i may be $1/\sigma_i$. (Statisticians almost always naturally assume that ϵ is a random variable. Whereas generally it is modeled this way, here we are allowing for more general interpretations and more general motives in fitting the model.)

The normal equations can be written as

$$\left(X^{\mathrm{T}}\mathrm{diag}((w_1, w_2, \ldots, w_n))X\right)\widehat{\beta} = X^{\mathrm{T}}\mathrm{diag}((w_1, w_2, \ldots, w_n))y.$$

More generally, we can consider W to be a weight matrix that is not necessarily diagonal. We have the same set of normal equations:

$$(X^{\mathrm{T}}WX)\widehat{\beta}_W = X^{\mathrm{T}}Wy. \tag{6.11}$$

When W is a diagonal matrix, the problem is called "weighted least squares". Use of a nondiagonal W is also called weighted least squares, but is sometimes called "generalized least squares". The weight matrix is symmetric and generally positive definite, or at least nonnegative definite. The weighted least squares estimator is

$$\widehat{\beta}_W = (X^{\mathrm{T}}WX)^{+}X^{\mathrm{T}}Wy.$$

As we have mentioned many times, an expression such as this is not necessarily a formula for computation. The matrix factorizations discussed above for the unweighted case can also be used for weighted least squares.

In a model $y = X\beta + \epsilon$ where ϵ is taken to be a random variable with variance-covariance matrix Σ, the choice of W as Σ^{-1} yields estimators with certain desirable statistical properties. (Because this is a natural choice for many models, statisticians sometimes choose the weighting matrix without fully considering the reasons for the choice.)

6.2.4 Updating Linear Regression Statistics

In Section 3.6 we discussed the general problem of updating a least squares solution to an overdetermined system when either the number of equations (rows) or the number of variables (columns) is changed. In the linear regression problem these correspond to adding or deleting observations and adding or deleting terms in the linear model, respectively. The Sherman-Morrison-Woodbury formulas (3.19) and (3.20) can be used to update a solution to reflect these changes.

Another way of approaching the problem of adding or deleting observations is by viewing the problem as weighted least squares. In this approach we also

have more general results for updating of regression statistics. Following Escobar and Moser (1993), we can consider two weighted least squares problems, one with weight matrix W and one with weight matrix V. Suppose we have the solutions $\widehat{\beta}_W$ and $\widehat{\beta}_V$. Now let

$$\Delta = V - W,$$

and use the subscript $*$ on any matrix or vector to denote the subarray that corresponds only to the nonnull rows of Δ. The symbol Δ_*, for example, is the square subarray of Δ consisting of all of the nonzero rows and columns of Δ, and X_* is the subarray of X consisting of all the columns of X and only the rows of X that correspond to Δ_*. From the normal equations (6.11) using W and V, and with the solutions $\widehat{\beta}_W$ and $\widehat{\beta}_V$ plugged in, we have,

$$(X^{\mathrm{T}}WX)\widehat{\beta}_W + (X^{\mathrm{T}}\Delta X)\widehat{\beta}_V = X^{\mathrm{T}}Wy + X^{\mathrm{T}}\Delta y,$$

and so

$$\widehat{\beta}_V - \widehat{\beta}_W = (X^{\mathrm{T}}WX)^{+}X_*^{\mathrm{T}}\Delta_*(y - X\widehat{\beta}_V)_*.$$

This gives

$$(y - X\widehat{\beta}_V)_* = (I + X(X^{\mathrm{T}}WX)^{+}X_*^{\mathrm{T}}\Delta_*)^{+}(y - X\widehat{\beta}_W)_*,$$

and finally,

$$\widehat{\beta}_V = \widehat{\beta}_W + (X^{\mathrm{T}}WX)^{+}X_*^{\mathrm{T}}\Delta_*\left(I + X_*(X^{\mathrm{T}}WX)^{+}X_*^{\mathrm{T}}\Delta_*\right)^{+}(y - X\widehat{\beta}_W)_*.$$

If Δ_* can be written as $\pm GG^{\mathrm{T}}$, using this equation and equation (2.7), on page 62, we have

$$\widehat{\beta}_V = \widehat{\beta}_W \pm (X^{\mathrm{T}}WX)^{+}X_*^{\mathrm{T}}G(I \pm G^{\mathrm{T}}X_*(X^{\mathrm{T}}WX)^{+}X_*^{\mathrm{T}}G)^{+}G^{\mathrm{T}}(y - X\widehat{\beta}_W)_*. \tag{6.12}$$

The sign of GG^{T} is positive when observations are added and negative when they are deleted.

Equation (6.12) is particularly simple in the case where W and V are identity matrices (of different sizes, of course). Suppose, for example, that we have obtained more observations. So the regression model is

$$\begin{bmatrix} y \\ y_+ \end{bmatrix} \approx \begin{bmatrix} X \\ X_+ \end{bmatrix} \beta,$$

where y_+ and X_+ represent the additional observations. (In the following, the reader must be careful to distinguish "+" as a subscript to represent more data, and "+" as a superscript with its usual meaning of a Moore-Penrose inverse.) Suppose we already have the least squares solution for $y \approx X\beta$, say $\widehat{\beta}_W$. Now $\widehat{\beta}_W$ is the weighted least squares solution to the model with the additional data and with weight matrix

$$W = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}.$$

We now seek the solution to the same system with weight matrix V, which is a larger identity matrix. From equation (6.12) the solution is

$$\widehat{\beta} = \widehat{\beta}_W + (X^{\mathrm{T}}X)^{+}X_{+}^{\mathrm{T}}(I + X_{+}(X^{\mathrm{T}}X)^{+}X_{+}^{\mathrm{T}})^{+}(y - X\widehat{\beta}_W)_{*}. \tag{6.13}$$

To add or delete variables from the model, the sweep operator can also be used conveniently, as we mentioned above.

6.2.5 Tests of Hypotheses

For the model (6.3) the general linear hypothesis is

$$\mathrm{H}_0\colon\ L^{\mathrm{T}}\beta = c,$$

where L is $m \times q$, of rank q, and such that $\mathrm{span}(L) \subseteq \mathrm{span}(X)$.

This hypothesis is tested using an F statistic whose numerator is the difference in the residual sum of squares from fitting the model with the restriction $L^{\mathrm{T}}\beta = c$ and the residual sum of squares from fitting the unrestricted model. This reduced sum of squares is

$$(L^{\mathrm{T}}\widehat{\beta} - c)^{\mathrm{T}}\,(L^{\mathrm{T}}(X^{\mathrm{T}}X)^{*}L)^{-1}\,(L^{\mathrm{T}}\widehat{\beta} - c), \tag{6.14}$$

where $(X^{\mathrm{T}}X)^{*}$ is any g_2 inverse of $X^{\mathrm{T}}X$. (See a text on linear models, such as Searle, 1971.)

To compute the quantity in (6.14), first observe

$$L^{\mathrm{T}}(X^{\mathrm{T}}X)^{*}L = (X(X^{\mathrm{T}}X)^{*}L)^{\mathrm{T}}\,(X(X^{\mathrm{T}}X)^{*}L). \tag{6.15}$$

Now if $X(X^{\mathrm{T}}X)^{*}L$, which has rank q, is decomposed as

$$X(X^{\mathrm{T}}X)^{*}L = P\begin{bmatrix} T \\ 0 \end{bmatrix},$$

where P is an $m \times m$ orthogonal matrix and T is a $q \times q$ upper triangular matrix, we can write the reduced sum of squares (6.14) as

$$(L^{\mathrm{T}}\widehat{\beta} - c)^{\mathrm{T}}\,(T^{\mathrm{T}}T)^{-1}\,(L^{\mathrm{T}}\widehat{\beta} - c),$$

or

$$\left((T^{\mathrm{T}})^{-1}(L^{\mathrm{T}}\widehat{\beta} - c)\right)^{\mathrm{T}}\,\left((T^{\mathrm{T}})^{-1}(L^{\mathrm{T}}\widehat{\beta} - c)\right),$$

or

$$v^{\mathrm{T}}v. \tag{6.16}$$

The objective now is to compute v.

Given the relation

$$X^{+} = U\begin{bmatrix} R_1^{-1} & 0 \\ 0 & 0 \end{bmatrix}Q^{\mathrm{T}}$$

as in equation (3.12), we have

$$\begin{aligned} X^+(X^+)^{\mathrm{T}} &= (X^{\mathrm{T}}X)^* \\ &= U \begin{bmatrix} (R_1^{\mathrm{T}}R_1)^{-1} & 0 \\ 0 & 0 \end{bmatrix} U^{\mathrm{T}}, \end{aligned}$$

and so

$$X(X^{\mathrm{T}}X)^* = Q \begin{bmatrix} (R_1^{\mathrm{T}})^{-1} & 0 \\ 0 & 0 \end{bmatrix} U^{\mathrm{T}}.$$

The computations are completed by solving

$$T^{\mathrm{T}}v = L^{\mathrm{T}}\widehat{\beta} - c$$

for the v in (6.16). The reduced sum of squares is then formed as $v^{\mathrm{T}}v$.

6.2.6 D-Optimal Designs

When an experiment is designed to explore the effects of some variables (usually called "factors") on another variable, the settings of the factors (independent variables) should be determined so as to yield a maximum amount of information from a given number of observations. The basic problem is to determine from a set of candidates the best rows for the data matrix X. For example, if there are six factors and each can be set at three different levels, there is a total of $3^6 = 729$ combinations of settings. In many cases, because of the expense in conducting the experiment, only a relatively small number of runs can be made. If, in the case of the 729 possible combinations, only 30 or so runs can be made, the scientist must choose the subset of combinations that will be most informative. A row in X may contain more elements than just the number of factors (because of interactions), but the factor settings completely determine the row.

We may quantify the information in terms of variances of the estimators. If we assume a linear relationship expressed by

$$y = \beta_0 1 + X\beta + \epsilon,$$

and make certain assumptions about the probability distribution of the residuals, the variance-covariance matrix of estimable linear functions of the least squares solution (6.5) are formed from

$$(X^{\mathrm{T}}X)^-\sigma^2.$$

(The assumptions are that the residuals are independently distributed with a constant variance, σ^2. We will not dwell on the statistical properties here, however.) If the emphasis is on estimation of β, then X should be of full rank. In the following we assume X is of full rank, that is, that $(X^{\mathrm{T}}X)^{-1}$ exists.

An objective is to minimize the variances of estimators of linear combinations of the elements of β. We may identify three types of relevant measures

of the variance of the estimator $\widehat{\beta}$: the average variance of the elements of $\widehat{\beta}$, the maximum variance of any elements, and the "generalized variance" of the vector $\widehat{\beta}$. The property of the design resulting from maximizing the information by reducing these measures of variance is called, respectively, A-optimality, E-optimality, and D-optimality. They are achieved when X is chosen as follows:

- A-optimality: minimize $\text{trace}((X^{\mathrm{T}}X)^{-1})$.
- E-optimality: minimize $\rho((X^{\mathrm{T}}X)^{-1})$.
- D-optimality: minimize $\det((X^{\mathrm{T}}X)^{-1})$.

Using the properties of eigenvalues and determinants that we discussed in Chapter 2, we see that E-optimality is achieved by maximizing $\rho(X^{\mathrm{T}}X)$ and D-optimality is achieved by maximizing $\det(X^{\mathrm{T}}X)$.

The D-optimal criterion is probably used most often. If the residuals have a normal distribution (and the other distributional assumptions are satisfied), the D-optimal design results in the smallest volume of confidence ellipsoids for β (see Nguyen and Miller, 1992, and Atkinson and Donev, 1992). The computations required for the D-optimal criterion are the simplest, and this may be another reason it is used often.

To construct an optimal X with a given number of rows, n, from a set of N potential rows, one usually begins with an initial choice of rows, perhaps random, and then determines the effect on the determinant by exchanging a selected row with a different row from the set of potential rows.

