Software for Numerical Linear Algebra

There is a variety of computer software available to perform the operations on vectors and matrices discussed in Chapter 11. We can distinguish the software based on the kinds of applications it emphasizes, the level of the objects it works with directly, and whether or not it is interactive. Some software is designed only to perform certain functions, such as eigenanalysis, while other software provides a wide range of computations for linear algebra. Some software supports only real matrices and real associated values, such as eigenvalues. In some software systems, the basic units must be scalars, and so operations on matrices or vectors must be performed on individual elements. In these systems, higher-level functions to work directly on the arrays are often built and stored in libraries. In other software systems, the array itself is a fundamental operand. Finally, some software for linear algebra is interactive and computations are performed immediately in response to the user’s input.

There are many software systems that provide capabilities for numerical linear algebra. Some of these grew out of work at universities and government labs. Others are commercial products. These include the IMSL™ Libraries, MATLAB®, S-PLUS®, the GAUSS Mathematical and Statistical System™, IDL®, PV-Wave®, Maple®, Mathematica®, and SAS IML®. In this chapter, we briefly discuss some of these systems and give some of the salient features from the user’s point of view. We also occasionally refer to two standard software packages for linear algebra, LINPACK (Dongarra et al., 1979) and LAPACK. (Anderson et al., 2000).

The Guide to Available Mathematical Software (GAMS) is a good source of information about software. This guide is organized by types of computations. Computations for linear algebra are in Class D. The web site is

http://gams.nist.gov/serve.cgi/Class/D/

Much of the software is available through statlib or netlib (see page 505 in the Bibliography).

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 11.1 beginning on
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 12.6 on page 476.

Operations on matrices are often viewed from the narrow perspective of the numerical analyst rather than from the broader perspective of a user with a task to perform. For example, the user may seek a solution to the linear system $A x = b$. Most software to solve a linear system requires $A$ to be square and of full rank. If this is not the case, then there are three possibilities: the system has no solution, the system has multiple solutions, or the system has a unique solution. A program to solve a linear system that requires $A$ to be square and of full rank does not distinguish among these possibilities but rather always refuses to provide any solution. This can be quite annoying to a user who wants to solve a large number of systems using the same code.

**Writing Mathematics and Writing Programs**

In writing either mathematics or programs, it is generally best to think of objects at the highest level that is appropriate for the problem at hand. The details of some computational procedure may be of the form

$$
\sum_i \sum_j \sum_k a_{ki} x_{kj}.
$$

We sometimes think of the computations in this form because we have programmed them in some low-level language at some time. In some cases, it is important to look at the computations in this form, but usually it is better
to think of the computations at a higher level, say

\[ A^T X. \]  \hspace{1cm} (12.2)

The compactness of the expression is not the issue (although it certainly is more pleasant to read). The issue is that expression (12.1) leads us to think of some nested computational loops, while expression (12.2) leads us to look for more efficient computational modules, such as the BLAS, which we discuss below. In a higher-level language system such as R, the latter expression is more likely to cause us to use the system more efficiently.

12.1 Fortran and C

Fortran and C are the most commonly used procedural languages for scientific computation. The American National Standards Institute (ANSI) and its international counterpart, the International Organization for Standardization (ISO), have specified standard definitions of these languages. Whenever ANSI and ISO both have a standard for a given version of a language, the standards are the same. There are various dialects of these languages, most of which result from “extensions” provided by writers of compilers. While these extensions may make program development easier and occasionally provide modest enhancements to execution efficiency, a major effect of the extensions is to lock the user into a specific compiler. Because users usually outlive compilers, it is best to eschew the extensions and to program according to the ANSI/ISO standards. Several libraries of program modules for numerical linear algebra are available both in Fortran and in C.

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.

C++ is an object-oriented programming language built on C. The object-oriented features make it much more useful in computing with vectors and matrices or other arrays and more complicated data structures. Class libraries can be built in C++ to provide capabilities similar to those available in Fortran. There are ANSI standard versions of both C and C++.

An advantage of C is that it provides for easier communication between program units, so it is often used when larger program systems are being put together. Another advantage of C is that inexpensive compilers are readily available, and it is widely taught as a programming language in beginning courses in computer science.

Fortran has evolved over many years of use by scientists and engineers. There are two related families of Fortran languages, which we will call “Fortran 77” and “Fortran 95” or “Fortran 90 and subsequent versions”, after
the model ISO/ANSI standards. Both ANSI and ISO have specified standard
definitions of various versions of Fortran. A version called FORTRAN was de-
defined in 1977 (see ANSI, 1978). We refer to this version along with a modest
number of extensions as Fortran 77. If we meant to exclude any extensions
or modifications, we refer to it as ANSI Fortran 77. A new standard (not
a replacement standard) was adopted in 1990 by ANSI, at the insistence of
ISO. This standard language is called ANSI Fortran 90 or ISO Fortran 90 (see
ANSI, 1992). It has a number of features that extend its usefulness, especially
in numerical linear algebra. There have been a few revisions of Fortran 90 in
the past several years. There are only small differences between Fortran 90
and subsequent versions, which are called Fortran 95, Fortran 2000, and For-
tran 2003. Most of the features I discuss are in all of these versions, and
since the version I currently use is Fortran 95, I will generally just refer to
“Fortran 95”, or to “Fortran 90 and subsequent versions”.

Fortran 95 provides additional facilities for working directly with arrays.
For example, to add matrices $A$ and $B$ we can write the Fortran expression
$A + B$ (see Lemmon and Schafer, 2005; Metcalf, Reid, and Cohen, 2004; or Press
et al., 1996).

Compilers for Fortran are often more expensive and less widely available
than compilers for C/C++. An open-source compiler for Fortran 95 is avail-
able at

http://www.g95.org/

Another disadvantage of Fortran is that fewer people outside of the nu-
merical computing community know the language.

