Chapter 2

LINPACK — A PACKAGE FOR SOLVING LINEAR SYSTEMS

*J. J. Dongarra**
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

G. W. Stewart†
Department of Computer Science
University of Maryland
College Park, Maryland 20742

INTRODUCTION

LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal. In addition, the package computes the *QR* and singular value decompositions of rectangular matrices and applies them to least squares problems.

The software for LINPACK can be obtained from either

National Energy Software Center (NESC)
Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois 60439
Phone: 312-972-7250
Cost: Determined by NESC policy

or

IMSL
Sixth Floor, NBC Building
7500 Bellaire Boulevard
Houston, Texas 77036
Phone: 713-772-1927
Cost: $75.00 (Tape included).

Requestors in European Organization for Economic Cooperation and Development countries may obtain the software by writing to

* Work supported in part by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract W-31-109-Eng-38.
† Work supported in part by the Computer Science Section of the National Science Foundation under Contract MCS 7603297.

NEA Data Bank
B.P. No. 9 (Bat. 45)
F-91191 Gif-sur-Yvette
France
Cost: Free.

The documentation for the codes is contained in the following book:

J.J. Dongarra, J.R. Bunch, C.B. Moler, G.W. Stewart,
LINPACK Users' Guide
Society for Industrial and Applied Mathematics (1979)
Cost: $17.00.

HISTORY

In June 1974 Jim Pool, then director of the Research Section of the Applied Mathematics Division (AMD) at Argonne National Laboratory, initiated informal meetings to consider producing a package of high-quality software for the solution of linear systems and related problems. The participants included members of the AMD staff, visiting scientists, and various consultants and speakers at AMD colloquia. It was decided that such a package was needed and that a secure technological basis was available for its production.

In February 1975 the participants met at Argonne to lay the groundwork for the project and hammer out what was and was not to be included in the package. A proposal was submitted to the National Science Foundation (NSF) in August 1975. NSF agreed to fund such a project for three years beginning January 1976; the Department of Energy also provided support at Argonne.

The package was developed by four participants working from their respective institutions:

J.J. Dongarra	Argonne National Laboratory
J.R. Bunch	University of California at San Diego
C.B. Moler	University of New Mexico
G.W. Stewart	University of Maryland

Argonne served as a center for the project. The participants met there summers to coordinate their work. In addition Argonne provided administrative and technical support; it was responsible for collecting and editing the programs as well as distributing them to test sites.

In the summer of 1976 the members of the project visited Argonne for a month. They brought with them a first draft of the software and documentation. It became clear that more uniformity was needed to create a coherent package. After much discussion and agonizing, formats and conventions for the package were established. By the end of that summer an early version of the codes emerged, along with rough documentation.

The fall of 1976 and winter of 1977 saw the further development of the package. The participants worked on their respective parts at their home institutions and met once during the winter to discuss progress.

During the summer of 1977 the participants developed a set of test programs to support the codes and documentation that were to become LINPACK. This set was sent to 26 test sites in the fall of 1977. The test sites included universities, government laboratories, and private industry. In addition to running the test programs on their local computers and reporting any problems that occurred, the sites also installed the package at their computer centers and announced it to their user communities. Thus, the package received real user testing on a wide variety of systems.

As a result of the testing, in mid 1978 some changes were incorporated into the package, and a second test version was sent to the test sites. The programs were retested and timing information returned for the LINPACK routines on various systems.

By the end of 1978 the codes were sent to NESC and IMSL for distribution, and the users' guide was completed and sent to SIAM for printing. At the beginning of 1979 both the programs and the documentation were available to the public.

LINPACK AND MATRIX DECOMPOSITIONS

LINPACK is based on a *decompositional* approach to numerical linear algebra. The general idea is the following. Given a problem involving a matrix A, one factors or decomposes A into a product of simpler matrices from which the problem can easily be solved. This divides the computational problem into two parts: first the computation of an appropriate decomposition, then its use in solving the problem at hand. Since LINPACK is organized around matrix decompositions, it is appropriate to begin with a general discussion of the decompositional approach to numerical linear algebra.

Consider the problem of solving the linear system

$$Ax = b, \tag{1}$$

where A is a nonsingular matrix of order n. In older textbooks this problem is treated by writing (1) as a system of scalar equations and eliminating unknowns in such a way that the system becomes upper triangular (Gaussian elimination) or even diagonal (Gauss-Jordan elimination). This approach has the advantage that it is easy to understand and that it leads to pretty computational tableaus suitable for hand calculation. However, it has the drawback that the level of detail obscures the very broad applicability of the method.