If the matrix X has n rows and the row vector x^{T} is appended, the determinant of interest is

$$\det(X^{\mathrm{T}}X + xx^{\mathrm{T}})$$

or its inverse. Using the relationship $\det(AB) = \det(A)\det(B)$, it is easy to see that

$$\det(X^{\mathrm{T}}X + xx^{\mathrm{T}}) = \det(X^{\mathrm{T}}X)(1 + x^{\mathrm{T}}(X^{\mathrm{T}}X)^{-1}x). \tag{6.17}$$

Now if a row x_+^{T} is exchanged for the row x_-^{T}, the effect on the determinant is given by

$$\begin{aligned}\det(X^{\mathrm{T}}X + x_+x_+^{\mathrm{T}} - x_-x_-^{\mathrm{T}}) \;=\; & \det(X^{\mathrm{T}}X) \times \\ & \Big(1 + x_+^{\mathrm{T}}(X^{\mathrm{T}}X)^{-1}x_+ - \\ & x_-^{\mathrm{T}}(X^{\mathrm{T}}X)^{-1}x_-(1 + x_+^{\mathrm{T}}(X^{\mathrm{T}}X)^{-1}x_+) + \\ & (x_+^{\mathrm{T}}(X^{\mathrm{T}}X)^{-1}x_-)^2\Big). \end{aligned} \tag{6.18}$$

(See Exercise 6.8.)

Following Miller and Nguyen (1994), writing $X^{\mathrm{T}}X$ as $R^{\mathrm{T}}R$ from the QR decomposition of X, and introducing z_+ and z_- as

$$Rz_+ = x_+$$

and

$$Rz_- = x_-,$$

we have the right-hand side of (6.18):

$$z_+^{\mathrm{T}} z_+ - z_-^{\mathrm{T}} z_- (1 + z_+^{\mathrm{T}} z_+) + (z_-^{\mathrm{T}} z_+)^2. \tag{6.19}$$

Even though there are $n(N-n)$ possible pairs (x_+, x_-) to consider for exchanging, various quantities in (6.19) need be computed only once. The corresponding (z_+, z_-) are obtained by back substitution using the triangular matrix R. Miller and Nguyen use the Cauchy-Schwarz inequality (2.1) (page 51) to show that the quantity (6.19) can be no larger than

$$z_+^{\mathrm{T}} z_+ - z_-^{\mathrm{T}} z_-; \tag{6.20}$$

hence, when considering a pair (x_+, x_-) for exchanging, if the quantity (6.20) is smaller than the largest value of (6.19) found so far, than the full computation of (6.19) can be skipped. Miller and Nguyen also suggest not allowing the last point added to the design be considered for removal in the next iteration and not allowing the last point removed to be added in the next iteration.

The procedure begins with an initial selection of design points, yielding the $n \times m$ matrix $X^{(0)}$ that is of full rank. At the k^{th} step, each row of $X^{(k)}$ is considered for exchange with a candidate point, subject to the restrictions mentioned above. Equations (6.19) and (6.20) are used to determine the best exchange. If no point is found to improve the determinant, the process terminates. Otherwise, when the optimal exchange is determined, $R^{(k+1)}$ is formed using the updating methods discussed in the previous sections. (The programs of Gentleman, 1974, referred to in Section 3.7.3, can be used.)

6.3 Ill-Conditioning in Statistical Applications

We have described ill-conditioning heuristically as a situation in which small changes in the input data may result in large changes in the solution. Ill-conditioning in statistical modeling is often the result of high correlations among the independent variables. When such correlations exist, the computations may be subject to severe rounding error. This was a problem in using computer software many years ago, as Longley (1967) pointed out. When there are large correlations among the independent variables, the model itself must be examined, as Beaton, Rubin, and Barone (1976) emphasize in reviewing the analysis performed by Longley. Although the work of Beaton, Rubin, and Barone was criticized for not paying proper respect to high-accuracy computations, ultimately it is the utility of the fitted model that counts, not the accuracy of the computations.

Large correlations are reflected in the condition number of the X matrix. A large condition number may indicate the possibility of harmful numerical

errors. Some of the techniques for assessing the accuracy of a computed result may be useful. In particular, the analyst may try the suggestion of Mullet and Murray (1971) to regress $y + dx_j$ on $x_1, \ldots, x_m$, and compare the results with the results obtained from just using y.

Other types of ill-conditioning may be more subtle. Large variations in the leverages may be the cause of ill-conditioning.

Often, numerical problems in regression computations indicate that the linear model may not be entirely satisfactory for the phenomenon being studied. Ill-conditioning in statistical data analysis often means that the approach or the model is wrong.

6.4 Testing the Rank of a Matrix

In Section 3.8 we referred to the rank-revealing QR (or LU) method for estimating the rank of a matrix. "Estimation" in that sense refers to "approximation", rather than to statistical estimation. This is an important distinction that is often lost. Estimation and testing in a statistical sense do not apply to a given entity; these methods of inference apply to properties of a random variable.

Gill and Lewbel (1992) describe a statistical test of the rank of a matrix. The test uses factors from an LDU factorization. The rows and/or columns of the matrices are permuted so that the values of D that are larger in magnitude occur in the earlier positions. The $n \times m$ matrix A (with $n \geq m$, without loss of generality) can be decomposed as

$$\begin{aligned} PAQ &= LDU \\ &= \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & I_{n-m} \end{bmatrix} \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \\ 0 & 0 \end{bmatrix}, \end{aligned} \tag{6.21}$$

where the matrices L_{11}, U_{11}, and D_1 are $r \times r$, and the elements of the diagonal submatrices D_1 and D_2 are arranged in nonincreasing order. If the rank of A is r, the entries in D_2 are 0.

Now let $\widehat{A}$ be an estimate of A based on k such realizations, and assume that $\sqrt{k}\,\mathrm{vec}(\widehat{A} - A)$ is asymptotically $N(0, V)$. (Note that V is $nm \times nm$.) Now if $D_2 = 0$, that is, if A has rank r, and $\widehat{A}$ is decomposed in the same way as A in equation 6.21, then

$$\sqrt{k}\,\mathrm{diag}(\widehat{D}_2)$$

has an asymptotic $N(0, W)$ distribution, and

$$n\widehat{d}_2^{\mathrm{T}} W \widehat{d}_2, \tag{6.22}$$

where

$$\widehat{d}_2 = \mathrm{diag}(\widehat{D}_2)$$

has an asymptotic chi-squared distribution with $(m - r)$ degrees of freedom. If a consistent estimator of W, say $\widehat{W}$, replaces W, the expression (6.22) is

a test statistic for the hypothesis that the rank of A is r. (Note that W is $m-r \times m-r$.)

Gill and Lewbel (1992) derive a consistent estimator to use in the test statistic (6.22). Following their derivation, first, let $\widehat{V}$ be a consistent estimator of V. (It would typically be a sample variance-covariance matrix.) Then

$$(\widehat{Q}^{\mathrm{T}} \otimes \widehat{P})\widehat{V}(\widehat{Q} \otimes \widehat{P}^{\mathrm{T}})$$

is a consistent estimator of the variance-covariance of $\mathrm{vec}(\widehat{P}(\widehat{A}-A)\widehat{Q})$. Next, define the matrices

$$\widehat{H} = \left[-\widehat{L}_{22}^{-1}\widehat{L}_{21}\widehat{L}_{11}^{-1} \;\middle|\; \widehat{L}_{22}^{-1} \;\middle|\; 0\right],$$

$$\widehat{K} = \begin{bmatrix} -\widehat{U}_{11}^{-1}\widehat{U}_{12}\widehat{U}_{22}^{-1} \\ \widehat{U}_{22}^{-1} \end{bmatrix},$$

and T such that

$$\mathrm{vec}(\widehat{D}_2) = T\widehat{d}_2.$$

The matrix T is $(m-r)^2 \times (m-r)$, consisting of a stack of square matrices with 0's in all positions except for a 1 in one diagonal element. The matrix is orthogonal, that is,

$$T^{\mathrm{T}}T = I_{m-r}.$$

The matrix

$$(\widehat{K} \otimes \widehat{H}^{\mathrm{T}})T$$

transforms $\mathrm{vec}(\widehat{P}(\widehat{A}-A)\widehat{Q})$ into $\widehat{d}_2$, hence the variance-covariance estimator, $(\widehat{Q}^{\mathrm{T}} \otimes \widehat{P})\widehat{V}(\widehat{Q} \otimes \widehat{P}^{\mathrm{T}})$, is adjusted by this matrix. The estimator $\widehat{W}$ therefore is given by

$$\widehat{W} = T^{\mathrm{T}}(\widehat{K}^{\mathrm{T}} \otimes \widehat{H})(\widehat{Q}^{\mathrm{T}} \otimes \widehat{P})\widehat{V}(\widehat{Q} \otimes \widehat{P}^{\mathrm{T}})(\widehat{K} \otimes \widehat{H}^{\mathrm{T}})T.$$

The test statistic is

$$n\widehat{d}_2^{\mathrm{T}}\widehat{W}\widehat{d}_2, \tag{6.23}$$

with an approximate chi-squared distribution with $(m-r)$ degrees of freedom.

The decomposition in (6.21) can affect the limiting distribution of $\sqrt{k}\,\mathrm{vec}(\widehat{A}-A)$. The effect can be exacerbated by complete pivoting, and Gill and Lewbel (1992) recommend pivoting be limited to row pivoting.

A test for the rank of matrices has many applications, especially in time series, and Gill and Lewbel give examples of several.

6.5 Stochastic Processes

In many stochastic processes, we encounter problems that can be formulated as a state vector being updated by a transition matrix that represents probabilities. For example, in an animal population the numbers of organisms in various age categories may be of interest. The state vector x_t often contains the counts of organisms in each of k age categories at time t, and the expression

$$x_{t+1} = Mx_t$$

is used to study the process. The M matrix in such a model is of the form

$$\begin{bmatrix} s_0 & s_1 & s_2 & \cdots & s_k \\ p_0 & 0 & 0 & \cdots & 0 \\ 0 & p_1 & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & p_{k-1} & 0 \end{bmatrix},$$

where s_i is the number of offspring produced during the period $(t, t+1)$ by organisms in the i^{th} age category at time t, and p_i is the probability of survival to time $t+1$ of an organism in the i^{th} age category at time t. A relevant question is whether the age distribution ever becomes stable, that is, whether for some T, and for $t \geq T$, the proportions in the vector x_{t+1} are the same as those in x_t, or whether there is a λ such that $x_{t+1} = \lambda x_t$. This is clearly the question of the existence of a positive eigenvalue of the matrix M.

Use of this model for study of population dynamics involves computation of the eigenvalues of M. An eigenvalue of 1 allows a population that remains constant in size for some starting age distributions in x_0. Various starting age distributions are used in this model to study the population dynamics.

Another type of Markov chain model has a matrix of transition probabilities $P = (p_{ij})$, where p_{ij} is the probability of a transition from state i at time t to state j at time $t+1$. Such a transition matrix for a homogeneous Markov chain would have the property that all elements are nonnegative and each column would sum to one. Such a matrix is called a *stochastic matrix*. The magnitude of the eigenvalues of P determines important properties of the Markov chain. See Waterman (1995) for a discussion of applications and for the use of Metropolis sampling for analyzing such Markov chains.

Another type of application arises in the p^{th} order autoregressive time series defined by the stochastic difference equation

$$x_t + \alpha_1 x_{t-1} + \cdots + \alpha_p x_{t-p} = e_t,$$

where the e_t are mutually independent normal random variables with mean 0, and $\alpha_p \neq 0$. If the roots of the associated polynomial $m^p + \alpha_1 m^{p-1} + \cdots + \alpha_p = 0$ are less than 1 in absolute value, we can express the parameters of the time series as

$$R\alpha = -\rho, \tag{6.24}$$

where α is the vector of the α_i's, the i^{th} element of the vector; ρ is the autocovariance of lag i; and the $(i,j)^{\text{th}}$ element of the $p \times p$ matrix $R(h)$ is the autocorrelation between x_i and x_j. Equation (6.24) is called the Yule-Walker equation. Because the autocorrelation depends only on the difference $|i-j|$, the diagonals of R are constant, that is,

$$R = \begin{bmatrix} 1 & \rho_1 & \rho_2 & \cdots & \rho_{p-1} \\ \rho_1 & 1 & \rho_1 & \cdots & \rho_{p-2} \\ \rho_2 & \rho_1 & 1 & \cdots & \rho_{p-3} \\ \vdots & & & \ddots & \vdots \\ \rho_{p-1} & \rho_{p-2} & \rho_{p-3} & \cdots & 1 \end{bmatrix}.$$

Such a matrix is called a Toeplitz matrix. Durbin (1960) used the method shown in Algorithm 6.2 to solve the system (6.24). The algorithm is $O(p)$ (see Golub and Van Loan, 1996).