12.1.1 Programming Considerations

Both users and developers of Fortran and C software need to be aware of a
number of programming details.

Indexing Arrays

Neither Fortran 77 nor C allow 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 the earlier versions
of Fortran and in C is as array variables with indexes. Fortran handles arrays
as multiply indexed memory locations, consistent with the nature of the ob-
ject. 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 $\text{vec}(A)$. 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 $3 \times 4$ matrix $A$ is as stored in the C array $A$, the $(2, 3)$ element $A_{2,3}$ is referenced as $A[1][2]$. This disconnect between the usual mathematical representations and the C representations results from the historical development of C by computer scientists, who deal with arrays, rather than by mathematical scientists, who deal with matrices and vectors.

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 $\text{vec}(A^T)$. 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.

**Reverse Communication in Iterative Algorithms**

Sometimes within the execution of an iterative algorithm it is necessary to perform some operation outside of the basic algorithm itself. The simplest example of this is in an online algorithm, in which more data must be brought in between the operations of the online algorithm. The simplest example of this is perhaps the online computation of a correlation matrix using an algorithm similar to equations (10.7) on page 411. When the first observation is passed to the program doing the computations, that program must be told that this is the first observation (or, more generally, the first $n_1$ observations). Then, for each subsequent observation (or set of observations), the program must be told that these are intermediate observations. Finally, when the last observation (or set of observations, or even a null set of observations) is passed to the computational program, the program must be told that these are the last observations, and wrap-up computations must be performed (computing correlations from sums of squares). Between the first and last invocations of the computational program, the computational program may preserve intermediate results that are not passed back to the calling program. In this simple example, the communication is one-way, from calling routine to called routine.

In more complicated cases using an iterative algorithm, the computational routine may need more general input or auxiliary computations, and hence there may be two-way communication between the calling routine and the called routine. This is sometimes called reverse communication. An example is the repetition of a preconditioning step in a routine using a conjugate gradient method; as the computations proceed, the computational routine may detect a need for rescaling and so return to a calling routine to perform those services.
Barrett et al. (1994) and Dongarra and Eijkhout (2000) describe a variety of uses of reverse communication in software for numerical linear algebra.

**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 for similar reasons.

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 10.1). For software to achieve high levels of efficiency, computations must be organized to match the length of the vector processors as often as possible.

Another thing that affects the performance of software is the execution of loops. In the simple loop

```fortran
    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:

```fortran
  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 $7\lceil n/7 \rceil$. 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. The extra programming is definitely worthwhile for programs that are to be widely distributed and used, such as the BLAS that we discuss later.

### 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 indexing method for storing symmetric matrices, called *symmetric storage mode*, uses a linear array to store only the unique elements. Symmetric storage mode is a much more efficient and useful method of storing a symmetric matrix than would be achieved by a vech(·) operator because with symmetric storage mode, the size of the matrix affects only the elements of the vector near the end. If the number of rows and columns of the matrix is increased, the length of the vector is increased, but the elements are not rearranged. For example, the symmetric matrix

\[
\begin{bmatrix}
1 & 2 & 4 & \cdots \\
2 & 3 & 5 & \cdots \\
4 & 5 & 6 & \cdots \\
& & & \cdots
\end{bmatrix}
\]

in symmetric storage mode is represented by the array

\[ (1, 2, 3, 4, 5, 6, \cdots). \]

By comparison, the vech(·) operator yields $(1, 2, 4, \cdots, 3, 5, \cdots, 6, \cdots, \cdots)$. 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 other sparse matrices, the savings in storage can be much larger.

### 12.1.2 Fortran 95

For the scientific programmer, one of the most useful features of Fortran 95 and other versions in that family of Fortran languages is the provision of primitive constructs for vectors and matrices. Whereas all of the Fortran 77 intrinsics are scalar-valued functions, Fortran 95 provides array-valued functions. For example, if \( \text{aa} \) and \( \text{bb} \) represent matrices conformable for multiplication, the statement

\[ \text{cc} = \text{matmul}(\text{aa}, \text{bb}) \]

yields the Cayley product in \( \text{cc} \). The \text{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 95, 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 \text{reshape} function can be used to initialize matrices.

For example, a Fortran 95 statement to declare that the variable \( \text{aa} \) is to be used as a 3 \( \times \) 4 array and to allocate the necessary space is

\[ \text{real, dimension}(3,4) :: \text{aa} \]

A Fortran 95 statement to initialize \( \text{aa} \) with the matrix

\[
\begin{bmatrix}
1 & 4 & 7 & 10 \\
2 & 5 & 8 & 11 \\
3 & 6 & 9 & 12
\end{bmatrix}
\]

is

\[ \text{aa} = \text{reshape(} (/ 1., 2., 3., & \\
4., 5., 6., & \\
7., 8., 9., & \\
10.,11.,12./), & \\
(/3,4/)) \) \]
Fortran 95 has an intuitive syntax for referencing subarrays, shown in Table 12.1.

**Table 12.1. Subarrays in Fortran 95**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa(2:3,1:3)</td>
<td>the $2 \times 3$ submatrix in rows 2 and 3 and columns 1 to 3 of aa</td>
</tr>
<tr>
<td>aa(:,1:4:2)</td>
<td>refers to the submatrix with all three rows and the first and third columns of aa</td>
</tr>
<tr>
<td>aa(:,4)</td>
<td>refers to the column vector that is the fourth column of aa</td>
</tr>
</tbody>
</table>

Notice that because the indexing starts with 1 (instead of 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 95 function reference

\[
\text{matmul}(aa(1:2,2:3), bb(3:4,:))
\]

yields the Cayley product

\[
\begin{bmatrix}
4 & 7 \\
5 & 8
\end{bmatrix}
\begin{bmatrix}
3 & 7 \\
4 & 8
\end{bmatrix}
\]

(12.3)

Libraries built on Fortran 95 allow some of the basic operations of linear algebra to be implemented as operators whose operands are vectors or matrices.