In contrast, the decompositional approach begins with the observation that it is possible to factor A in the form

$$A = LU, \tag{2}$$

where L is a lower triangular matrix with ones on its diagonal and U is upper triangular. * The solution to (1) can then be written in the form

$$x = A^{-1}b = U^{-1}L^{-1}b = U^{-1}c,$$

where $c = L^{-1}b$. This suggests the following algorithm for solving (1):

1: *Factor A in the form* (2);

2: *Solve the system* $Lc = b$; (3)

3: *Solve the system* $Ux = c$.

Since both L and U are triangular, steps 2 and 3 of the above algorithm are easily done.

The approach to matrix computations through decompositions has turned out to be quite fruitful. Here are some of the advantages. First, the approach separates the computation into two stages: the computation of a decomposition and the use of the decomposition to solve the problem at hand. These stages are exemplified by the contrast between statement 1 and statements 2 and 3 of (3). In particular, it means that the decomposition can be used repeatedly to solve new problems. For example, if (1) must be solved for many right-hand sides b, it is necessary to factor A only once. This may represent an enormous savings, since the factorization of A is an $O(n^3)$ process, whereas steps 2 and 3 of (3) require only $O(n^2)$ operations.

Second, the approach suggests ways of avoiding the explicit computation of matrix inverses or generalized inverses. This is important because the first thing a computationally naive person thinks of when faced with a formula like $x = A^{-1}b$ is to invert and multiply; and such a procedure is always computationally expensive and numerically risky.

* This is not strictly true. It may be necessary to permute the rows of A (a process called pivoting) in order to ensure the existence of the factorization (2). In finite precision arithmetic, pivoting *must* be done to ensure numerical stability.

Third, a decomposition practically begs for new jobs to do. For example, from (2) and the fact that $det(L) = 1$, it follows that

$$det(A) = det(L)\,det(U) = det(U).$$

Since U is triangular, $det(A)$ is just the product of the diagonal elements of U. As another example, consider the problem of solving the transposed system $A^T x = b$. Since $x = A^{-T}b = (LU)^{-T}b = L^{-T}U^{-T}b$, this system may be solved by replacing statements 2 and 3 in (3) with

2′: *Solve* $U^T c = b$;

3′: *Solve* $L^T x = c$.

Note that it is not at all trivial to see how row elimination as it is usually presented can be adapted to solve transposed systems.

Fourth, the decompositional approach introduces flexibility into matrix computations. There are many decompositions, and a knowledgeable person can select the one best suited to his application.

Fifth, if one is given a decomposition of a matrix A and a simple change is made in A (e.g., the alteration of a row or column), one frequently can compute the decomposition of the altered matrix from the original decomposition at far less cost than the *ab initio* computation of the decomposition. This general idea of *updating* a decomposition has been an important theme during the past decade of numerical linear algebra research.

Finally, the decompositional approach provides theoretical simplification and unification. This is true both inside and outside of numerical analysis. For example, the realization that the Crout, Doolittle, and square root methods all compute LU decompositions enables one to recognize that they are all variants of Gaussian elimination. Outside of numerical analysis, the spectral decomposition has long been used by statisticians as a canonical form for multivariate models.

LINPACK is organized around four matrix decompositions: the LU decomposition, the (pivoted) Cholesky decomposition, the QR decomposition, and the singular value decomposition. The term LU decomposition is used here in a very general sense to mean the factorization of a square matrix into a lower triangular part and an upper triangular part, perhaps with some pivoting. These decompositions will be treated at greater length later, when the actual LINPACK subroutines are discussed. But first a digression on nomenclature and organization is necessary.

NOMENCLATURE AND CONVENTIONS

The name of a LINPACK subroutine is divided into a prefix, an infix, and a suffix as follows:

TXXYY

The prefix **T** signifies the type of arithmetic and takes the following values:

S	single precision
D	double precision
C	complex

Where it is supported, a prefix of **Z**, signifying double precision complex arithmetic, is permitted.

The infix **XX** is used in two different ways, which reflects a fundamental division in the LINPACK subroutines. The first group of codes is concerned principally with solving the system $Ax = b$ for a square matrix A, and incidentally with computing condition numbers, determinants, and inverses. Although all of these codes use variations of the LU decomposition, the user is rarely interested in the decomposition itself. However, the structural properties of the matrix, such as symmetry and bandedness, make a great deal of difference in arithmetic and storage costs. Accordingly, the infix **XX** in this part of LINPACK* is used to designate the structure of the matrix.