Algorithm 6.2 Solution of the Yule-Walker System (6.24)

1. Set $k = 0$; $\alpha_1^{(k)} = -\rho_1$; $b^{(k)} = 1$; and $a^{(k)} = -\rho_1$.
2. Set $k = k + 1$.
3. Set $b^{(k)} = (1 - a^2))b(k-1)$.
4. Set $a^{(k)} = -(\rho_{k+1} + \sum_{i=1}^{k} \rho_{k+1-i}\alpha_1^{(k-1)})/b^{(k)}$.
5. For $i = 1, 2, \ldots, k$
   set $y_i = \alpha_i^{(k-1)} + a\alpha_{k+1-i}^{(k-1)}$.
6. For $i = 1, 2, \ldots, k$
   set $\alpha_i^{(k)} = y_i$.
7. Set $\alpha_{k+1}^{(k)} = a$.
8. If $k < p - 1$, go to step 1; otherwise terminate. ∎

The Yule-Walker equations arise in many places in analysis of stochastic processes. Multivariate versions of the equations are used for a vector time series. (See Fuller, 1976, for example.)

Exercises

6.1. Sample variance-covariance matrices and sample correlation matrices.

(a) Let X be an $n \times m$ matrix with real entries. Show that $X^{\mathrm{T}}X$ is nonnegative definite.

(b) Let X be an $n \times m$ matrix, with $n > m$, and with entries sampled independently from a continuous distribution (of a real-valued random variable). What is the probability that $X^{\mathrm{T}}X$ is positive definite?

(c) A Monte Carlo study may involve use of an approximate correlation matrix. The methods we discussed require that the correlation matrix be positive definite. The approximate correlation may not be positive definite, however. This can happen because the given correlation matrix is estimated using $X^{\mathrm{T}}X$ with missing values (i.e., not exactly the same observations are used for each correlation between two variables), or because the practitioner has an estimate of the correlation matrix that was not based on a single sample. Construct an example of a data matrix such that the matrix computed in the way that a sample correlation matrix would be computed using all valid data is not positive definite. *Hint:* Construct a matrix of the form

$$\begin{bmatrix} \mathtt{X} & \mathtt{X} & \mathtt{NaN} \\ \mathtt{X} & \mathtt{X} & \mathtt{NaN} \\ \mathtt{X} & \mathtt{X} & \mathtt{NaN} \\ \mathtt{X} & \mathtt{NaN} & \mathtt{X} \\ \mathtt{X} & \mathtt{NaN} & \mathtt{X} \\ \mathtt{X} & \mathtt{NaN} & \mathtt{X} \\ \mathtt{NaN} & \mathtt{X} & \mathtt{X} \\ \mathtt{NaN} & \mathtt{X} & \mathtt{X} \\ \mathtt{NaN} & \mathtt{X} & \mathtt{X} \end{bmatrix}.$$

A correlation matrix can be constructed from data with missing values in different ways. In one way, the variances that are used in computing the correlations are computed from all available data. If the correlation matrix is computed in this way, it is actually possible that the correlation matrix has off-diagonal elements greater than 1. In this method applied to data as shown above, each variance is computed from 6 observations and each covariance is computed from just 3 observations. If your example does not result in a "correlation matrix" with some off-diagonal element greater than 1, construct a new data matrix that does exhibit this anomaly.

If the variances that are used in computing the correlations are computed only from the data with no missing vales in either variable in the pair, obviously no entry in the correlation matrix will be greater than 1 in absolute value. See Weisberg (1980, Chapter 10), for a discussion of the issues in constructing a correlation matrix using data with missing values.

(d) Consider an $m \times m$, nonsingular matrix, R, with 1's on the diagonal and with all off-diagonal elements less than 1 in absolute value. If this matrix is positive definite, it is a correlation matrix. Suppose, however, that some of the eigenvalues are negative. Iman and Dav-

enport describe a method of adjusting the matrix to a "near-by" matrix that is positive definite. (See Ronald L. Iman and James M. Davenport, 1982, *An Iterative Algorithm to Produce a Positive Definite Correlation Matrix from an "Approximate Correlation Matrix"*, Sandia Report SAND81-1376, Sandia National Laboratories, Albuquerque, New Mexico.) For their method, they assumed the eigenvalues are unique, but this is not necessary in the algorithm.

Before beginning the algorithm, choose a small positive quantity, ϵ, to use in the adjustments, set $k = 0$, and set $R^{(k)} = R$.

1. Compute the eigenvalues of $R^{(k)}$:

$$\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_m,$$

and let p be the number of the eigenvalues that are negative. If $p = 0$, stop. Otherwise, set

$$\lambda_i^* = \begin{cases} \epsilon & \text{if } \lambda_i < \epsilon \\ \lambda_i & \text{otherwise} \end{cases} \qquad \text{for } i = p+1, \ldots \min(2p, m). \quad (6.25)$$

2. Let

$$\sum_i \lambda_i v_i v_i^{\mathrm{T}}$$

be the spectral decomposition of R (equation (2.12), page 69), and form the matrix R^*:

$$R^* = \sum_{i=1}^{p} \epsilon v_i v_i^{\mathrm{T}} + \sum_{i=p+1}^{\min(2p,m)} \lambda_i^* v_i v_i^{\mathrm{T}} + \sum_{i=\min(2p,m)+1}^{m} \lambda_i v_i v_i^{\mathrm{T}}.$$

3. Form $R^{(k)}$ from R^* by setting all diagonal elements to 1.
4. Set $k = k + 1$, and go to step 1. (The algorithm iterates on k until $p = 0$.)

Write a program to implement this adjustment algorithm. Write your program to accept any size matrix and a user-chosen value for ϵ. Test your program on the correlation matrix from Exercise 6.1c above.

(e) Consider some variations of the method in Exercise 6.1d. For example, do not make the adjustments (6.25), or make different ones. Consider different adjustments of R^*; for example, adjust any off-diagonal elements that are greater than 1 in absolute value.

Compare the performance of the variations.

(f) Investigate the convergence of the method in Exercise 6.1d. Note that there are several ways the method could converge.

(g) Suppose the method in Exercise 6.1d converges to a positive definite matrix $R^{(n)}$. Prove that all off-diagonal elements of $R^{(n)}$ are less than 1 in absolute value. (This is true for any positive definite matrix with 1's on the diagonal.)

Another approach to handling an approximate variance-covariance matrix that is not positive definite is to use a modified Cholesky decomposition. If the symmetric matrix S is not positive definite, a diagonal matrix D can be determined so that $S+D$ is positive definite. Eskow and Schnabel (1991) describe a method to determine D with values near zero and to compute a Cholesky decomposition of $S+D$.

6.2. Consider the least squares regression estimator (6.5), for full rank $n \times m$ X $(n > m)$:

$$\widehat{\beta} = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y.$$

(a) Compare this with the estimator

$$\widehat{\beta}_{R(d)} = (X^{\mathrm{T}}X + dI_m)^{-1}X^{\mathrm{T}}y.$$

for $d \geq 0$.
Show that

$$\|\widehat{\beta}_{R(d)}\| \leq \|\widehat{\beta}\|.$$

The estimator $\widehat{\beta}_{R(d)}$ is called the *ridge regression* estimator (Hoerl and Kennard, 1970a, 1970b).

(b) Show that $\widehat{\beta}_{R(d)}$ is the least squares solution to the regression model similar to (5.1), $y = X\beta + \epsilon$, except with some additional, artificial data: y is replaced with

$$\begin{pmatrix} y \\ 0 \end{pmatrix},$$

where 0 is an m-vector of 0's, and X is replaced with

$$\begin{bmatrix} X \\ dI_m \end{bmatrix}. \tag{6.26}$$

Now explain why $\widehat{\beta}_{R(d)}$ is shorter than $\widehat{\beta}$.

6.3. Formally prove from the definition that the sweep operator is its own inverse.

6.4. Consider the regression model

$$y = X\beta + \epsilon, \tag{6.27}$$

subject to the linear equality constraints,

$$L\beta = c, \tag{6.28}$$

and assume that X is full column rank.

(a) Let λ be the vector of Lagrange multipliers. Form

$$(b^{\mathrm{T}}L^{\mathrm{T}} - c^{\mathrm{T}})\lambda$$

and

$$(y - Xb)^{\mathrm{T}}(y - Xb) + (b^{\mathrm{T}}L^{\mathrm{T}} - c^{\mathrm{T}})\lambda.$$

Now differentiate these two expressions with respect to λ and b respectively, set the derivatives equal to zero, and solve to obtain

$$\begin{aligned}\widehat{\beta}_C &= (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y - \frac{1}{2}(X^{\mathrm{T}}X)^{-1}L^{\mathrm{T}}\widehat{\lambda}_C \\ &= \widehat{\beta} - \frac{1}{2}(X^{\mathrm{T}}X)^{-1}L^{\mathrm{T}}\widehat{\lambda}_C\end{aligned}$$

and

$$\widehat{\lambda}_C = -2(L(X^{\mathrm{T}}X)^{-1}L^{\mathrm{T}})^{-1}(c - L\widehat{\beta}).$$

Now combine and simplify these expressions to obtain expression (6.9) (page 166).

(b) Prove that the stationary point obtained in 6.4a actually minimizes the residual sum of squares subject to the equality constraints. *Hint:* First express the residual sum of squares as

$$(y - X\widehat{\beta})^{\mathrm{T}}(y - X\widehat{\beta}) + (\widehat{\beta} - b)^{\mathrm{T}}X^{\mathrm{T}}X(\widehat{\beta} - b),$$

and show that is equal to

$$(y-X\widehat{\beta})^{\mathrm{T}}(y-X\widehat{\beta})+(\widehat{\beta}-\widehat{\beta}_C)^{\mathrm{T}}X^{\mathrm{T}}X(\widehat{\beta}-\widehat{\beta}_C)+(\widehat{\beta}_C-b)^{\mathrm{T}}X^{\mathrm{T}}X(\widehat{\beta}_C-b),$$

which is minimized when $b = \widehat{\beta}_C$.

(c) Show that sweep operations applied to the matrix (6.10), page 166, yield the restricted least squares estimate in the (1,2) block.

(d) For the weighting matrix W, derive the expression, analogous to (6.9), for the generalized or weighted least squares estimator for β in (6.27) subject to the equality constraints (6.28).

6.5. Obtain the "Longley data". (It is a dataset in S-Plus, and it is also available from `statlib`.) Each observation is for a year from 1947 to 1962, and consists of the number of people employed, five other economic variables, and the year itself. Longley (1967) fitted the number of people employed to a linear combination of the other variables, including the year.

(a) Use a regression program to obtain the fit.

(b) Now consider the year variable. The other variables are measured (estimated) at various times of the year, so replace the year variable with a "midyear" variable (i.e., add $\frac{1}{2}$ to each year). Redo the regression. How do your estimates compare?

(c) Compute the L_2 condition number of the matrix of independent variables. Now add a ridge regression diagonal matrix, as in (6.26), and compute the condition number of the resulting matrix. How do the two condition numbers compare?

6.6. Derive a formula similar to equation (6.13) to update $\widehat{\beta}$ due to the deletion of the i^{th} observation.

6.7. When data are used to fit a model such as $y = X\beta + \epsilon$, a large leverage of an observation is generally undesirable. If an observation with large leverage just happens not to fit the "true" model well, it will cause $\widehat{\beta}$ to be farther from β than a similar observation with smaller leverage.

(a) Use artificial data to study influence. There are two main aspects to consider in choosing the data: the pattern of X and the values of the residuals in ϵ. The true values of β are not too important, so β can be chosen as 1. Use 20 observations. First, use just one independent variable ($y_i = \beta_0 + \beta_1 x_i + \epsilon_i$). Generate 20 x_i's more or less equally spaced between 0 and 10, generate 20 ϵ_i's, and form the corresponding y_i's. Fit the model, and plot the data and the model. Now, set $x_{20} = 20$, set ϵ_{20} to various values, form the y_i's and fit the model for each value. Notice the influence of x_{20}.

Now, do similar studies with 3 independent variables. (Do not plot the data, but perform the computations and observe the effect.)

Carefully write up a clear description of your study, with tables and plots.

(b) Heuristically, the leverage of a point arises from the distance of the point to a fulcrum. In the case of a linear regression model the measure of the distance of observation i is

$$\Delta(x_i, X1/n) = \|x_i, X1/n\|.$$

(This is not the same quantity from the hat matrix that is defined as the leverage on page 164, but it should be clear that the influence of a point for which $\Delta(x_i, X1/n)$ is large is greater than that of a point for which the quantity is small.) It may be possible to overcome some of the undesirable effects of differential leverages by use of weighted least squares to fit the model. The weight w_i would be a decreasing function of $\Delta(x_i, X1/n)$.