Fortran 95 also contains some of the constructs, such as `forall`, that have evolved to support parallel processing.

More extensive later revisions (Fortran 2000 and subsequent versions) include such features as exception handling, interoperability with C, allocatable components, parameterized derived types, and object-oriented programming.

### 12.1.3 Matrix and Vector Classes in C++

In an object-oriented language such as C++, it is useful to define classes corresponding to matrices and vectors. Operators and/or functions corresponding to the usual operations in linear algebra can be defined so as to allow use of simple expressions to perform these operations.

A class library in C++ can be defined in such a way that the computer code corresponds more closely to mathematical code. The indexes to the arrays
can be defined to start at 1, and the double index of a matrix can be written within a single pair of parentheses. For example, in a C++ class defined for use in scientific computations, the \((10,10)\) element of the matrix \(A\) (that is, \(a_{10,10}\)) could be referenced as 

\[ aa(10,10) \]

instead of as 

\[ aa[9][9] \]

as it would be in ordinary C. Many computer engineers prefer the latter notation, however.

There are various C++ class libraries or templates for matrix and vector computations; for example, those of *Numerical Recipes* (Press et al., 2000). The Template Numerical Toolkit

http://math.nist.gov/tnt/

and the Matrix Template Library

http://www.osl.iu.edu/research/mtl/

are templates based on the design approach of the C++ Standard Template Library

http://www.sgi.com/tech/stl/

The class library in *Numerical Recipes* comes with wrapper classes for use with the Template Numerical Toolkit or the Matrix Template Library.

Use of a C++ class library for linear algebra computations may carry a computational overhead that is unacceptable for large arrays. Both the Template Numerical Toolkit and the Matrix Template Library are designed to be computationally efficient (see Siek and Lumsdaine, 2000).

12.1.4 Libraries

There are a number of libraries of Fortran and C subprograms. The libraries vary in several ways: free or with licensing costs or user fees; low-level computational modules or higher-level, more application-oriented programs; specialized or general purpose; and quality, from high to low.

**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. While the dot product is relatively simple, the details of how the computations are performed and the order in which they are performed
can have effects on both the efficiency and the accuracy. See the discussion beginning on page 396 about the order of summing a list of numbers.

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). An updated set of BLAS is described by Blackford et al. (2002).

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 10 multiplies one vector by a constant and then adds another vector \((ax + y)\). The BLAS routine axpy performs this operation. The interface is
\texttt{axpy(n, a, x, incx, y, incy)}

where

- \texttt{n} is the number of elements in each vector,
- \texttt{a} is the scalar constant,
- \texttt{x} is the input/output one-dimensional array that contains the elements of the vector \(x\),
- \texttt{incx} is the stride in the array \(x\) that defines the vector,
- \texttt{y} is the input/output one-dimensional array that contains the elements of the vector \(y\), and
- \texttt{incy} is the stride in the array \(y\) that defines the vector.

Another example, the routine \texttt{rot} to apply a Givens rotation (similar to the routine \texttt{rotm} for Fast Givens that we referred to earlier), has the interface

\texttt{rot(n, x, incx, y, incy, c, s)}

where

- \texttt{n} is the number of elements in each vector,
- \texttt{x} is the input/output one-dimensional array that contains the elements of the vector \(x\),
- \texttt{incx} is the stride in the array \(x\) that defines the vector,
- \texttt{y} is the input/output one-dimensional array that contains the elements of the vector \(y\),
- \texttt{incy} is the stride in the array \(y\) that defines the vector,
- \texttt{c} is the cosine of the rotation, and
- \texttt{s} is the sine of the rotation.

This routine is invoked after \texttt{rotg} has been called to determine the cosine and the sine of the rotation (see Exercise 12.3, page 476).

Source programs and additional information about the BLAS can be obtained at

\url{http://www.netlib.org/blas/}

There is a software suite called ATLAS (Automatically Tuned Linear Algebra Software) that provides Fortran and C interfaces to a portable BLAS binding as well as to other software for linear algebra for various processors. Information about the ATLAS software can be obtained at

\url{http://math-atlas.sourceforge.net/}
Other 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., 2000). A Fortran 95 version, LAPACK95, is described by Barker et al. (2001). Information about LAPACK is available at

http://www.netlib.org/lapack/

There is a graphical user interface to help the user navigate the LAPACK site and download LAPACK routines.

ARPACK is a collection of Fortran 77 subroutines to solve large-scale eigenvalue problems. It is designed to compute a few eigenvalues and corresponding eigenvectors of a general matrix, but it also has special abilities for large sparse or structured matrices. See Lehoucq, Sorensen, and Yang (1998) for a more complete description and for the software itself.

Two of the most widely used Fortran and C libraries are the IMSL Libraries and the Nag Library. The GNU Scientific Library (GSL) is a widely used and freely distributed C library. See Galassi et al., (2002) and the web site

http://www.gnu.org/gsl/

All of these libraries provide large numbers 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.

12.1.5 The IMSL Libraries

The IMSL libraries are available in both Fortran and C versions and in both single and double precisions. These libraries use the BLAS and other software from LAPACK.

Matrix Storage Modes

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,

$$aa(i - j + w_u + 1, j) = a_{i,j}$$

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

There are separate IMSL routines for single and double precisions. The names of the Fortran routines share a common root; the double-precision version has a D as its first character, usually just placed in front of the common root. Functions that return a floating-point number but whose mnemonic root begins with an I through an N have an A in front of the mnemonic root for the single-precision version and have a D in front of the mnemonic root for the double-precision version. Likewise, the names of the C functions share a common root. The function name is of the form $\text{imsl}_f\_\text{root}_\text{name}$ for single precision and $\text{imsl}_d\_\text{root}_\text{name}$ for double precision.