In the square part the infix can have the following values:

GE	General matrix — no assumptions about the structure
GB	General banded matrix
PO	Symmetric positive definite matrix
PP	Symmetric positive definite matrix in packed storage format
PB	Symmetric positive definite banded matrix
SI	Symmetric indefinite matrix
SP	Symmetric indefinite matrix in packed storage format
HI	Hermitian indefinite matrix (with prefix **C** or **Z** only)
HP	Hermitian indefinite matrix in packed storage format (with prefix **C** or **Z** only)
GT	General tridiagonal matrix
PT	Positive definite tridiagonal matrix

* Because this part deals exclusively with square matrices, it will be called the *square part.*

The second part of LINPACK is called the least squares part because one of its chief uses is to solve linear least squares problems. It is built around subroutines to compute the Cholesky decomposition, the *QR* decomposition, and the singular value decomposition. Here nothing special is assumed about the form of the matrix, except symmetry for the Cholesky decomposition; however, it is now the decomposition itself that the user is interested in. Accordingly, in the least squares part of LINPACK the infix is used to designate the decomposition. There are three options:

CH	Cholesky decomposition
QR	*QR* decomposition
SV	Singular value decomposition

The suffix **YY** specifies the task the subroutine is to perform. The possibilities are

FA*	Compute an *LU* factorization.
CO*	Compute an *LU* factorization and estimate the condition number.
DC†	Compute a decomposition.
SL	Apply the results of **FA, CO,** or **DC** to solve a problem.
DI*	Compute the determinant or inverse.
UD	Update a Cholesky decomposition.
DD	Downdate a Cholesky decomposition.
EX	Update a Cholesky decomposition after a permutation (exchange).

One small corner of LINPACK, devoted to triangular systems, is not decompositional, since a triangular matrix needs no reduction. Codes in this part are designated by an infix of **TR,** and the only two suffixes are **SL** and **CO.**

In addition to the uniform manner of treating subroutine names, there are a number of other uniformities of nomenclature in LINPACK. A square matrix is always designated by **A** (**AP** and **ABD** in the packed and banded cases), and the dimension is always **N**. Rectangular input matrices are denoted by **X**, and they are always **N** × **P**. The use of **P** as an integer is the sole deviation in the calling sequences from the Fortran implicit typing convention; it was done to keep LINPACK in conformity with standard statistical notation.

Since Fortran associates no dope vectors with arrays, it is necessary to pass the first dimension of an array to the subroutine. This number is always denoted by **LDA** or **LDX,** depending on the array name. A frequent, though unavoidable, source of errors in the use of LINPACK is the confusion of **LDA** (the leading dimension of the *array* **A**) with **N** (the order of the *matrix* **A**), which may be smaller than **LDA.**

Since many LINPACK subroutines perform more than one task, it is necessary to have a parameter to say which tasks are to be done. This parameter

* Square part only.
† Least squares part only.

is always called **JOB** in LINPACK, although the method of encoding options varies from subroutine to subroutine. (The **JOB** parameter in **SQRSL,** which has a lot to do, is of Byzantine complexity, although it is very easy to use once the trick is known.)

The status of a computation on return from a LINPACK subroutine is always signaled by **INFO.** The name was deliberately chosen to have neutral connotations, since it is not necessarily an error flag. As with **JOB**, the exact meaning of **INFO** varies from subroutine to subroutine.

THE SQUARE PART OF LINPACK

The square part of LINPACK is well illustrated by the codes with the infix **GE,** i.e., those codes dealing with a general square matrix. The basic decomposition used by the **GE** codes is of the form

$$PA = LU,$$

where L is a unit lower triangular matrix and U is upper triangular. The matrix P is a permutation matrix that represents row interchanges made to ensure numerical stability. The algorithm used to determine the interchanges is called *partial pivoting* (cf. Forsythe and Moler [1967] or Stewart [1974]). The decomposition is computed by the subroutine **SGEFA,*** which uses Gaussian elimination and overwrites A with L and U. (This is typical of the LINPACK codes; the original matrix is always overwritten by its decomposition.) Information on the interchanges is returned in **IPVT.**

There are two **GE** subroutines to manipulate the decomposition computed by **SGEFA. SGESL** solves the system $Ax = b$ or the system $A^T x = b$, as specified by the parameter **JOB.** The solution x overwrites the right-hand side b. The subroutine **SGEDI** computes the determinant and inverse of A. Because the value of a determinant can easily underflow or overflow (if A is 50×50, $det(10*A) = 10^{50} det(A)$), the determinant is coded into an array **DET** of length two in the form

$$det(A) = \mathbf{DET}\,(1)*10^{**}\,\mathbf{DET}\,(2).$$

As was indicated earlier, an explicit matrix inverse is seldom needed for most problems. Consequently, **SGEDI** should be used to compute the inverse only when there is no alternative. The inverse overwrites the LU decomposition, so that the array A cannot be used in subsequent calls to **SGESL** and **SGEDI.** It is technically possible to recover A by factoring A^{-1} **(SGEFA)** and inverting it **(SGEDI)**; but, owing to rounding errors, the matrix obtained in this way may differ from the original.