Now, using datasets similar to those used in the previous part of this exercise, study the use of various weighting schemes to control the influence. Weight functions that may be interesting to try include

$$w_i = e^{-\Delta(x_i, X1/n)}$$

and

$$w_i = \max(w_{\max}, \|\Delta(x_i, X1/n)\|^{-p}),$$

for some $w_{\max}$ and some $p > 0$. (Use your imagination!)

Carefully write up a clear description of your study, with tables and plots.

(c) Now repeat Exercise 6.7b except use a decreasing function of the leverage, h_{ii} from the hat matrix in equation (6.7) instead of the function $\Delta(x_i, X1/n)$.

Carefully write up a clear description of this study, and compare it with the results from Exercise 6.7b.

6.8. Formally prove the relationship expressed in equation (6.18), page 171. *Hint:* Use equation (6.17) twice.

6.9. Given the matrix

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}.$$

Assume the random 3×2 matrix X is such that

$$\text{vec}(X - A)$$

has a $N(0, V)$ distribution, where V is block diagonal with the matrix

$$\begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}$$

along the diagonal. Generate 10 realizations of X matrices, and use them to test that the rank of A is 2. Use the test statistic (6.23) on page 174.

Appendix A

Notation and Definitions

All notation used in this work is "standard". I have opted for simple notation, which, of course, results in a one-to-many map of notation to object classes. Within a given context, however, the overloaded notation is generally unambiguous. I have endeavored to use notation consistently.

This appendix is not intended to be a comprehensive listing of definitions. The Subject Index, beginning on page 217, is a more reliable set of pointers to definitions, except for symbols that are not words.

General Notation

Uppercase italic Latin and Greek letters, A, B, E, Λ, etc., are generally used to represent either matrices or random variables. Random variables are usually denoted by letters nearer the end of the Latin alphabet, X, Y, Z, and by the Greek letter E. Parameters in models, that is, unobservables in the models, whether or not they are considered to be random variables, are generally represented by lower case Greek letters. Uppercase Latin and Greek letters, especially P and Φ, are also used to represent cumulative distribution functions. Also, uppercase Latin letters are used to denote sets. Notice that uppercase Greek letters appear to be written in a roman font instead of an italic font.

Lowercase Latin and Greek letters are used to represent ordinary scalar or vector variables and functions. **No distinction in the notation is made between scalars and vectors**; thus, β may represent a vector and β_i may represent the i^{th} element of the vector β. In another context, however, β may represent a scalar. All vectors are considered to be column vectors, although we may write a vector as $x = (x_1, x_2, \ldots, x_n)$. Transposition of a vector or a matrix is denoted by a superscript "T".

Uppercase calligraphic Latin letters, $\mathcal{F}$, $\mathcal{V}$, $\mathcal{W}$, etc., are generally used to represent vector spaces.

Subscripts generally represent indexes to a larger structure, for example, x_{ij} may represent the $(i,j)^{\text{th}}$ element of a matrix, X. A subscript in paren-

theses represents an order statistic. A superscript in parentheses represents an iteration, for example, $x_i^{(k)}$ may represent the value of x_i at the k^{th} step of an iterative process.

x_i	The i^{th} element of a structure (including a sample, which is a multiset).
$x_{(i)}$	The i^{th} order statistic.
$x^{(i)}$	The value of x at the i^{th} iteration.

Realizations of random variables and placeholders in functions associated with random variables are usually represented by lowercase letters corresponding to the uppercase letters; thus, ϵ may represent a realization of the random variable E.

A single symbol in an italic font is used to represent a single variable. A roman font or a special font is often used to represent a standard operator or a standard mathematical structure. Sometimes a string of symbols in a roman font is used to represent an operator (or a standard function), for example, exp represents the exponential function; but a string of symbols in an italic font on the same baseline should be interpreted as representing a composition (probably by multiplication) of separate objects, for example, exp represents the product of e, x, and p.

A fixed-width font is used to represent computer input or output, for example,

```
a = bx + sin(c).
```

In computer text, a string of letters or numerals with no intervening spaces or other characters, such as `bx` above, represents a single object, and there is no distinction in the font to indicate the type of object.

Some important mathematical structures and other objects are:

$\mathbb{R}$	The field of reals, or the set over which that field is defined.
$\mathbb{R}^d$	The usual d-dimensional vector space over the reals, or the set of all d-tuples with elements in $\mathbb{R}$.
$\mathbb{Z}$	The ring of integers, or the set over which that ring is defined.
i	The imaginary unit, $\sqrt{-1}$ (also, a real number, often an integer used as an index; the meaning should be clear from the context).

Computer Number Systems

Computer number systems are used to simulate the more commonly used number systems. It is important to realize that they have different properties, however. Some notation for computer number systems follows.

$\mathbb{F}$	The set of floating-point numbers with a given precision, on a given computer system, or this set together with the the four operators, +, -, *, and / ($\mathbb{F}$ is similar to $\mathbb{R}$ in some useful ways; see page 4).
$\mathbb{I}$	The set of fixed-point numbers with a given length, on a given computer system, or this set together with the the four operators, +, -, *, and / ($\mathbb{I}$ is similar to $\mathbb{Z}$ in some useful ways; see page 4).
$e_{\min}$ and $e_{\max}$	The minimum and maximum values of the exponent in the set of floating-point numbers with a given length (see page 6).
$\epsilon_{\min}$ and $\epsilon_{\max}$	The minimum and maximum spacings around 1 in the set of floating-point numbers with a given length (see page 9).
ϵ or ϵ_{mach}	The machine epsilon, the same as $\epsilon_{\min}$ (see page 9).
$[\cdot]_c$	The computer version of the object $\cdot$ (see page 18).
NaN	Not-a-Number (see page 12).

General Mathematical Functions and Operators

Functions such as sin, max, span, and so on that are commonly associated with groups of Latin letters are generally represented by those letters in a roman font.

Operators such as d (the differential operator) that are commonly associated with a Latin letter are generally represented by that letter in a roman font.

$\times$	Cartesian or cross product of sets, or multiplication of elements of a field or ring.
$\log x$	The natural logarithm evaluated at x.
$\sin x$	The sine evaluated at x (in radians), and similarly for other trignometric functions.

$\lceil x \rceil$	The ceiling function evaluated at the real number x: $\lceil x \rceil$ is the largest integer less than or equal to x.
$\lfloor x \rfloor$	The floor function evaluated at the real number x: $\lfloor x \rfloor$ is the smallest integer greater than or equal to x.
$x!$	The factorial of x. If x is a positive integer, $x! = x(x-1)\cdots 2\cdot 1$.
$\oplus$	Direct sum of vector spaces (see page 50).
$O(f(n))$	Big O; $g(n) = O(f(n))$ means $g(n)/f(n) \to c$ as $n \to \infty$, where c is a nonzero finite constant.
$o(f(n))$	Little o; $g(n) = o(f(n))$ means $g(n)/f(n) \to 0$ as $n \to \infty$.
$o_P(f(n))$	Convergent in probability; $X(n) = o_P(f(n))$ means that for any positive ϵ, $\Pr(\lvert X(n) - f(n)\rvert > \epsilon) \to 0$ as $n \to \infty$.
d	The differential operator.
δ	A perturbation operator; δx represents a perturbation of x, and not a multiplication of x by δ, even if x is a type of object for which a multiplication is defined.
$\Delta(\cdot,\cdot)$	A real-valued difference function; $\Delta(x,y)$ is a measure of the difference of x and y; for simple objects, $\Delta(x,y) = \lvert x-y\rvert$; for more complicated objects, a subtraction operator may not be defined, and Δ is a generalized difference.
$\tilde{x}$	A perturbation of the object x; $\Delta(x,\tilde{x}) = \delta x$.
$\tilde{x}$	An average of a sample of objects generically denoted by x.
$\bar{x}$	The mean of a sample of objects generically denoted by x.
$\bar{x}$	The complex conjugate of the object x; that is, if $x = r + ic$, then $\bar{x} = r - ic$.
$\mathcal{V}(G)$	For the set of vectors (all of the same order) G, the vector space generated by that set.
$\mathcal{V}(X)$	For the matrix X, the vector space generated by the columns of X.
span(Y)	for Y either a set of vectors or a matrix, the vector space $\mathcal{V}(Y)$

$\perp$ Orthogonality relationship (vectors, see page 64; vector spaces, see page 64).

$\mathcal{V}^\perp$ The orthogonal complement of the vector space $\mathcal{V}$ (see page 64).

∇f For the scalar-valued function f of a vector variable, the gradient (that is, the vector of partial derivatives), also often denoted as g_f.

∇f For the vector-valued function f of a vector variable, the transpose of the Jacobian, which is often denoted as J_f; so $\nabla f = J_f^{\mathrm{T}}$.

A^{T} For the matrix A, its transpose (also used for a vector to represent the corresponding row vector).

A^{H} The conjugate transpose of the matrix A; $A^{\mathrm{H}} = \bar{A}^{\mathrm{T}}$.

A^{-1} The inverse of the square, nonsingular matrix A.

A^+ The g_4 inverse, or the Moore-Penrose inverse, or the pseudoinverse, of the matrix A (see page 63).

A^* A g_2 inverse of the matrix A (see page 63).

A^- A g_1, or generalized, inverse of the matrix A (see page 63).

$\otimes$ Kronecker multiplication (see page 57).

$\mathrm{sign}(x)$ For the vector x, a vector of units corresponding to the signs:

$$\begin{aligned} \mathrm{sign}(x)_i &= 1 \quad \text{if } x_i > 0, \\ &= 0 \quad \text{if } x_i = 0, \\ &= -1 \quad \text{if } x_i < 0; \end{aligned}$$

with a similar meaning for a scalar.

L_p For real $p \geq 1$, a norm formed by accumulating the p^{th} powers of the moduli of individual elements in an object and then taking the $(1/p)^{\text{th}}$ power of the result (see page 71).

$\|\cdot\|$ In general, the norm of the object $\cdot$.

$\|\cdot\|_p$ In general, the L_p norm of the object $\cdot$.

$\|x\|_p$ For the vector x, the L_p norm:

$$\|x\|_p = \left(\sum |x_i|^p\right)^{\frac{1}{p}}$$

(see page 71).

$\|X\|_p$ For the matrix X, the L_p norm:

$$\|X\|_p = \max_{\|v\|_p=1} \|Xv\|_p$$

(see page 72).

$\|X\|_F$ For the matrix X, the Frobenius norm:

$$\|X\|_F = \sqrt{\sum_{i,j} x_{ij}^2}$$

(see page 73).

$\langle x, y \rangle$ The inner product of x and y (see page 50).

$\kappa_p(A)$ The L_p condition number of the nonsingular square matrix A with respect to inversion (see page 76).

$\text{diag}(v)$ For the vector v, the diagonal matrix whose nonzero elements are those of v; that is, the square matrix, A, such that $A_{ii} = v_i$ and for $i \neq j$, $A_{ij} = 0$.

$\text{diag}(A_1, A_2, \ldots, A_k)$
The block diagonal matrix whose submatrices along the diagonal are $A_1, A_2, \ldots, A_k$.

$\text{vec}(A)$ The vector consisting of the columns of the matrix A, all strung into one vector; if the column vectors of A are $a_1, a_2, \ldots, a_m$ then

$$\text{vec}(A) = (a_1^{\mathrm{T}}, a_2^{\mathrm{T}}, \ldots, a_m^{\mathrm{T}}).$$

$\text{vech}(A)$ For the symmetric the matrix A, the vector consisting of the unique elements all strung into one vector:

$$\text{vech}(A) = (a_{11}, a_{21}, a_{22}, a_{31}, \ldots, a_{m1}, \ldots, a_{mm}).$$

$\text{trace}(A)$ The trace of the square matrix A, that is, the sum of the diagonal elements.

$\text{rank}(A)$	The rank of the matrix A, that is, the maximum number of independent rows (or columns) of A.
$\rho(A)$	The spectral radius of the matrix A (the maximum absolute value of its eigenvalues).
$\det(A)$	The determinant of the square matrix A, $\det(A) = \lvert A \rvert$.
$\lvert A \rvert$	The determinant of the square matrix A, ; $\lvert A \rvert = \det(A)$.
$\lvert x \rvert$	The modulus of the real or complex number x; if x is real, $\lvert x \rvert$ is the absolute value of x.