Consider the problem of solving the system of linear equations

$$x_1 + 4x_2 + 7x_3 = 10,$$
$$2x_1 + 5x_2 + 8x_3 = 11,$$
$$3x_1 + 6x_2 + 9x_3 = 12.$$

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 ATx = b is to be solved; and
X, the solution.

The difference in the two routines is whether or not they do iterative refine-
ment. A program to solve the system using LSARG (without iterative refine-
ment) is shown in Figure 12.1.

C Fortran 77 program

```fortran
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
```

Fig. 12.1. IMSL Fortran Program to Solve the System of Linear Equations

The IMSL C function to solve this problem is lin_sol_gen, which is avail-
able as float *imsl_f_linsol_gen or double *imsl_d_linsol_gen. The only
required arguments for *imsl_f_linsol_gen are:

int n, the problem size;
float a[], the coefficient matrix; and
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 ATx = 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 12.2. Note the difference between the column orientation of Fortran
and the row orientation of C.
Software for Numerical Linear Algebra

\* 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
",
            x[0], x[1], x[2]);
}

Fig. 12.2. IMSL C Program to Solve the System of Linear Equations

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 12.2 only for illustration.

12.1.6 Libraries for Parallel Processing

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. Filippone and Colajanni (2000) provide a set of parallel BLAS for sparse matrices. Their system, called PSBLAS, shares the general design of the PBLAS for dense matrices and the design of the level 3 BLAS for sparse matrices proposed by Duff et al. (1997).

A distributed memory version of LAPACK, called ScaLAPACK (see Blackford et al., 1997a), has been built on the BLACS and the PBLAS modules.

A parallel version of the ARPACK library is also available. The message-passing layers currently supported are BLACS and MPI. Parallel ARPACK (PARPACK) is provided as an extension to the current ARPACK library (Release 2.1).

Standards for message passing in a distributed-memory parallel processing environment are evolving. The MPI (message-passing interface) standard being developed primarily at Argonne National Laboratories allows for standardized message passing across languages and systems. See Gropp, Lusk, and
Skjellum (1999) for a description of the MPI system. IBM has built the Message Passing Library (MPL) in both Fortran and C, which provides message-passing kernels. PLAPACK is a package for linear algebra built on MPI (see Van de Geijn, 1997).

Trilinos is a collection of compatible software packages that support parallel linear algebra computations, solution of linear and nonlinear equations and eigensystems of equations and related capabilities. The majority of packages are written in C++ using object-oriented techniques. All packages are self-contained, with the Trilinos top layer providing a common look and feel and infrastructure.

The main Trilinos web site is

http://software.sandia.gov/trilinos/

All of these packages are available on a range of platforms, especially on high-performance computers.

General references that describe parallel computations and software for linear algebra include Nakano (2004), Quinn (2003), and Roosta (2000).

12.2 Interactive Systems for Array Manipulation

Many of the computations for linear algebra are implemented as simple operators on vectors and matrices in some interactive systems. Some of the more common interactive systems that provide for direct array manipulation are Octave or Matlab, R or 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.

Mathematical Objects and Computer Objects

Some difficult design decisions must be made when building systems that provide objects that simulate mathematical objects. One issue is how to treat scalars, vectors, and matrices when their sizes happen to coincide.

• Is a vector with one element a scalar?
• Is a $1 \times 1$ matrix a scalar?
• Is a $1 \times n$ matrix a row vector?
• Is an $n \times 1$ matrix a column vector?
• Is a column vector the same as a row vector?

While the obvious answer to all these questions is “no”, it is often convenient to design software systems as if the answer, at least to some questions some of the time, is “yes”. The answer to any such software design question always must be made in the context of the purpose and intended use (and users) of the software. The issue is not the purity of a mathematical definition. We
have already seen that most computer objects and operators do not behave exactly like the mathematical entities they simulate.

The experience of most people engaged in scientific computations over many years has shown that the convenience resulting from the software’s equivalent treatment of such different objects as a $1 \times 1$ matrix and a scalar outweighs the programming error detection that could be possible if the objects were made to behave as nearly as possible to the way the mathematical entities they simulate behave.

Consider, for example, the following arrays of numbers:

\[
A = [1 \ 2], \quad B = \begin{bmatrix} \frac{1}{2} \end{bmatrix}, \quad C = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.
\] (12.4)

If these arrays are matrices with the usual matrix algebra, then $ABC$, where juxtaposition indicates Cayley multiplication, is not a valid expression. (Under Cayley multiplication, of course, we do not need to indicate the order of the operations because the operation is associative.)

If, however, we are willing to allow mathematical objects to change types, we come up with a reasonable interpretation of $ABC$. If the $1 \times 1$ matrix $AB$ is interpreted as the scalar 5, then the expression $(AB)C$ can be interpreted as $5C$, that is, a scalar times a matrix.

There is no (reasonable) interpretation that would make the expression $A(BC)$ valid.

If $A$ is a row vector and $B$ is a column vector, it hardly makes sense to define an operation on them that would yield another vector. A vector space cannot consist of such mixtures. Under a strict interpretation of the operations, $(AB)C$ is not a valid expression.

We often think of the “transpose” of a vector (although this is not a viable concept in a vector space), and we denote a dot product in a vector space as $x^T y$. If we therefore interpret a row vector such as $A$ in (12.4) as $x^T$ for some $x$ in the vector space of which $B$ is a member, then $AB$ can be interpreted as a dot product (that is, as a scalar) and again $(AB)C$ is a valid expression.

The software systems discussed in this section treat the arrays in (12.4) as different kinds of objects when they evaluate expressions involving the arrays. The possible objects are scalars, row vectors, column vectors, and matrices, corresponding to ordinary mathematical objects, and arrays, for which there is no common corresponding mathematical object. The systems provide different subsets of these objects; some may have only one class of object (matrix would be the most general), while some distinguish all five types. Some systems enforce the mathematical properties of the corresponding objects, and some systems take a more pragmatic approach and coerce the object types to ones that allow an expression to be valid if there is an unambiguous interpretation.