---

* For definiteness the prefix **S** will be used for all LINPACK codes, it being understood that **D, C,** and **Z** are also options.

A perennial question that arises when linear systems are solved is how accurate the computed solution is. The answer to this question is usually cast in terms of the *condition number*, $\kappa(A)$, defined by

$$\kappa(A) = ||A||||A^{-1}||, \tag{4}$$

where $||\cdot||$ is a suitable matrix norm. For example, if the system $Ax = b$ is solved by **SGEFA** and **SGESL** in t-digit decimal arithmetic and $\bar{x}$ is the computed solution, then

$$\frac{||x - \bar{x}||}{||\bar{x}||} \leqslant f(n)\kappa(A)10^{-t}, \tag{5}$$

where $f(n)$ is a slowly growing function of the order n of A [Wilkinson, 1963].

The LINPACK subroutine **SGECO,** in addition to factoring A, returns an estimate of $\kappa(A)$. Because $\kappa(A)$ can become arbitrarily large, **SGECO** actually returns the reciprocal of $\kappa(A)$ in the parameter **RCOND.** The user can inspect this number to determine the accuracy of the purported solution. However, it must be borne in mind that the interpretation of the condition number depends on how the problem has been scaled, a point that will be discussed further in the section on Numerical Properties.

Most of the remaining square part of LINPACK can be regarded as adaptations of the **GE** routines to special forms of square matrices. One of the most frequently occurring forms is the positive definite matrix, where the matrix A is symmetric ($A^T = A$) and satisfies $x^TAx > 0$ whenever $x \neq 0$. In this case the LU factorization can be written in the form

$$A = R^TR, \tag{6}$$

where R is upper triangular.* This *Cholesky factorization* is what the subroutine **SPOFA** computes. The advantage of **SPOFA** over **SGEFA** is that it requires half the storage and half the work. Specifically, since A is symmetric, only its upper half need be stored. Likewise, the upper triangular factor R in (6) can overwrite the upper half of A. The lower part of the array containing A is not referenced by **SPOFA** and hence can be used to store other information. The **PO** routines include an **SL** subroutine to solve linear systems and a **DI** subroutine to compute the determinant or the inverse.

One conventional way of storing a symmetric matrix is to pack either its lower or upper part into a linear array. The LINPACK subroutine **SPPFA** computes the Cholesky factorization of a positive definite matrix with its upper part packed in the order indicated below:

* This decomposition is often written $A = L^TL$, where L is lower triangular. The upper triangular form was chosen for its consistency with the QR decomposition.

$$\begin{array}{ccccc} 1 & 2 & 4 & 7 & 11 \\ & 3 & 5 & 8 & 12 \\ & & 6 & 9 & 13 \\ & & & 10 & 14 \\ & & & & 15 \end{array}$$

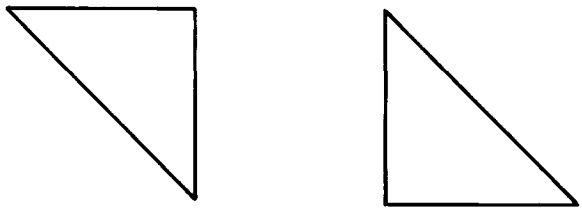
The Cholesky factor R is returned in the same order. There are corresponding **SPPCO, SPPSL,** and **SPPDI** subroutines.

When a symmetric matrix is indefinite (i.e., there are vectors x and y such that $y^TAy < 0 < x^TAx$), it has no Cholesky factorization. However, there is a permutation matrix P such that P^TAP can be decomposed stably in the form

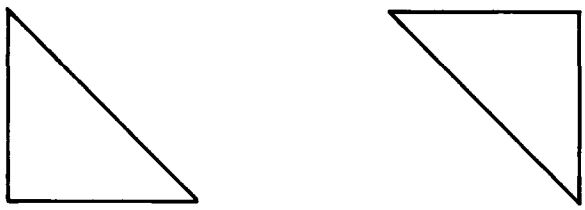
$$P^TAP = UDU^T, \tag{7}$$

where U is triangular and D is block diagonal with only 1×1 or 2×2 blocks. The subroutines **SSIFA** and **SSICO** compute this factorization, and **SSISL** and **SSIDI** apply it, as usual, to solve systems or compute determinants and inverses. **SSIDI** also computes the inertia of A. There are corresponding packed-storage subroutines.