Special Vectors and Matrices

1 or 1_n	A vector (of length n) whose elements are all 1's.
0 or 0_n	A vector (of length n) whose elements are all 0's.
I or I_n	The $(n \times n)$ identity matrix.
e_i	The i^{th} unit vector (with implied length) (see page 60).
E_{jk}	The $(i,j)^{\text{th}}$ elementary permutation matrix (see page 65).

Models and Data

A form of model used often in statistics and applied mathematics has three parts: a left-hand side representing an object of primary interest; a function of another variable and a parameter, each of which is likely to be a vector; and an adjustment term to make the right-hand side equal the left-hand side. The notation varies depending on the meaning of the terms. One of the most common models used in statistics, the linear regression model with normal errors, is written as

$$Y = \beta^{\mathrm{T}} x + \mathrm{E}. \tag{A.1}$$

The adjustment term is a random variable, denoted by an uppercase epsilon. The term on the left is also a random variable. This model does not represent observations or data. A slightly more general form is

$$Y = f(x;\theta) + \mathrm{E}. \tag{A.2}$$

A single observation or a single data item that corresponds to model (A.1) may be written as

$$y = \beta^{\mathrm{T}} x + \epsilon,$$

or, if it is one of several,

$$y_i = \beta^{\mathrm{T}} x_i + \epsilon_i.$$

Similar expressions are used for a single data item that corresponds to model (A.2).

In these cases, rather than being a random variable, ϵ or ϵ_i may be a realization of a random variable, or it may just be an adjustment factor with no assumptions about its origin.

A set of n such observations is usually represented in an n-vector y, a matrix X with n rows, and an n-vector ϵ:

$$y = X\beta + \epsilon$$

or

$$y = f(X; \theta) + \epsilon.$$

Appendix B

Solutions and Hints for Selected Exercises

1.2. The function is $\log(n)$ and Euler's constant is 0.57721...

1.5. 2^{-56} (The standard has 53 bits, normalized, so the last bit is 2^{-55}, and half of that is 2^{-56}.)

1.6a. normalized: $2b^{p-1}(b-1)(e_{\max} - e_{\min} + 1) + 1$
nonnormalized: $2b^{p-1}(b-1)(e_{\max} - e_{\min} + 1) + 1 + 2b^{p-1}$

1.6b. normalized: $b^{e_{\min}-1}$
nonnormalized: $b^{e_{\min}-p}$

1.6c. $1 + b^{-p}$

1.6d. b^p

1.6e. 22

1.11. First of all, we recognize that the full sum in each case is 1. We therefore accumulate the sum from the direction in which there are fewer terms. After computing the first term from the appropriate direction, take a logarithm to determine a scaling factor, say s^k. (This term will be the smallest in the sum.) Next, proceed to accumulate terms until the sum is of a different order of magnitude than the next term. At that point, perform a scale adjustment by dividing by s. Resume summing, making similar scale adjustments as necessary, until the limit of the summation is reached.

1.12. The result is close to 1.
What is relevant here is that numbers close to 1 have only a very few digits of accuracy; therefore, it would be better to design this program so

that it returns $1 - \Pr(X \leq x)$ (the "significance level"). The purpose and the anticipated use of a program determines how it should be designed.

1.16c.

$$
\begin{array}{l}
a = x_1 \\
b = y_1 \\
s = 0 \\
\text{for } i = 2, n \\
\{ \\
\quad d = (x_i - a)/i \\
\quad e = (y_i - b)/i \\
\quad a = d + a \\
\quad b = e + b \\
\quad s = i(i-1)de + s \\
\}
\end{array}
$$

2.1. Let one vector space consist of all vectors of the form $(a, 0)$ and the other consist of all vectors of the form $(0, b)$. The vector (a, b) is not in the union if $a \neq 0$ and $b \neq 0$.

2.12. Let A and B be such that AB is defined.

$$
\begin{aligned}
\|AB\|_F^2 &= \sum_{ij} \left| \sum_k a_{ik} b_{kj} \right|^2 \\
&\leq \sum_{ij} \left(\sum_k a_{ik}^2 \right) \left(\sum_k b_{kj}^2 \right) \quad \text{(Cauchy – Schwarz)} \\
&= \left(\sum_{i,k} a_{ik}^2 \right) \left(\sum_{k,j} b_{kj}^2 \right) \\
&= \|A\|_F^2 \|B\|_F^2
\end{aligned}
$$

2.16. First, show that

$$
\max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \left(\min_{x \neq 0} \frac{\|A^{-1}x\|}{\|x\|} \right)^{-1}
$$

and

$$
\max_{x \neq 0} \frac{\|A^{-1}x\|}{\|x\|} = \left(\min_{x \neq 0} \frac{\|Ax\|}{\|x\|} \right)^{-1}
$$

2.17. 1. No; 2. Yes; 3. No; 4. No.

2.18. A very simple example is

$$
\begin{bmatrix} 1 & 1+\epsilon \\ 1 & 1 \end{bmatrix},
$$

where $\epsilon < b^{-p}$, because in this case, the matrix stored in the computer would be singular. Another example is

$$\begin{bmatrix} 1 & a(1+\epsilon) \\ a(1+\epsilon) & a^2(1+2\epsilon) \end{bmatrix},$$

where ϵ is the machine epsilon.

3.3a. The matrix at the first elimination is

$$\begin{bmatrix} 2 & 5 & 3 & 19 \\ 1 & 4 & 1 & 12 \\ 1 & 2 & 2 & 9 \end{bmatrix}.$$

The solution is $(3, 2, 1)$.

3.3b. The matrix at the first elimination is

$$\begin{bmatrix} 5 & 2 & 3 & 19 \\ 4 & 1 & 1 & 12 \\ 2 & 1 & 2 & 9 \end{bmatrix},$$

and x_1 and x_2 have been interchanged.

3.3c.

$$D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$L = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 2 & 0 \end{bmatrix}$$

$$U = \begin{bmatrix} 0 & -5 & -3 \\ 0 & 0 & -1 \\ 0 & 0 & -2 \end{bmatrix}$$

$$\rho((D+L)^{-1}U) = 0.9045.$$

3.4a. $O(nk)$

3.4c. At each successive stage in the fan-in, the number of processors doing the additions goes down by approximately one-half.
If $p \approx k$, then $O(n \log k)$ (fan-in on one element of c at a time)
If $p \approx nk$, then $O(\log k)$ (fan-in on all elements of c simultaneously)
If p is a fixed constant smaller than k, the order of time does not change; only the multiplicative constant changes.
Notice the difference in order of time and order of number of computations. Often there is very little that can be done about the order of computations.

3.4d. Because in a serial algorithm the magnitudes of the summands become more and more different. In the fan-in, they are more likely to remain relatively equal. Adding magnitudes of different quantities results in benign roundoff; but many benign roundoffs become bad. (This is not catastrophic cancellation.) Clearly, if all elements are nonnegative, this argument would hold. Even if the elements are randomly distributed, there is likely to be a drift in the sum (this can be thought of as a random walk). *There is* **no difference** *in the number of computations.*

3.4e. Case 1: $p \approx n$. Give each c_i a processor – do outer loop on each. This would likely be more efficient because all processors are active at once.
Case 2: $p \approx nk$. Give each $a_{ij}b_j$ a processor – fan-in for each. This would be the same as the other.
If p is a fixed constant smaller than n, set it up as in Case 1, using n/p groups of c_i's.

3.4f. If $p \approx n$, then $O(k)$.
If $p \approx nk$, then $O(\log k)$.
If p is some small fixed constant, the order of time does not change; only the multiplicative constant changes.

3.9b. Given the $n \times m$ matrix A, with $n \geq m$. It is assumed that A is of full rank, but this should be checked in the program. In the following, $q_k^{(0)}$ is the k^{th} column of A.

0. Set $k = 1$.
1. Set $r_{kk} = \|q_k^{(k-1)}\|_2$.
2. If $r_{kk} = 0$, the first k columns are dependent; set error flag and exit.
3. Set $q_k^{(k)} = r_{kk}q^{(k-1)}$.
4. Set $j = k + 1$.
5. Set $r_{kj} = q_j^{(k-1)^{\mathrm{T}}} q_k^{(k)}$.
6. Set $q_k^{(k)} = r_{kk}q^{(k-1)}$.
7. If $j < m$, set $j = j + 1$ and go to Step 5.
8. Set $k = k + 1$; if $k < m$, go to Step 1.
9. Set $r_{kk} = \|q_k^{(k-1)}\|_2$.
10. Set $q_k^{(k)} = r_{kk}q^{(k-1)}$.

3.11a. $nm(m + 1) - m(m + 1)/2$. (Remember $A^{\mathrm{T}}A$ is symmetric.)

3.11g. Use of the normal equations with the Cholesky decomposition requires only about half as many flops as the QR, when n is much larger than m. The QR method oftens yields better accuracy, however.

4.1a. 1

4.1b. 1

4.1d. 1 (All that was left was to determine the probability that $c_n \neq 0$ and $c_{n-1} \neq 0$.)

4.2a. 11.6315

4.2b.

$$\begin{bmatrix} 3.08 & -0.66 & 0 & 0 \\ -0.66 & 4.92 & -3.27 & 0 \\ 0 & -3.27 & 7.00 & -3.74 \\ 0 & 0 & -3.74 & 7.00 \end{bmatrix}.$$

5.1. Here is a recursive Matlab function for the Strassen algorithm due to Coleman and Van Loan. When it uses the Strassen algorithm, it requires the matrices to have even dimension.

```
   function C = strass(A,B,nmin)
%
% Strassen matrix multiplication C=AB
%       A, B must be square and of even dimension
% From Coleman and Van Loan
% If n <= nmin, the multiplication is done conventionally
%
   [n n ] = size(A);
   if n <= nmin
      C = A * B;   % n is small, get C conventionally
   else
      m = n/2; u = 1:m; v = m+1:n;
      P1 = strass(A(u,u)+A(v,v), B(u,u)+B(v,v), nmin);
      P2 = strass(A(v,u)+A(v,v), B(u,u),        nmin);
      P3 = strass(A(u,u),        B(u,v)-B(v,v), nmin);
      P4 = strass(A(v,v),        B(v,u)-B(u,u), nmin);
      P5 = strass(A(u,u)+A(u,v), B(v,v),        nmin);
      P6 = strass(A(v,u)-A(u,u), B(u,u)+B(u,v), nmin);
      P7 = strass(A(u,v)-A(v,v), B(v,u)+B(v,v), nmin);
      C = [P1+P4-P5+P7 P3+P5; P2+P4 P1+P3-P2+P6];
   end
```

5.3a.

```
      real a(4,3)
      data a/3.,6.,8.,2.,5.,1.,6.,3.,6.,2.,7.,1./
      n = 4
      m = 3
```

```
x1 = a(2,2) ! Temporary variables must be used because of
x2 = a(4,2) ! the side effects of srotg.
call srotg(x1, x2,, c, s)
call srot(m, a(2,1), n, a(4,1), n, c, s)
print *, c, s
print *, a
end
```

This yields 0.3162278 and 0.9486833 for c and s. The transformed matrix is

$$\begin{bmatrix} 3.000000 & 5.000000 & 6.000000 \\ 3.794733 & 3.162278 & 1.581139 \\ 8.000000 & 6.000000 & 7.000000 \\ -5.059644 & -0.00000002980232 & -1.581139 \end{bmatrix}$$

5.5. 10.7461941829033 and 10.7461941829034.

6.1a. Let y be a vector of length m, and let $z = Xy$. Then

$$y^{\mathrm{T}}(X^{\mathrm{T}}X)y = (yX)^{\mathrm{T}}(Xy) = \sum z_i^2 \geq 0.$$

6.1. 1

6.1f. This is an open question. If you get a proof, submit it for publication. You may wish to try several examples and observe the performance of the intermediate steps. I know of no case in which the method has not converged.

6.4d.