In the next two sections we briefly describe the facilities for linear algebra in Matlab and R. The purpose is to give a very quick comparative introduction.
12.2 Interactive Systems for Array Manipulation

12.2.1 MATLAB and Octave

MATLAB®, or Matlab®, is a proprietary software package distributed by The Mathworks, Inc. It is built on an interactive, interpretive expression language. The package also has a graphical user interface.

Octave is a freely available package that provides essentially the same core functionality in the same language as Matlab. The graphical interfaces for Octave are more primitive than those for Matlab and do not interact as seamlessly with the operating system.

General Properties

The basic object in Matlab is a rectangular array of numbers (possibly complex). Scalars (even indices) are $1 \times 1$ matrices; equivalently, a $1 \times 1$ matrix can be treated as a scalar.

Statements in Matlab are line-oriented. A statement is assumed to end at the end of the line, unless the last three characters on the line are periods (...). 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 to the standard output device (which is likely to be the monitor).

A comment statement in Matlab begins with a percent sign, “%”.

Basic Operations with Vectors and Matrices and for Subarrays

The indexing of arrays in Matlab starts with 1.

A matrix is initialized in Matlab by listing the elements row-wise within brackets and with a semicolon marking the end of a row. (Matlab also has a reshape function similar to that of Fortran 95 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 number of rows and columns of $aa$ must be the same as the number of rows and columns of $bb$. The transpose of a vector or matrix is obtained by using a postfix operator “$^\prime$”, which is the same ASCII character as the apostrophe:
% 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
yy = a*xx'
yy = xx(1:3)*aa
cc = aa*bb

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 95, 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 12.2. Compare these with the Fortran 95 references shown in Table 12.1.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa(2:3,1:3)</td>
<td>the 2 × 3 submatrix in rows 2 and 3 and columns 1 to 3 of aa</td>
</tr>
<tr>
<td>aa(:,1:2:4)</td>
<td>the submatrix with all three rows and the first and third columns of aa</td>
</tr>
<tr>
<td>aa(:,4)</td>
<td>the column vector that is the fourth column of aa</td>
</tr>
</tbody>
</table>

The subarrays can be used directly in expressions. For example, the expression
\[ aa(1:2,2:3) \times bb(3:4,:) \]
yields the product
\[
\begin{bmatrix}
4 & 7 \\
5 & 8
\end{bmatrix}
\begin{bmatrix}
3 & 7 \\
4 & 8
\end{bmatrix}
\]
as on page 453.

**Functions of Vectors and Matrices**

Matlab has functions for many of the basic operations on vectors and matrices, some of which are shown in Table 12.3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>norm</strong></td>
<td>Matrix or vector norm. For vectors, all L_\text{p} norms are available. For matrices, the L_1, L_2, L_\infty, and Frobenius norms are available.</td>
</tr>
<tr>
<td><strong>rank</strong></td>
<td>Number of linearly independent rows or columns.</td>
</tr>
<tr>
<td><strong>det</strong></td>
<td>Determinant.</td>
</tr>
<tr>
<td><strong>trace</strong></td>
<td>Trace.</td>
</tr>
<tr>
<td><strong>cond</strong></td>
<td>Matrix condition number.</td>
</tr>
<tr>
<td><strong>null</strong></td>
<td>Null space.</td>
</tr>
<tr>
<td><strong>orth</strong></td>
<td>Orthogonalization.</td>
</tr>
<tr>
<td><strong>inv</strong></td>
<td>Matrix inverse.</td>
</tr>
<tr>
<td><strong>pinv</strong></td>
<td>Pseudoinverse.</td>
</tr>
<tr>
<td><strong>lu</strong></td>
<td>LU decomposition.</td>
</tr>
<tr>
<td><strong>qr</strong></td>
<td>QR decomposition.</td>
</tr>
<tr>
<td><strong>chol</strong></td>
<td>Cholesky factorization.</td>
</tr>
<tr>
<td><strong>svd</strong></td>
<td>Singular value decomposition.</td>
</tr>
<tr>
<td><strong>linsolve</strong></td>
<td>Solve system of linear equations.</td>
</tr>
<tr>
<td><strong>lscov</strong></td>
<td>Weighted least squares. The operator &quot;&quot; can be used for ordinary least squares.</td>
</tr>
<tr>
<td><strong>nnls</strong></td>
<td>Nonnegative least squares.</td>
</tr>
<tr>
<td><strong>eig</strong></td>
<td>Eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td><strong>poly</strong></td>
<td>Characteristic polynomial.</td>
</tr>
<tr>
<td><strong>hess</strong></td>
<td>Hessenberg form.</td>
</tr>
<tr>
<td><strong>schur</strong></td>
<td>Schur decomposition.</td>
</tr>
<tr>
<td><strong>balance</strong></td>
<td>Diagonal scaling to improve eigenvalue accuracy.</td>
</tr>
<tr>
<td><strong>expm</strong></td>
<td>Matrix exponential.</td>
</tr>
<tr>
<td><strong>logm</strong></td>
<td>Matrix logarithm.</td>
</tr>
<tr>
<td><strong>sqrtm</strong></td>
<td>Matrix square root.</td>
</tr>
<tr>
<td><strong>funm</strong></td>
<td>Evaluate general matrix function.</td>
</tr>
</tbody>
</table>

In addition to these functions, Matlab has special operators "\" and "/" for solving linear systems or for multiplying one matrix by the inverse of another. While the statement
\texttt{aa\bb}

refers to a quantity that has the same value as the quantity indicated by

\[ \text{inv}(aa) \cdot bb \]

the computations performed are different (and, hence, the values produced may be different). The second expression is evaluated by performing the two operations indicated: \texttt{aa} is inverted, and the inverse is used as the left factor in matrix or matrix/vector multiplication. The first expression, \texttt{aa\bb}, indicates that the appropriate computations to evaluate \( x \) in \( Ax = b \) should be performed to evaluate the expression. (Here, \( x \) and \( b \) may be matrices or vectors.)