There are two exceptional aspects of symmetric indefinite programs. First, the matrix U in (7) is *upper* triangular, giving the factorization the appearance



as opposed to



for the classical LU factorization (2). This unusual factorization is required so

that the algorithm can both work with the upper half of A (which keeps it consonant with the **PO** routines) and also remain column oriented for efficiency (more on this later).

The other aspect concerns the passage from real to complex arithmetic. A complex positive definite matrix is generally required to be Hermitian (i.e., equal to its conjugate transpose); hence there is no ambiguity in the properties of A for the **CPOYY** routines. In the indefinite case, a complex matrix may be either symmetric ($A^T = A$) or Hermitian. This point was resolved by letting the **CSIYY** routines handle complex symmetric matrices and devising a new infix **HI** for Hermitian matrices.

In many applications the nonzero elements of A are clustered around the diagonal of A. Such a matrix is called a *band matrix*. The distance m_l to the left along a row from the diagonal to the farthest off-diagonal element is the *lower band width*; the distance m_u to the right from the diagonal to the farthest element is the *upper band width*. The structure of an 8×8 matrix with lower band width one and upper band width two is illustrated below:

$$\begin{matrix} X & X & X & 0 & 0 & 0 & 0 & 0 \\ X & X & X & X & 0 & 0 & 0 & 0 \\ 0 & X & X & X & X & 0 & 0 & 0 \\ 0 & 0 & X & X & X & X & 0 & 0 \\ 0 & 0 & 0 & X & X & X & X & 0 \\ 0 & 0 & 0 & 0 & X & X & X & X \\ 0 & 0 & 0 & 0 & 0 & X & X & X \\ 0 & 0 & 0 & 0 & 0 & 0 & X & X \end{matrix}$$

By storing only the nonzero diagonals of a band matrix A, the matrix may be placed in an array of dimensions $n\times(m_l + m_u + 1)$, where n is the order of A. Thus, if m_l and m_u are fixed, the storage requirements grow only linearly with increasing n, and it is possible to represent very large systems in a modest amount of memory. The amount of work required to solve band systems also grows linearly with n.

LINPACK provides routines **SGBFA, SGBCO, SGBSL,** and **SGBDI** to manipulate band matrices. Because pivoting is required for numerical stability, the user must arrange the nonzero elements in an $n\times(2m_l + m_u + 1)$ array. Several possible schemes for storing band matrices were considered for LINPACK. The final choice was dictated by a decision to have the **GB** routines reflect the loop structure and arithmetic properties of the **GE** routines. For matrix elements within the band structure, the two sets of routines perform the same arithmetic operations in the same order. The computed solution to a band system obtained by **SGBSL** and the solution to the same system stored as a full matrix and computed by **SGESL** should agree to the last bit.

Some users of LINPACK have found that its approach to storing band matrices is complicated and unnatural. However, a simple program, listed in the *LINPACK Users' Guide,* will automatically place the elements where they belong. The subroutine **SGBDI** computes only the determinant, since the inverse of a band matrix is not itself a band matrix.

There are corresponding routines with infix **PB** for banded positive definite matrices. Since these are symmetric and require no pivoting, the storage requirement is reduced to $n\times(m_u + 1)$, and the work is correspondingly reduced. For the important case of tridiagonal matrices ($m_l = m_\mu = 1$), two special subroutines, **SGTSL** and **SPTSL,** are provided. The latter subroutine employs the "Millay" algorithm, which simultaneously factors the matrix from each end to save looping overhead.*

THE LEAST SQUARES PART OF LINPACK

Although the least squares part of LINPACK has many and varied applications, it will unify the exposition if we concentrate on the linear least squares problem. Here we are given an $n\times p$ matrix X and an n-vector y and wish to determine a p-vector b such that

$$||y - Xb|| = \min, \tag{8}$$

where $||\cdot||$ denotes the usual Euclidean norm. It can be shown that any solution of (8) must satisfy the *normal equations*

$$Ab = c, \tag{9}$$

where

$$A = X^T X, \qquad c = X^T y. \tag{10}$$

At any solution the vector Xb is the projection of y onto the column space of X, and the residual vector $r = y - Xb$ is the projection of y onto the orthogonal complement of the column space of X.