$\widehat{\beta}_{W,C} =$

$$(X^{\mathrm{T}}WX)^{-1}X^{\mathrm{T}}Wy + (X^{\mathrm{T}}WX)^{-1}L^{\mathrm{T}}(L(X^{\mathrm{T}}WX)^{+}L^{\mathrm{T}})^{+}(c - L(X^{\mathrm{T}}WX)^{+}X^{\mathrm{T}}Wy)$$

Bibliography

As might be expected, the literature in the interface of computer science, numerical analysis, and statistics is quite diverse; and relevant articles are likely to appear in journals devoted to quite different disciplines. There are at least ten journals and serials whose titles contain some variants of both "computing" and "statistics"; but there are far more journals in numerical analysis and in areas such as "computational physics", "computational biology", and so on that publish articles relevant to the fields of statistical computing and computational statistics. The journals in the mainstream of statistics also have a large proportion of articles in the fields of statistical computing and computational statistics because, as we suggested in the preface, recent developments in statistics and in the computational sciences have paralleled each other to a large extent.

There are two well-known learned societies whose primary focus is in statistical computing: the International Association for Statistical Computing (IASC), which is an affiliated society of the International Statistical Institute, and the Statistical Computing Section of the American Statistical Association (ASA). The Statistical Computing Section of the ASA has a regular newsletter carrying news and notices as well as articles on practicum. The activities of the Society for Industrial and Applied Mathematics (SIAM) are often relevant to computational statistics.

There are two regular conferences in the area of computational statistics: COMPSTAT, held biennially in Europe and sponsored by the IASC, and the Interface Symposium, generally held annually in North America and sponsored by the Interface Foundation of North America with cooperation from the Statistical Computing Section of the ASA.

In addition to literature and learned societies in the traditional forms, an important source of communication and a repository of information are computer databases and forums. In some cases the databases duplicate what is available in some other form, but often the material and the communications facilities provided by the computer are not available elsewhere.

Literature in Computational Statistics

In the Library of Congress classification scheme, most books on statistics, including statistical computing, are in the QA276 section, although some are classified under H, HA, and HG. Numerical analysis is generally in QA279, and computer science in QA76. Many of the books in the interface of these disciplines are classified in these or other places within QA.

Current Index to Statistics, published annually by the American Statistical Association and the Institute for Mathematical Statistics, contains both author and subject indexes that are useful in finding journal articles or books in statistics. The *Index* is available in hard copy and on CD-ROM. The CD-ROM version with software developed by Ron Thisted and Doug Bates is particularly useful. In passing, I take this opportunity to acknowledge the help this database and software were to me in tracking down references for this book.

The Association for Computing Machinery (ACM) publishes an annual index, by author, title, and keyword, of the literature in the computing sciences.

Mathematical Reviews, published by the American Mathematical Society (AMS), contains brief reviews of articles in all areas of mathematics. The areas of "Statistics", "Numerical Analysis", and "Computer Science" contain reviews of articles relevant to computational statistics. The papers reviewed in *Mathematical Reviews* are categorized according to a standard system that has slowly evolved over the years. In this taxonomy, called the AMS MR classification system, "Statistics" is 62Xyy; "Numerical Analysis", including random number generation, is 65Xyy; and "Computer Science" is 68Xyy. ("X" represents a letter and "yy" represents a two-digit number.)

Mathematical Reviews is also available to subscribers via the World Wide Web at MathSciNet.

There are various handbooks of mathematical functions and formulas that are useful in numerical computations. Three that should be mentioned are Abramowitz and Stegun (1964), Spanier and Oldham (1987) and Thompson (1997). Anyone doing serious scientific computations should have ready access to at least one of these volumes.

Almost all journals in statistics have occasional articles on computational statistics and statistical computing. The following is a list of journals and proceedings that emphasize this field.

ACM Transactions on Mathematical Software, published quarterly by the ACM (Association for Computing Machinery). (Includes algorithms in Fortran and C. Most of the algorithms are available through `netlib`. The ACM collection of algorithms is sometimes called *CALGO*.)

ACM Transactions on Modeling and Computer Simulation, published quarterly by the ACM.

Applied Statistics, published quarterly by the Royal Statistical Society. (Includes algorithms in Fortran. Most of the algorithms are available through `statlib` at Carnegie Mellon University. Some of these algorithms, with corrections, were collected by Griffiths and Hill, 1985)

Communications in Statistics — Simulation and Computation, published quarterly by Marcel Dekker. (Includes algorithms in Fortran. Until 1982, this journal was designated as *Series B*.)

Computational Statistics, published quarterly by Physica-Verlag. (Formerly called *Computational Statistics Quarterly*.)

Computational Statistics. Proceedings of the xxth Symposium on Computational Statistics (COMPSTAT), published biennially by Physica-Verlag. (Not refereed.)

Computational Statistics and Data Analysis, published quarterly by North Holland. (This is also the official journal of the International Association for Statistical Computing.)

Computing Science and Statistics. This is an annual publication containing papers presented at the Interface Symposium. Until 1992, these proceedings were named *Computer Science and Statistics: Proceedings of the xxth Symposium on the Interface*. (The 24$^{\text{th}}$ symposium was held in 1992.) These proceedings are now published by the Interface Foundation of North America. (Not refereed.)

Journal of Computational and Graphical Statistics, published quarterly by the American Statistical Association.

Journal of Statistical Computation and Simulation, published quarterly by Gordon Breach.

Proceedings of the Statistical Computing Section, published annually by the American Statistical Association. (Not refereed.)

SIAM Journal on Scientific Computing, published bimonthly by SIAM. This journal was formerly *SIAM Journal on Scientific and Statistical Computing*. (Is this a step backward?)

Statistics and Computing, published quarterly by Chapman & Hall.

World Wide Web, News Groups, List Servers, and Bulletin Boards

The best way of storing information is in a digital format that can be accessed by computers. In some cases the best way for people to access information is by computers; in other cases the best way is via hard copy, which means that the information stored on the computer must go through a printing process resulting in books, journals, or loose pages.

A huge amount of information and raw data is available online. Much of it is in publicly accessible sites. Some of the repositories give space to ongoing discussions to which anyone can contribute.

There are various ways of remotely accessing the computer databases and discussion groups. The high-bandwidth wide-area network called the "Internet" is the most important way to access information. Early development of the Internet was due to initiatives within the United States Department of Defense

and the National Science Foundation. The Internet is making fundamental changes to the way we store and access information.

The references that I have cited in this text are generally traditional books, journal articles, or compact disks. This usually means that the material has been reviewed by someone other than the author. It also means that the author possibly has newer thoughts on the same material. The Internet provides a mechanism for the dissemination of large volumes of information that can be updated readily. The ease of providing material electronically is also the source of the major problem with the material: it is often half-baked and has not been reviewed critically. Another reason that I have refrained from making frequent reference to material available over the Internet is the unreliability of some sites. It has been estimated that the average life of a Web site is 45 days (in early 1998).

The World Wide Web (WWW)

Mechanisms to access information from various sites on the Internet have been developed, beginning with early work at CERN in Switzerland. The development of the Mosaic Web browser at the National Center for Supercomputer Applications at the University of Illinois marked a major turning point in ease-of-access of information over the Internet.

The Web browsers rely on standard ways of formatting text and images and of linking sites. These methods are independent of the Internet; indeed, they are useful on a single computer for developing an information access system. The basic coding schemes are incorporated in `html` and `xml`. The main new facility provided by `xml` are content tags, which allow specification of the meanings of markings, thus facilitating searches. The World Wide Web Consortium (W3C) povides directions and promotes standards for these markup languages. See

`http://www.w3.org/`

A very important extension of `xml` is `mathml`, which provides special markups and tags for mathematical objects.

Actions can be initiated remotely over the Web, using programming languages such as Java.

For linking Internet sites, the Web browsers use a "Universal Resource Locator"(URL), which determines the location and the access method. "URL" is also used to mean the method and the site ("location", instead of "locator"); for example,

`http://www.science.gmu.edu/~jgentle/linbk`

is called a URL. (As mentioned in the preface, this is the URL for a site that I maintain to store information about this book.)

For statistics, one of the most useful sites on the Internet is the electronic repository `statlib`, maintained at Carnegie Mellon University, which contains programs, datasets, and other items of interest. The URL is

`http://lib.stat.cmu.edu`.

The collection of algorithms published in *Applied Statistics* is available in `statlib`. These algorithms are sometimes called the *ApStat* algorithms.

The `statlib` facility can also be accessed by email or anonymous `ftp` at

`statlib@temper.stat.cmu.edu`.

An automatic email processor will reply with files that provide general information or programs and data. The general introductory file can be obtained by sending email to the address above with the message "`send index`".

Another very useful site for scientific computing is `netlib`, which was established by research workers at AT&T (now Lucent) Bell Laboratories and national laboratories, primarily Oak Ridge National Laboratories. The URL is

`http://www.netlib.org`

The collection of ACM algorithms (*CALGO*) is available in `netlib`.

There is also an X Windows, socket-based system for accessing `netlib`, called `Xnetlib`; see Dongarra, Rowan, and Wade (1995).

The Guide to Available Mathematical Software (GAMS), to which I have referred several times in this book, can be accessed at

`http://gams.nist.gov`

A different interface, using Java, is available at

`http://math.nist.gov/HotGAMS/`

There are two major problems in using the WWW to gather information. One is the sheer quantity of information and the number of sites providing information. The other is the "kiosk problem"; anyone can put up material. Sadly, the average quality is affected by a very large denominator. The kiosk problem may be even worse than a random selection of material; the "fools in public places" syndrome is much in evidence.

There is not much that can be done about the second problem. It was not solved for traditional postings on uncontrolled kiosks, and it will not be solved on the WWW.

For the first problem, there are remarkable programs that automatically crawl through WWW links to build a database that can be searched for logical combinations of terms and phrases. Such systems and databases have been built by several people and companies.
Two of the most useful are Alta Vista, provided by Digital Equipment Corporation, at

`http://www.altavista.digital.com`

and HotBot at

`http://www.hotbot.com`

In a study by Lawrence and Giles (1998), these two full-text search engines provided far more complete coverage of the scientific literature than four other

search engines considered. The Lawrence and Giles study indicated that use of all six search engines provided about 3.5 times as many documents on average as use of just a single engine.

A very widely used search program is "Yahoo" at

`http://www.yahoo.com`

A neophyte can be quickly disabused of an exaggerated sense of the value of such search engines by doing a search on "Monte Carlo". Aside from the large number of hits that relate to a car and to some place in Europe, the hits (in mid 1998) that relate to the interesting topic are dominated by references to some programs for random number generation put together by a group at a university somewhere. (Of course, "interesting" is in the eye of the beholder.)

It is not clear at this time what will be the media for the scientific literature within a few years. Many of the traditional journals will be converted to an electronic version of some kind. Journals will become Web sites. That is for certain; the details, however, are much less certain. Many bulletin boards and discussion groups have already evolved into "electronic journals'. A publisher of a standard commercial journal has stated that "we reject 80% of the articles submitted to our journal; those are the ones you can find on the Web". Lesk (1997) discusses many of the issues that must be considered as the standard repositories of knowledge change from paper books and journals to digital libraries.

References

The following bibliography obviously covers a wide range of topics in statistical computing and computational statistics. Except for a few of the general references, all of these entries have been cited in the text.

The purpose of this bibliography is to help the reader get more information; hence I eschew "personal communications" and references to technical reports that may or may not exist. Those kinds of references are generally for the author rather than for the reader.

A Note on the Names of Authors

In these references, I have generally used the names of authors as they appear in the original sources. This may mean that the same author will appear with different forms of names, sometimes with given names spelled out, and sometimes abbreviated. In the author index, beginning on page 213, I use a single name for the same author. The name is generally the most unique (i.e., least abbreviated) of any of the names of that author in any of the references. This convention may occasionally result in an entry in the author index that does not occur exactly in any references. A reference to J. Paul Jones together with one to John P. Jones, if I know that the two names refer to the same person, would result in an Author Index entry for John Paul Jones.

Abramowitz, Milton, and Irene A. Stegun (Editors) (1964), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards (NIST), Washington. (Reprinted by Dover Publications, Inc., New York.)

Alefeld, Göltz, and Jürgen Herzberger (1983), *Introduction to Interval Computation*, Academic Press, New York.

Amdahl, G. M. (1967), Validity of the single processor approach to achieving large-scale computing capabilities, *Proceedings of the American Federation of Information Processing Societies* **30**, Washington, D.C., 483-485.

Anderson, E.; Z. Bai; C. Bischof; J. Demmel; J. Dongarra; J. DuCroz; A. Greenbaum; S. Hammarling; A. McKenney; S. Ostrouchov; and D. Sorensen (1995), *LAPACK Users' Guide*, second edition, Society for Industrial and Applied Mathematics, Philadelphia.