Another difference between the two expressions is that \texttt{inv(aa)} requires \texttt{aa} to be square algorithmically nonsingular, whereas \texttt{aa\bb} produces a value that simulates \( A^{-1}b \).

**References**

There are a number of books on Matlab, including, for example, Hanselman and Littlefield (2004). The book by Coleman and Van Loan (1988) is not specifically on Matlab but shows how to perform matrix computations in Matlab.

**12.2.2 R and S-PLUS**

The software system called S was developed at Bell Laboratories in the mid-1970s. S is both a data analysis system and an object-oriented programming language.

S-PLUS\textsuperscript{®} is an enhancement of S, developed by StatSci, Inc. (now a part of Insightful Corporation). The enhancements include graphical interfaces with menus for common analyses, more statistical analysis functionality, and support.

There is a freely available open source system called R that provides generally the same functionality in the same language as S. This system, as well as additional information about it, is available at

[http://www.r-project.org/](http://www.r-project.org/)

There are graphical interfaces for installation and maintenance of R that interact well with the operating system. The menus for analyses provided in S-Plus are not available in R.

In the following, rather than continuing to refer to each of the systems, I will generally refer only to R, but most of the discussion applies to either of the systems. There are some functions that are available in S-Plus and not in R and some available in R and not in S-Plus.
General Properties

The most important R entity is the function. In R, all actions are “functions”, and R has an extensive set of functions (that is, verbs). Many functions are provided through packages that although not part of the core R can be easily installed.

Assignment is made by “<-” or “.”. (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 such as norm.L.p, in which the first “.” is an assignment. Using this symbol instead of <- saves exactly one unshifted keystroke!)

A comment statement in R begins with a pound sign, “#”.

R has a natural syntax and powerful functions for dealing with vectors and matrices, which are objects in the base language. R has functions for printing, but if a statement consists of just the name of an object, the object is printed to the standard output device (which is likely to be the monitor).

Basic Operations with Vectors and Matrices and for Subarrays

Indexing of arrays starts at 1, and storage is column-major. Indexes are indicated by “[ ]”; for example, xx[1] refers to the first element of the one-dimensional array xx.

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. A matrix can also be constructed by binding vectors as the columns of the matrix (the cbind function) or by binding vectors as the rows of the matrix (the rbind 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

\[
\text{aa} \%*\% \text{bb}
\]

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

\[
\text{aa} * \text{bb}
\]

indicates the Hadamard product of the matrices, where the number of rows and columns of aa must be the same as the number of rows and columns of bb. The transpose of a vector or matrix is obtained by using the function “t”:

\[
t(\text{aa})
\]

Figure 12.4 below shows R code that does the same thing as the Matlab code in Figure 12.3; that is, initialize two matrices and a vector, and then form and print their products.
# R 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

Fig. 12.4. R Code to Define and Initialize Two Matrices and a Vector and Then
Form and Print Their Product

To the extent that R distinguishes between row vectors and column vec-
tors, a vector is considered to be a column vector. In many cases, however, it
does not distinguish. For example, if

    xx <- c(1,2)
    yy <- c(1,2)

the expression xx %*% yy is the dot product; that is, xx %*% yy is the same
as t(xx) %*% yy; that is, the transpose operator is not required.

The outer product, however, requires either explicit transposition or use
of a special binary operator. The outer product is formed by xx %*% t(yy)
or by using the special outer product operator %o%; thus, xx %o% yy=xx %*% t(yy). There is also a useful function, outer, that allows more general combi-
nations of the elements of two vectors. For example, if func is a scalar function
of two scalar variables, outer(xx,yy,FUN=func) forms a matrix with the rows
corresponding to xx and the columns corresponding to yy, and whose (ij)th
element corresponds to func(xx[i],yy[j]). Strings can be used as the ar-
gument FUN; thus, outer(xx,yy,FUN="*") = xx %o% yy.

In the expressions

    yy <- aa %*% xx

and

    yy <- xx[c(1,2,3)] %*% aa

in Figure 12.4, 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 12.3 in which a distinction is made between column and row vectors.

Like many other software systems for array manipulation, R usually does not distinguish between scalars and arrays of size 1. For example, if

\[
\begin{align*}
xx &\leftarrow c(1,2) \\
yy &\leftarrow c(1,2) \\
zz &\leftarrow c(1,2,3)
\end{align*}
\]

the expression \(xx \times yy \times zz\) yields the same value as \(5 \times zz\) because the expression \(xx \times yy \times zz\) is interpreted as \((xx \times yy) \times zz\) and \((xx \times yy)\) is a scalar. The expression \(xx \times (yy \times zz)\) is invalid because \(yy\) and \(zz\) are not conformable for multiplication.

Examples of subarray references in R are shown in Table 12.4. Compare these with the Fortran 95 references shown in Table 12.1 and the Matlab references shown in Table 12.2. In R, 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 95.

\textbf{Table 12.4. Subarrays in R}

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(aa[c(2,3),c(1,3)])</td>
<td>the 2 \times 3 submatrix in rows 2 and 3 and columns 1 to 3 of (aa)</td>
</tr>
<tr>
<td>(aa[,seq(1,4,2)])</td>
<td>the submatrix with all 3 rows and the 1\textsuperscript{st} and 3\textsuperscript{rd} columns of (aa)</td>
</tr>
<tr>
<td>(aa[,4])</td>
<td>the column vector that is the 4\textsuperscript{th} column of (aa)</td>
</tr>
</tbody>
</table>

The subarrays can be used directly in expressions. For example, the expression

\[
aa[c(1,2),c(2,3)] \times bb[c(3,4),] \]

yields the product

\[
\begin{bmatrix}
4 & 7 \\
5 & 8 \\
\end{bmatrix}
\begin{bmatrix}
3 & 7 \\
4 & 8 \\
\end{bmatrix}
\]

as on page 453.