If the columns of X are independent, the normal equations (9) are positive definite. Consequently, the solution b can be obtained by first calling **SPOFA** to compute the Cholesky factorization of A and then calling **SPOSL** to solve (9). In fact, any least squares problem involving an initial set of columns of X can be solved in this manner. To see this, partition X in the form $X = (X_1\ X_2)$, where X_1 is $n\times k$, and consider the problem

$$||y - X_1\bar{b}|| = \min. \tag{11}$$

Then the normal equations assume the form

$$A_{11}\bar{b} = c_1,$$

* *My candle burns at both ends;/ It will not last the night;/ But, ah my foes, and, oh, my friends —/ It makes a lovely light.* Edna St. Vincent Millay, 1892-1950.

where $A_{11} = X_1^T X_1$. Moreover, if the Cholesky factor R of A is partitioned in the form

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \tag{12}$$

where R_{11} is $k \times k$, then

$$A_{11} = R_{11}^T R_{11}.$$

Thus, R_{11} is the Cholesky factor of A_{11}. From the point of view of LINPACK, this means that once the Cholesky factor of A has been obtained from **SPOFA,** the reduced problem (11) can be solved by calling **SPOSL** with k replacing p as the order of the matrix.

It frequently happens that the columns of the least squares matrix are linearly dependent, or nearly so. In this case it is necessary to stabilize the least squares solution in some way. The subroutine **SCHDC** provides one way by effectively moving the dependent columns to the end of X. Specifically, when **SCHDC** is applied to A, it produces a permutation matrix P and a Cholesky factorization

$$P^T A P = R^T R \tag{13}$$

that satisfies

$$r_{kk}^2 \geqslant \sum_{i=k}^{j} r_{ij}^2 \quad (j = k, k+1, \cdots, p).$$

Thus in the partition (12), if the leading element of R_{22} is small, all the elements are small, and the last columns of XP are nearly dependent on the first ones. These may then be discarded, and a least squares solution involving the initial columns of XP may be obtained by calling **SPOSL** as described above. Incidentally, the ratio $(r_{11}/r_{pp})^2$ from the pivoted Cholesky factorization is a reliable estimator of the condition number of A, and its reciprocal may be used in place of the number **RCOND** produced by **SPOCO** [Stewart, 1980].

In some applications, it is necessary to add or delete rows from a least squares fit. For the addition of a row x^T, this amounts to computing the Cholesky factorization of

$$\tilde{A} = A + xx^T,$$

where A is defined in (10). The subroutine **SCHUD** (**UD** = update) provides a way of computing the Cholesky factor $\tilde{R}$ of $\tilde{A}$ from that of A. This procedure is cheaper $[O(p^2)]$ than computing $\tilde{A}$ from A and factoring with **SPOFA** $[O(p^3)]$. **SCHUD** may also be used to solve least squares problems for which n is too large to allow X to fit into high-speed memory. The trick is to bring in X one row at a time and use repeated calls to **SCHUD** to incorporate the rows into R.

The deletion of a row amounts to computing the Cholesky decomposition of $\tilde{A} = A - xx^T$ from that of A, a process that is sometimes called *downdating*. The subroutine **SCHDD** accomplishes this; however, it is important to realize that while updating is a very stable numerical procedure, downdating is not, and the uncritical use of **SCHDD** can result in anomalous output.

In data analysis and model building it is often necessary to compare the least squares approximations corresponding to different subsets of columns of X. We have seen that this can be done if the subset in question can be moved to the beginning of X, that is, if the Cholesky decomposition of P^TAP can be computed from that of A, where P is a permutation matrix such that XP has the selected columns at the beginning. The subroutine **SCHEX** (**EX** = exchange) does this for a class of permutations from which all others can be built up.

It is a commonplace in numerical linear algebra that whenever possible one should avoid using the normal equations to solve linear least squares problems. One way of accomplishing this is by the row-wise formation of R described above. Another way is to use the LINPACK subroutines **SQRDC** and **SQRSL** to compute and manipulate the QR decomposition of X. This decomposition has the form

$$Q^TX = \begin{bmatrix} R \\ 0 \end{bmatrix},$$

where Q is an orthogonal matrix and R is upper triangular. From the orthogonality of Q it follows that

$$R^TR = XQQ^TX = X^TX,$$

which implies that the R factor in the QR decomposition of X is just the Cholesky factor of X^TX. It can further be shown that if Q is partitioned in the form

$$\begin{matrix} & p & n-p \end{matrix}$$
$$Q = (\, Q_X \quad Q_\perp),$$

then the least squares solution b satisfies

$$Rb = z,$$

where $z = Q_X^Ty$. Moreover,

$$Xb = Q_Xz, \qquad r = y - Xb = Q_\perp s,$$

where $s = Q_\perp^Ty$. Thus the least squares approximation to y and its residual vector can be obtained from the QR decomposition.