ANSI (1978), *American National Standard for Information Systems – Programming Language FORTRAN*, Document X3.9-1978, American National Standards Institute, New York.

ANSI (1989), *American National Standard for Information Systems – Programming Language C*, Document X3.159-1989, American National Standards Institute, New York.

ANSI (1992), *American National Standard for Information Systems – Programming Language Fortran-90*, Document X3.9-1992, American National Standards Institute, New York.

Atkinson, A. C., and A. N. Donev (1992), *Optimum Experimental Designs*, Oxford University Press, Oxford, United Kingdom.

Axelsson, Owe (1994), *Iterative Solution Methods*, Cambridge University Press, Cambridge, United Kingdom.

Bailey, David H. (1993), Algorithm 719: Multiprecision translation and execution of FORTRAN programs, *ACM Transactions on Mathematical Software* **19**, 288–319.

Bailey, David H.; King Lee; and Horst D. Simon (1990), Using Strassen's algorithm to accelerate the solution of linear systems, *Journal of Supercomputing* **4**, 358–371.

Bareiss, E. H., and J. L. Barlow (1980), Roundoff error distribution in fixed point multiplication, *BIT (Nordisk Tidskrift for Informationsbehandling)* **20** 247–250.

Basilevsky, Alexander (1983), *Applied Matrix Algebra in the Statistical Sciences*, North Holland, New York.

Beaton, Albert E.; Donald B. Rubin; and John L. Barone (1976), The acceptability of regression solutions: Another look at computational accuracy, *Journal of the American Statistical Association* **71**, 158–168.

Bickel, Peter J., and Joseph A. Yahav (1988), Richardson extrapolation and the bootstrap, *Journal of the American Statistical Association* **83**, 387–393.

Bischof, Christian H. (1990), Incremental condition estimation, *SIAM Journal of Matrix Analysis and Applications* **11**, 312–322.

Björck, Åke (1967), Solving least squares problems by Gram-Schmidt orthogonalization, *BIT* **7**, 1–21.

Björck, Åke (1996), *Numerical Methods for Least Squares Problems*, Society for Industrial and Applied Mathematics, Philadelphia.

Brent, Richard P. (1978), A FORTRAN multiple-precision arithmetic package, *ACM Transactions on Mathematical Software* **4**, 57–70.

Brown, Peter N., and Homer F. Walker (1997), GMRES on (nearly) singular systems, *SIAM Journal of Matrix Analysis and Applications* **18**, 37–51.

Bruce, Andrew, and Hong-Ye Gao (1996), *Applied Wavelet Analysis with S-Plus*, Springer-Verlag, New York.

Bunch, James R., and Linda Kaufman (1977), Some stable methods for calculating inertia and solving symmetric linear systems, *Mathematics of Computation* **31**, 163–179.

Calvetti, Daniela (1991), Roundoff error for floating point representation of real data, *Communications in Statistics* **20**, 2687–2695.

Carrig, James J., Jr., and Gerard G. L. Meyer (1997), Efficient Householder QR factorization for superscalar processors, *ACM Transactions on Mathematical Software* **23**, 362–378.

Chaitin-Chatelin, Françoise, and Valérie Frayssé (1996), *Lectures on Finite Precision Computations*, Society for Industrial and Applied Mathematics, Philadelphia.

Chambers, John M. (1977), *Computational Methods for Data Analysis*, John Wiley & Sons, New York.

Chambers, John M., and Trevor J. Hastie (Editors) (1992), *Statistical Models in S*, Wadsworth & Brooks/Cole, Pacific Grove, California.

Chan, T. F. (1982a), An improved algorithm for computing the singular value decomposition, *ACM Transactions on Mathematical Software* **8**, 72–83.

Chan, T. F. (1982b), Algorithm 581: An improved algorithm for computing the singular value decomposition, *ACM Transactions on Mathematical Software* **8**, 84–88.

Chan, T. F.; G. H. Golub; and R. J. LeVeque (1982), Updating formulae and a pairwise algorithm for computing sample variances, *Compstat 1982: Proceedings in Computational Statistics* (edited by H. Caussinus, P. Ettinger, and R. Tomassone), Physica-Verlag, Vienna, 30–41.

Chan, Tony F.; Gene H. Golub; and Randall J. LeVeque (1983), Algorithms for computing the sample variance: Analysis and recommendations, *The American Statistician* **37**, 242–247.

Chan, Tony F., and John Gregg Lewis (1979), Computing standard deviations: Accuracy, *Communications of the ACM* **22**, 526–531.

Cline, Alan K.; Andrew R. Conn; and Charles F. Van Loan (1982), Generalizing the LINPACK condition estimator, *Numerical Analysis, Mexico, 1981* (edited by J. P. Hennart), Springer-Verlag, Berlin, 73–83.

Cline, A. K.; C. B. Moler; G. W. Stewart; and J. H. Wilkinson (1979), An estimate for the condition number of a matrix, *SIAM Journal of Numerical Analysis* **16**, 368–375.

Cline, A. K., and R. K. Rew (1983), A set of counter-examples to three condition number estimators, *SIAM Journal on Scientific and Statistical Computing* **4**, 602–611.

Cody, W. J. (1988a), Floating-point standards — theory and practice, *Reliability in Computing: The Role of Interval Methods on Scientific Computing* (edited by Ramon E. Moore), Academic Press, Boston, 99–107

Cody, W. J. (1988b), Algorithm 665: MACHAR: A subroutine to dynamically determine machine parameters, *ACM Transactions on Mathematical Software* **14**, 303–329.

Coleman, Thomas F., and Charles Van Loan (1988), *Handbook for Matrix Computations*, Society for Industrial and Applied Mathematics, Philadelphia.

Demmel, James W. (1997), *Applied Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia.

Dempster, Arthur P., and Donald B. Rubin (1983), Rounding error in regression: The appropriateness of Sheppard's corrections, *Journal of the Royal Statistical Society, Series B* **39**, 1–38.

Dodson, David S.; Roger G. Grimes; and John G. Lewis (1991), Sparse extensions to the FORTRAN basic linear algebra subprograms, *ACM Transactions on Mathematical Software* **17**, 253–263.

Dongarra, J. J.; J. R. Bunch; C. B. Moler; and G. W. Stewart (1979), *LINPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia.

Dongarra, J. J.; J. DuCroz; S. Hammarling; and I. Duff (1990), A set of level 3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software* **16**, 1–17.

Dongarra, J. J.; J. DuCroz; S. Hammarling; and R. J. Hanson (1988), An extended set of Fortran basic linear algebra subprograms, *ACM Transactions on Mathematical Software* **14**, 1–17.

Dongarra, Jack J.; Iain S. Duff; Danny C. Sorensen; and Henk A. van der Vorst (1991), *Solving Linear Systems on Vector and Shared Memory Computers*, Society for Industrial and Applied Mathematics, Philadelphia.

Dongarra, Jack; Tom Rowan; and Reed Wade (1995), Software distribution using `Xnetlib`, *ACM Transactions on Mathematical Software* **21**, 79–88.

Dongarra, Jack J., and David W. Walker (1995), Software libraries for linear algebra computations on high performance computers, *SIAM Review* **37**, 151–180. (Also published as "Libraries for linear algebra", *High Performance Computing*, edited by Gary W. Sabot, 1995, Addison-Wesley Publishing Company, Reading, Massachusetts, 93–134.)

Draper, N. R., and H. Smith (1981), *Applied Regression Analysis*, second edition, John Wiley & Sons, New York.

Duff, Iain S.; Michele Marrone; Guideppe Radicati; and Carlo Vittoli (1997), Level 3 basic linear algebra subprograms for sparse matrices: A user-level interface, *ACM Transactions on Mathematical Software* **23**, 379–401.

Durbin, J. (1960), The fitting of time series models, *International Statistical Review* **28**, 233–243.

Ericksen, Wilhelm S. (1985), Inverse pairs of matrices, *ACM Transactions on Mathematical Software* **11**, 302–304.

Escobar, Luis A., and E. Barry Moser (1993), A note on the updating of regression estimates, *The American Statistician* **47**, 192–194.

Eskow, Elizabeth, and Robert B. Schnabel (1991), Algorithm 695: Software for a new modified Cholesky factorization, *ACM Transactions on Mathematical Software* **17**, 306–312.

Etter, D. M. (1996), *Introduction to MATLAB for Engineers and Scientists*, Prentice-Hall, Englewood Cliffs, New Jersey.

Everitt, Brian S. (1994), *A Handbook of Statistical Analyses Using S-Plus*, Chapman & Hall, New York.

Forsythe, George E. (1970), Pitfalls in computation, or why a math book isn't enough, *American Mathematics Monthly* **77**, 931–955.

Forsythe, George E., and Cleve B. Moler (1967), *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Fox, Geoffrey C.; Mark A. Johnson; Gregory A. Lyzenga; Steve W. Otto; John K. Salmon; and David W. Walker (1988), *Solving Problems on Concurrent Processors. Volume I: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Francis, J. G. F. (1961a), The QR transformation, I, *The Computer Journal* **4**, 265–271.

Francis, J. G. F. (1961b), The QR transformation, II, *The Computer Journal* **4**, 332–345.

Freund, R. W.; G. H. Golub; and N. M. Nachtigal (1992), Iterative solution of linear systems, *Acta Numerica 1992*, Cambridge University Press, Cambridge, United Kingdom, 57–100.

Fuller, Wayne A. (1976), *Introduction to Statistical Time Series*, John Wiley & Sons, New York.

Fuller, Wayne A. (1987), *Measurement Error Models*, John Wiley & Sons, New York.

Gallivan, K.; M. Heath; E. Ng; J. Ortega; B. Peyton; R. Plemmons; C. Romine; A. Sameh; and R. Voigt (1990), *Parallel Algorithms for Matrix Computations*, Society for Industrial and Applied Mathematics, Philadelphia.

Gallivan, K.; R. Plemmons; and A. Sameh (1990), Parallel algorithms for dense linear algebra computations, *SIAM Review* **32**, 54–135.

Garey, M. R., and D. S. Johnson (1979), *Computers and Intractibility: A Guide to NP-Completeness*, W. H. Freeman and Co., San Francisco.

Geist, Al; Adam Beguelin; Jack Dongarra; Weicheng Jiang; Robert Manchek; and Vaidy Sunderam (1994), *PVM. Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, Massachusetts.

Gentleman, W. M. (1974), Algorithm AS 75: Basic procedures for large, sparse or weighted linear least squares problems, *Applied Statistics* **23**, 448–454.

Gill, Len, and Arthur Lewbel (1992), Testing the rank and definiteness of estimated matrices with applications to factor, state-space and ARMA models,

Journal of the American Statistical Association **87**, 766–776.

Goldberg, David (1991), What every computer scientist should know about floating-point arithmetic, *ACM Computing Surveys* **23**, 5–48.

Golub, G. H., and W. Kahan (1965), Calculating the singular values and pseudo-inverse of a matrix, *SIAM Journal of Numerical Analysis, Series B* **2**, 205–224.

Golub, Gene, and James M. Ortega (1993), *Scientific Computing. An Introduction with Parallel Computing*, Academic Press, San Diego.

Golub, G. H., and C. Reinsch (1970), Singular value decomposition and least squares solutions, *Numerische Mathematik* **14**, 403–420.

Golub, G. H., and C. F. Van Loan (1980), An analysis of the total least squares problem, *SIAM Journal of Numerical Analysis* **17**, 883–893.

Golub, Gene H., and Charles F. Van Loan (1996), *Matrix Computations*, third edition, The Johns Hopkins Press, Baltimore.

Graybill, Franklin A. (1983), *Introduction to Matrices with Applications in Statistics*, second edition, Wadsworth Publishing Company, Belmont, California.

Gregory, Robert T., and David L. Karney (1969), *A Collection of Matrices for Testing Computational Algorithms*, John Wiley & Sons, New York.

Gregory, R. T., and E. V. Krishnamurthy (1984), *Methods and Applications of Error-Free Computation*, Springer-Verlag, New York.

Grewal, Mohinder S., and Angus P. Andrews (1993), *Kalman Filtering Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey.

Griffiths, P., and I. D. Hill (Editors) (1985), *Applied Statistics Algorithms*, Ellis Horwood Limited, Chichester, United Kingdom.

Haag, J. B., and D. S. Watkins (1993), *QR*-like algorithms for the nonsymmetric eigenvalue problem, *ACM Transactions on Mathematical Software* **19**, 407–418.