\textbf{Functions of Vectors and Matrices}

R has functions for many of the basic operations on vectors and matrices. Some of the R functions are shown in Table 12.5.
Table 12.5. Some R Functions for Vector/Matrix Computations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>norm</td>
<td>Matrix norm.</td>
</tr>
<tr>
<td>vecnorm</td>
<td>Vector $L_p$ norm.</td>
</tr>
<tr>
<td>det</td>
<td>Determinant.</td>
</tr>
<tr>
<td>rcond.M</td>
<td>Matrix condition number.</td>
</tr>
<tr>
<td>solve.M</td>
<td>Matrix inverse or pseudoinverse.</td>
</tr>
<tr>
<td>lu</td>
<td>$LU$ decomposition.</td>
</tr>
<tr>
<td>qr</td>
<td>$QR$ decomposition.</td>
</tr>
<tr>
<td>chol</td>
<td>Cholesky factorization.</td>
</tr>
<tr>
<td>svd</td>
<td>Singular value decomposition.</td>
</tr>
<tr>
<td>solve.M</td>
<td>Solve system of linear equations.</td>
</tr>
<tr>
<td>lsfit</td>
<td>Ordinary or weighted least squares.</td>
</tr>
<tr>
<td>nnls.fit</td>
<td>Nonnegative least squares.</td>
</tr>
<tr>
<td>eigen</td>
<td>Eigenvalues and eigenvectors.</td>
</tr>
</tbody>
</table>

References

Chambers (1998) provides a basic description of the S language. (John Chambers was the principal designer of S.) There are several texts that describe the use of R in statistical data analysis, such as Maindonald and Braun (2003), Venables and Ripley (2003), and Everitt and Nothorn (2006).

12.3 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 computations. We have discussed several considerations for software efficiency in previous chapters. Goedecker and Hoisie (2001) discuss some of these issues more extensively.

Parallel Processing

It is important that software for numerical linear algebra take full advantage of vector or parallel computer architecture. We discussed some of the issues on page 460. Surveys of specialized software for vector architectures and parallel processors are available in Dongarra and Walker (1995) and Dongarra et al. (2002).

ScaLAPACK, described by Blackford et al. (1997b), is a distributed memory version of LAPACK that uses the BLACS and the PBLAS modules. The computations in ScaLAPACK are organized as if performed in a “distributed linear algebra machine” (DLAM), which is constructed by interconnecting BLAS with a BLACS network. The BLAS perform the usual basic computations and the BLACS network exchanges data using primitive message-passing...
operations. The DLAM can be constructed either with or without a host process. If a host process is present, it would act like a server in receiving a user request, creating the BLACS network, distributing the data, starting the BLAS processes, and collecting the results. ScaLAPACK has routines for \( LU \), Cholesky, and \( QR \) decompositions and for computing eigenvalues of a symmetric matrix. The routines are similar to the corresponding routines in LAPACK. Even the names are similar, for example, in Fortran:

\[
\begin{array}{ll}
\text{LAPACK} & \text{ScaLAPACK} \\
dgetrf & pdgetrf \quad \text{\( LU \) factorization} \\
dpotrf & pdpotrf \quad \text{Cholesky factorization} \\
dgeqrf & pdgeqrf \quad \text{\( QR \) factorization} \\
dsyevx & pdsyevx \quad \text{eigenvalues/vectors of symmetric matrix}
\end{array}
\]

The constructs of Fortran 95 are helpful in thinking of operations in such a way that they are naturally parallelized. While the addition of arrays in Fortran 77 or C is an operation that leads to loops of sequential scalar operations, in Fortran 95 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 95 statements to add the arrays \( \text{aa} \) and \( \text{bb} \) and then to add \( \text{aa} \) and \( \text{cc} \)

\[
\begin{align*}
dd &= \text{aa} + \text{bb} \\
ee &= \text{aa} + \text{cc}
\end{align*}
\]

may be less efficient than loops because the array \( \text{aa} \) may be accessed twice.

### Clusters of Computers

The software package PVM, or Parallel Virtual Machine, which was developed at Oak Ridge National Laboratory, the University of Tennessee, and Emory University, provides a set of C functions or Fortran subroutines that allow a heterogeneous collection of Unix or Linux computers to operate smoothly as a multicomputer (see Geist et al., 1994). Likewise, the libraries built on the MPI standard provide functions that effectively build a multicomputer from a heterogeneous collection of Unix computers.

A cluster of computers is a very cost-effective method for high-performance computing. A standard technology for building a cluster of Unix or Linux computers is called Beowulf (see Gropp, Lusk, and Sterling, 2003). A system called Pooch is available for linking Apple computers into clusters (see Dauger and Decyk, 2005).

### Processing Sparse Matrices

If the matrices in large-scale problems are sparse, it is important to take advantage of that sparsity both in the storage and in all computations. We
discussed storage schemes on page 451. It is also important to preserve the sparsity during intermediate computations.


12.4 Software for Statistical Applications

Statistical applications have needs that go beyond simple linear algebra. The two most common additional requirements are for

- handling metadata and
- accommodating missing data.

Software packages designed for data analysis, such as SAS/IML and R, generally provide for metadata and missing values. Fortran/C libraries generally do not provide for metadata or for handling missing data.

Two other needs that often arise in statistical analysis but often are not dealt with adequately in available software, are the

- graceful handling of nonfull rank matrices and
- working with nonsquare matrices.

Aside from these general capabilities, of course, software packages for statistical applications, even if they are designed for some specific type of analysis, should provide the common operations such as computation of simple univariate statistics, linear regression computations, and some simple graphing capabilities.

12.5 Test Data

Testbeds for software consist of test datasets that vary in condition but have known solutions or for which there is an easy way of verifying the solution. Test data maybe fixed datasets or randomly generated datasets over some population with known and controllable properties.

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 a generates an \( n \times n \) Hilbert matrix.