Since Q is an $n \times n$ matrix, it will be impossible to store it explicitly, even when the $n \times p$ matrix X can be stored. To circumvent this difficulty, **SQRDC** computes Q in the form

$$Q = H_1 H_2 \cdots H_p,$$

where each H_j is a Householder transformation requiring only $n-j+1$ words of storage for its representation. Thus the entire QR decomposition, consisting of R and the factored form of Q, can be stored in X and an auxiliary array of length p.

SQRSL manipulates the QR decomposition computed by **SQRDC.** Under the control of the **JOB** parameter, **SQRSL** can return the vectors $Q^T y$, Qy, b, Xb, and r. Moreover, it can return the analogous quantities corresponding to the first k columns of X, in the same way as can be done with the Cholesky decomposition; cf. (10) and the following discussion.

SQRDC also has a pivoting option, which results in the computation of the QR decomposition of the permuted matrix XP. In the absence of rounding errors, the permutation matrix P is the same as the one produced by the pivoting Cholesky decomposition of $X^T X$; cf. (13). Thus the pivoting option in **SQRDC** can be used to estimate condition numbers and detect near-degeneracies in rank. When $n < p$, the pivoting option can be used to collect a well-conditioned $n \times n$ submatrix of X, providing one exists.

The final decomposition computed by LINPACK is the singular value decomposition. As above, let X be an $n \times p$ matrix where, for definiteness, $n \geqslant p$. Then there are orthogonal matrices U and V such that

$$U^T X V = \begin{bmatrix} \Sigma \\ 0 \end{bmatrix},$$

where $\Sigma = diag(\sigma_1, \cdots, \sigma_p)$ with

$$\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_p \geqslant 0.$$

This decomposition is a supple theoretical tool that is widely used in analysis involving matrices. If Σ is known, it can be used to detect degeneracies and compute the condition number, which is σ_1/σ_p in the 2-norm. The decomposition is also required in many kinds of statistical computations — ridge regression, canonical correlations, and cross validation, to name just three.

The singular value decomposition is computed by **SSVDC,** which implements the only iterative algorithm in LINPACK. In addition to Σ, **SSVDC** will return at the user's request any of V, U, or the first p columns of U.

NUMERICAL PROPERTIES

The previous four sections addressed the question "What does LINPACK do?" Here we shall consider the related question "How well does it do it?" In

other words, how do LINPACK codes behave on actual computers? It will be convenient to divide this question into two parts and ask, first, how LINPACK codes perform in the presence of inexact arithmetic and, second, how efficient the LINPACK codes are. The first question will be treated in this section and the second in the next.

For the nonexpert there is perhaps no more bewildering subject than rounding-error analysis. The attitudes about rounding error generally range from exaggerated fears that it hopelessly contaminates everything to unjustified confidence that it is never really important. This is not the place to enter into a detailed discussion of rounding-error analysis, which has been exhaustively treated by Wilkinson [1963, 1965] and others. However, it is impossible to describe how rounding-error affects LINPACK codes without presenting some technical background.

The natural question to ask about a computed solution to a problem is how accurate it is. In numerical linear algebra, however, the question is best asked in two stages. The first stage is to ask if the solution is *stable*. The term stable is used here in a very specific sense, which can be illustrated by considering the problem of solving the system $Ax = b$. Suppose that this system is solved in t-digit decimal floating-point arithmetic to give a computed solution $\bar{x}$. Then $\bar{x}$ is said to be stable if there is a small matrix E such that

$$(A + E)\bar{x} = b, \tag{14}$$

i.e., $\bar{x}$ satisfies the slightly perturbed system (14). The matrix E is required to be small in the sense that if $\alpha = \max(|a_{ij}|)$ and $\epsilon = \max(|e_{ij}|)$, then $\epsilon/\alpha = O(10^{-t})$. Thus if eight digits are carried in the computation and A is scaled so that its largest element is one, then the largest element of E must not exceed about 10^{-8}.

The notion of stability is useful because it frequently makes the problem of accuracy moot. Suppose, for example, that a user of LINPACK must solve a linear system $A_T x_T = b$, where A_T is the true value of the matrix and x_T is the true solution. Suppose, further — as often happens in practice — that owing to measurement or computational errors the matrix actually given to the LINPACK code is

$$A = A_T + F. \tag{15}$$

Then there are three "solutions" floating around: the true solution x_T; the exact solution x of $Ax = b$; and finally the computed solution $\bar{x}$ of $Ax = b$, which satisfies (14). Now often E in (14) will be smaller than F in (15), and consequently $\bar{x}$ and x will be nearer to each other than either is to x_T. In this case, the noise in the true matrix has already altered the solution more than the subsequent errors made by the LINPACK codes. If the user is unhappy with the

answer, further computational refinements will not relieve the situation; the user must go back and get more accurate data, i.e., a better approximation to A_T.