Hager, W. W. (1984), Condition estimates, *SIAM Journal on Scientific and Statistical Computing* **5**, 311–316.

Hanselman, Duane, and Bruce Littlefield (1996), *Mastering MATLAB*, Prentice-Hall, Englewood Cliffs, New Jersey.

Härdle, Wolfgang (1991), *Smoothing Techniques with Implementation in S*, Springer-Verlag, New York.

Harville, David A. (1997), *Matrix Algebra from a Statistician's Point of View*, Springer-Verlag, New York.

Heath, M. T.; E. Ng; and B. W. Peyton (1991), Parallel alogrithms for sparse linear systems, *SIAM Review* **33**, 420–460.

Heiberger, Richard M. (1989), *Computation for the analysis of designed experiments*, John Wiley & Sons, New York.

Henderson, Harold V., and S. R. Searle (1979), Vec and vech operators for matrices, with some uses in Jacobians and multivariate statistics, *Canadian Journal of Statistics* **7**, 65–81.

Higham, Nicholas J. (1987), A survey of condition number estimation for triangular matrices, *SIAM Review* **29**, 575–596.

Higham, Nicholas J. (1988), FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation, *ACM Transactions on Mathematical Software* **14**, 381–386.

Higham, Nicholas J. (1990), Experience with a matrix norm estimator, *SIAM Journal on Scientific and Statistical Computing* **11**, 804–809.

Higham, Nicholas J. (1991), Algorithm 694: A collection of test matrices in Matlab, *ACM Transactions on Mathematical Software* **17**, 289–305.

Higham, Nicholas J. (1996), *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia.

Higham, Nicholas J. (1997), Stability of the diagonal pivoting method with partial pivoting, *SIAM Journal of Matrix Analysis and Applications* **18**, 52–65.

Hoerl, Arthur E., and Robert W. Kennard (1970a), Ridge regression: Biased estimation for nonorthogonal problems, *Technometrics* **12**, 55–67.

Hoerl, Arthur E., and Robert W. Kennard (1970b), Ridge regression: Applications to nonorthogonal problems, *Technometrics* **12**, 68–82.

Hong, H. P., and C. T. Pan (1992), Rank-revealing QR factorization and SVD, *Mathematics of Computation* **58**, 213–232.

IEEE (1985), *IEEE Standard for Binary Floating-point Arithmetic, Std 754-1985*, IEEE, Inc. New York.

Jansen, Paul, and Peter Weidner (1986), High-accuracy arithmetic software – some tests of the ACRITH problem-solving routines, *ACM Transactions on Mathematical Software* **12**, 62–70.

Kearfott, R. Baker (1996), INTERVAL_ARITHMETIC: A Fortran 90 module for an interval data type. *ACM Transactions on Mathematical Software* **22**, 385–392.

Kearfott, R. B.; M. Dawande; K. Du; and C. Hu (1994), Algorithm 737: INTLIB: a portable Fortran 77 interval standard-function library, *ACM Transactions on Mathematical Software* **20**, 447–459.

Keller-McNulty, Sallie, and W. J. Kennedy (1986), An error-free generalized matrix inversion and linear least squares method based on bordering, *Communications in Statistics – Simulation and Computation* **15**, 769–785.

Kendall, M. G. (1961), *A Course in the Geometry of n Dimensions*, Charles Griffin & Company Limited, London.

Kennedy, William J., and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, Inc., New York.

Kenney, C. S., and A. J. Laub (1994), Small-sample statistical condition estimates for general matrix functions, *SIAM Journal on Scientific Computing* **15**, 191–209.

Kenney, C. S.; A. J. Laub; and M. S. Reese (1998), Statistical condition estimation for linear systems, *SIAM Journal on Scientific Computing* **19**, 566–583.

Kerrigan, James F. (1993), *Migrating to Fortran 90*, O'Reilly & Associates, Inc., Sebastapol, California.

Krause, Andreas, and Melvin Olson (1997), *The Basics of S and S-Plus*, Springer-Verlag, New York.

Kulisch, U. (1983), A new arithmetic for scientific computation, *A New Approach to Scientific Computation* (edited by U. Kulisch and W. L. Miranker), Academic Press, New York, 1–26.

Kulisch, U., and W. L. Miranker (1981), *Computer Arithmetic in Theory and Practice*, Academic Press, New York.

Kulisch, U., and W. L. Miranker (Editors) (1983), *A New Approach to Scientific Computation*, Academic Press, New York.

Lawrence, Steve, and C. Lee Giles (1998), Searching the World Wide Web, *Science* **280**, 98–100.

Lawson, Charles L., and Richard J. Hanson (1974), *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, New Jersey. (Reprinted with an appendix of updated material by Society for Industrial and Applied Mathematics, Philadelphia, 1995.)

Lawson, C. L.; R. J. Hanson; D. R. Kincaid; and F. T. Krogh (1979), Basic linear algebra subprograms for Fortran usage, *ACM Transactions on Mathematical Software* **5**, 308–323.

Lesk, Michael (1997), *Practical Digital Libraries: Books, Bytes, and Bucks*, Morgan Kaufman Publishers, San Francisco.

Liem, C. B.; T. Lü; and T. M. Shih (1995), *The Splitting Extrapolation Method*, World Scientific, Singapore.

Linnainmaa, Seppo (1975), Towards accurate statistical estimation of rounding errors in floating-point computations, *BIT (Nordisk Tidskrift for Informationsbehandling)* **15** 165–173.

Longley, James W. (1967), An appraisal of least squares problems for the electronic computer from the point of view of the user, *Journal of the American Statistical Association* **62**, 819–841.

Luk, F. T., and H. Park (1989), On parallel Jacobi orderings, *SIAM Journal on Scientific and Statistical Computing* **10**, 18–26.

Magnus, Jan R., and H. Neudecker (1988), *Matrix differential calculus with applications in statistics and econometrics*, John Wiley & Sons, New York.

Maindonald, J. H. (1984), *Statistical Computation*, John Wiley & Sons, New York.

Metcalf, Michael, and John Reid (1990), *Fortran 90 Explained*, Oxford Science Publications, Oxford, United Kingdom.

Miller, Alan J. (1992), Algorithm AS 274: Least squares routines to supplement those of Gentleman, *Applied Statistics* **41**, 458–478 (Corrections, 1994, *ibid.* **43**, 678).

Miller, Alan J., and Nam-Ky Nguyen (1994), A Fedorov exchange algorithm for D-optimal design, *Applied Statistics* **43**, 669–678.

Moore, R. E. (1979), *Methods and Applications of Interval Analysis*, Society for Industrial and Applied Mathematics, Philadelphia.

Mullet, Gary M., and Tracy W. Murray (1971), A new method for examining rounding error in least-squares regression computer programs, *Journal of the American Statistical Association* **66**, 496–498.

Nguyen, Nam-Ky, and Alan J. Miller (1992), A review of some exchange algorithms for constructing D-optimal designs, *Computational Statistics and Data Analysis* **14**, 489–498.

Pissanetzky, Sergio (1984), *Sparse Matrix Technology*, Academic Press, London.

Press, William H.; Saul A. Teukolsky; William T Vetterling; and Brian P. Flannery (1996), *Numerical Recipes in Fortran 90*, Cambridge University Press, Cambridge, United Kingdom. (Also called *Fortran Numerical Recipes, Volume 2*, second edition.)

Quinn, Michael J. (1994), *Parallel Computing, Theory and Practice*, McGraw-Hill, New York.

Rice, John R. (1966), Experiments on Gram-Schmidt orthogonalization, *Mathematics of Computation* **20**, 325–328.

Rice, John R. (1983), *Numerical Methods, Software, and Analysis*, McGraw-Hill Book Company, New York.

Rogers, Gerald S. (1980), *Matrix Derivatives*, Marcel Dekker, Inc., New York.

Rust, Bert W. (1994), Perturbation bounds for linear regression problems, *Computing Science and Statistics* **26**, 528–532.

Saad, Y., and M.H. Schultz (1986), GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM Journal on Scientific and Statistical Computing* **7**, 856–869.

Schott, James R. (1996), *Matrix Analysis for Statistics*, John Wiley & Sons, New York.

Searle, S. R. (1971), *Linear Models*, John Wiley & Sons, New York.

Searle, Shayle R. (1982), *Matrix Algebra Useful for Statistics*, John Wiley & Sons, New York.

Skeel, R. D. (1980), Iterative refinement implies numerical stability for Gaussian elimination, *Mathematics of Computation* **35**, 817–832.

Smith, B. T.; J. M. Boyle; J. J. Dongarra; B. S. Garbow; Y. Ikebe; V. C. Klema; and C. B. Moler (1976), *Matrix Eigensystem Routines – EISPACK Guide*, Springer-Verlag, Berlin.

Smith, David M. (1991), Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic, *ACM Transaction on Mathematical Software* **17**, 273–283.

Spanier, Jerome, and Keith B. Oldham (1987), *An Atlas of Functions*, Hemisphere Publishing Corporation, Washington. (Also Springer-Verlag, Berlin.)

Spector, Phil (1994), *An Introduction to S and S-Plus*, Duxbury Press, Belmont, California.

Stallings, W. T., and T. L. Boullion (1972), Computation of pseudo-inverse using residue arithmetic, *SIAM Review* **14**, 152–163.

Stewart, G. W. (1973), *Introduction to Matrix Computations*, Academic Press, New York.

Stewart, G. W. (1980), The efficient generation of random orthogonal matrices with an application to condition estimators, *SIAM Journal of Numerical Analysis* **17**, 403–409.

Stewart, G. W. (1990), Stochastic perturbation theory, *SIAM Review* **32**, 579–610.

Stewart, G. W. (1996), *Afternotes on Numerical Analysis*, Society for Industrial and Applied Mathematics, Philadelphia.

Szabó, S., and R. Tanaka (1967), *Residue Arithmetic and Its Application to Computer Technology*, McGraw-Hill, New York.

Thisted, Ronald A. (1988), *Elements of Statistical Computing*, Chapman & Hall, New York.

Thompson, William J. (1997), *Atlas for Computing Mathematical Functions: An Illustrated Guide for Practitioners with Programs in C and Mathematica*, John Wiley & Sons, New York.

Thrall, Robert M., and Leonard Tornheim (1957), *Vector Spaces and Matrices*, John Wiley & Sons, New York. (Reprinted by Dover Publications, Inc., New York.)

Tierney, Luke (1990), *Lisp-Stat: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, John Wiley & Sons, New York.

Trefethen, Lloyd N., and David Blau III (1997), *Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia.

Unicode Consortium (1990), *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Unicode Consortium (1992), *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 2*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Van de Geijn, Robert (1997), *Using PLAPACK: Parallel Linear Algebra Package*, The MIT Press, Cambridge, Massachusetts.

Van Loan, Charles F. (1987), On estimating the condition of eigenvalues and eigenvectors, *Linear Algebra and Its Applications* **88**, 715–732.

Van Loan, Charles F. (1997), *Introduction to Scientific Computing: A Matrix-Vector Approach Using MATLAB*, Prentice-Hall, Englewood Cliffs, New Jersey.

Venables, W. N., and B. D. Ripley (1997), *Modern Applied Statistics with S-Plus*, second edition, Springer-Verlag, New York.

Walker, Homer F. (1988), Implementation of the GMRES method using Householder transformations, *SIAM Journal on Scientific and Statistical Computing* **9**, 152–163.

Walker, Homer F., and Lu Zhou (1994), A simpler GMRES, *Numerical Linear Algebra with Applications* **1**, 571–581.

Waterman, Michael S. (1995), *Introduction to Computational Biology*, Chapman & Hall, New York.

Watkins, David S. (1991), *Fundamentals of Matrix Computations*, John Wiley & Sons, New York.

Weisberg, Sanford (1980), *Applied Linear Regression*, John Wiley & Sons, New York.

Wilkinson, J. H. (1959), The evaluation of the zeros of ill-conditioned polynomials, *Numerische Mathematik* **1**, 150–180.

Wilkinson, J. H. (1963), *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey. (Reprinted by Dover Publications, New York, 1994).

Wilkinson, J. H. (1965), *The Algebraic Eigenvalue Problem*, Oxford University Press, New York.

Yohe, J. M. (1979), Software for interval arithmetic: A reasonably portable package, *ACM Transactions on Mathematical Software* **5**, 50–63.

Author Index

Subject Index