Randomly generated test data can provide general information about the performance of a computational method over a range of datasets with specified
characteristics. Examples of studies using randomly generated datasets are the paper by Birkhoff and Gulati (1979) on the accuracy of computed solutions $x_c$ of the linear system $Ax = b$, where $A$ is $n \times n$ from a BMvN distribution, and the paper by Stewart (1980) using random matrices from a Haar distribution to study approximations to condition numbers (see page 169 and Exercise 4.7). As it turns out, matrices from the BMvN distribution are not sufficiently ill-conditioned often enough to be useful in studies of the accuracy of solutions of linear systems. Birkhoff and Gulati developed a procedure to construct arbitrarily ill-conditioned matrices from ones with a BMvN distribution and then used these matrices in their empirical studies.

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

\[
\begin{align*}
  a_{1j} &= x_1 \\
  a_{i1} &= x_i \\
  a_{ij} &= a_{i,j-1} + a_{i-1,j-1} \text{ for } i, j = 2, \ldots, n.
\end{align*}
\]

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

\[
\begin{align*}
  b_{in} &= (-1)^{i+k} \binom{n-1}{i-1} y_1 \\  b_{nj} &= y_{n+1-j} \\  b_{ij} &= x_1 b_{in} b_{nj} + \sum_{k=i+1}^{n} b_{k,j+1} \text{ for } i, j = 1, \ldots, n-1,
\end{align*}
\]

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

\[
x_1 = 2^m \text{ for } m \leq 0
\]

and
\[ x_i = \left( \begin{array}{c} k \\ i - 1 \end{array} \right) \text{ for } i = 2, \ldots, n \text{ and } k \geq 2, \]

in which case the $L_1$ condition number of $10 \times 10$ 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 1s on the off-diagonals. For an $n \times n$ Wilkinson matrix, the diagonal elements are

\[
\frac{n - 1}{2}, \frac{n - 3}{2}, \frac{n - 5}{2}, \ldots, \frac{n - 5}{2}, \frac{n - 3}{2}, \frac{n - 1}{2}.
\]

If $n$ is odd, the diagonal includes 0, otherwise all of the diagonal elements are positive. The $5 \times 5$ 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 are likewise, and so on. The largest pair is closest in size, and each smaller pair is less close in size.

The Matlab function \texttt{wilkinson(n)} generates an $n \times n$ Wilkinson matrix. Another test matrix available in Matlab is the Rosser test matrix, which is an $8 \times 8$ matrix with an eigenvalue of multiplicity 2 and three nearly equal eigenvalues. It is constructed by the Matlab function \texttt{rosser}.


Another set of test matrices is available through the “Matrix Market”, designed and developed by R. Boisvert, R. Pozo, and K. Remington of the U.S. National Institute of Standards and Technology with contributions by various other people. The test matrices can be accessed 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.
A set of test datasets for statistical analyses has been developed by the National Institute of Standards and Technology. This set, called “statistical reference datasets” (StRD), includes test datasets for linear regression, analysis of variance, nonlinear regression, Markov chain Monte Carlo estimation, and univariate summary statistics. It is available at

http://www.itl.nist.gov/div898/strd/

Assessing the Accuracy of a Computed Result

In real-life applications, the correct solution is not known, and this may also be the case for randomly generated test datasets. If the correct solution is not known, internal consistency tests as discussed in Section 11.2.3 may be used to assess the accuracy of the computations in a given problem.

Software Reviews

Reviews of available software play an important role in alerting the user to both good software to be used and bad software to be avoided. Software reviews also often have the salutary effect of causing the software producers to improve their products.

Exercises

12.1. Write a recursive function in Fortran, C, Octave or Matlab, R or S-Plus, or PV-Wave to multiply two square matrices using the Strassen algorithm (page 437). 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.

12.2. 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 95 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 1s, all in the second column are 2s, etc. The six programs arise from three matrices of different sizes $10,000 \times 10,000$, $100 \times 1,000,000$, and $1,000,000 \times 100$; 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:

<table>
<thead>
<tr>
<th></th>
<th>10000 x 10000</th>
<th>100 x 1000000</th>
<th>1000000 x 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran column-in-row</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>row-in-column</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C column-in-row</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>row-in-column</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

12.3. Obtain the BLAS routines `rotg` and `rot` for constructing and applying a Givens rotation. These routines exist 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 5.8 in Chapter 5,

\[
A = \begin{bmatrix}
3 & 5 & 6 \\
6 & 1 & 2 \\
8 & 6 & 7 \\
2 & 3 & 1
\end{bmatrix},
\]

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 nonfull rank matrices (for this exercise, assume that the matrix is of 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?), and others.

12.4. Using the BLAS routines `rotg` and `rot` for constructing and applying a Givens rotation and the program you wrote in Exercise 12.3, 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 249, 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 249 are performed in parallel.

12.5. Compute the two largest eigenvalues of the \(21 \times 21\) Wilkinson matrix to 15 digits.

12.6. 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.)

12.7. Consider the $3 \times 3$ symmetric Toeplitz matrix with elements $a$, $b$, and $c$; that is, the matrix that looks like this:

$$
\begin{bmatrix}
a & b & c \\
b & a & b \\
c & b & a
\end{bmatrix}.
$$

a) Invert this matrix.
b) Determine conditions for which the matrix would be singular.

12.8. Develop a class library in C++ for matrix and vector operations. Discuss carefully the issues you consider in designing the class constructors. Design them in such a way that the references

$$
\begin{align*}
xx(1) \\
YY(1,1)
\end{align*}
$$

refer to the implied mathematical entities. Design the operators “+” and “*” so that the references

$$
\begin{align*}
xx + yy \\
xx * yy
\end{align*}
$$

will determine whether $a$ and $b$ are matrices and/or vectors conformable for the implied mathematical operations and, if so, will produce the object corresponding to the implied mathematical entity represented by the expression.