All the decompositions computed by LINPACK are stable. For example, if $\bar{R}$ denotes the computed Cholesky factor of the positive definite matrix A, then $\bar{R}^T\bar{R}$ is equal to $A + E$ for some small matrix E. Again, the output of **SQRDC** is the numbers that would be obtained by performing exact computations on $X + E$, where E is small.

It is not to be expected that all things computed by LINPACK are done stably, although most of them are. The solutions of linear systems and linear least squares problems are stable, as are determinants. The updating routines are stable, with the exception of **SCHDD.** The most widespread unstable computation is that of the inverse.

Turning now from the question of stability to that of accuracy, we note that the inequality (5) already provides an answer in terms of the condition number $\kappa(A)$. Problems with large values of $\kappa(A)$ are said to be *ill conditioned*, and their solutions are very sensitive to perturbations in the matrix A. For the LINPACK routines, a rule of thumb is that if $\kappa(A) = 10^k$, one can expect to lose about k decimal digits of accuracy in the solution. It is important to remember that this rule accounts only for rounding errors made by LINPACK itself, and that it compares x and $\bar{x}$ defined as above; cf. (14). The relation between x and the true value x_T can be assessed only if the user is willing to provide additional information about the error F in (15).

The condition estimate provided by the LINPACK routines with suffix **CO** is generally reliable, although it can be fooled by highly contrived examples [O'Leary, 1980]. For large systems it is cheap to use, requiring only $O(n^2)$ operations as opposed to $O(n^3)$ for the factorization of A. The condition estimate obtained from the pivoted Cholesky decomposition is about as reliable [Stewart, 1980]. The singular value decomposition provides a completely reliable computation of $\kappa(A)$ in the spectral matrix norm.

We close this discussion of stability and condition estimates with a caveat. Bounds like (5) attempt to summarize the behavior of computed solutions to linear systems with a few numbers, and it is not surprising that something can be lost in this compression of the data. In particular, neither the condition number (4) nor the interpretation of the bound (5) is invariant under the scaling of the rows and the columns of A. Exactly what is the proper scaling is a complicated matter that at present is imperfectly understood. The *LINPACK Users' Guide* offers some advice, which must, however, be regarded as tentative.

The focus of the discussion up to this point has been on the effects of finite precision arithmetic on the LINPACK routines. However, something must also be said about how LINPACK deals with the finite range of the arithmetic, i.e., with underflow and overflow. The ideal in this regard would be to produce programs that succeed when both the problem and its answer are representable in the computer. Although the LINPACK programs do not attain this austere goal, they come near it by scaling strategic computations in such a way that overflows do not occur and underflows may be set to zero without affecting the results. Unfortunately this is not true everywhere, and some LINPACK programs can be made to fail by giving them data very near the underflow and overflow points. However, it is safe to say that LINPACK goes far in freeing the user from many of the difficulties associated with scaling.

EFFICIENCY

The efficiency of programs for manipulating matrices is not easy to discuss, since features that speed up a program on one system may slow it down on another. In this section we shall discuss in some detail the effects of two aspects of LINPACK on efficiency: the column orientation of the algorithms and the use of Basic Linear Algebra Subprograms (BLAS).

There was a time when one had to go out of one's way to code a matrix routine that would not run at nearly top efficiency on any system with an optimizing compiler. Owing to the proliferation of exotic computer architectures, this situation is no longer true. However, one of the new features of many modern computers — namely, hierarchical memory organization — can be exploited by some algorithmic ingenuity.

Typically, a hierarchical memory structure involves a sequence of computer memories, ranging from a small but very fast memory at the bottom to a capacious but slow memory at the top. Since a particular memory in the hierarchy (call it M) is not as big as the memory at the next higher level (M'), only part of the information in M' will be contained in M. If a reference is made to information that is in M, then it is retrieved as usual. However, if the information is not in M, then it must be retrieved from M', with a loss of time. In order to avoid repeated retrieval, information is transferred from M' to M in blocks, the supposition being that if a program references an item in a particular block, the next reference is likely to be in the same block. Programs having this property are said to have *locality of reference.*

LINPACK uses column-oriented algorithms to preserve locality of reference. By column orientation is meant that the LINPACK codes always reference arrays down columns, not across rows. This works because Fortran stores arrays

in column order. Thus, as one proceeds down a column of an array, the memory references proceed sequentially. On the other hand, as one proceeds across a row the memory references jump, the length of the jump being proportional to the length of a column. The effects of column orientation are quite dramatic; on systems with virtual or cache memories, the LINPACK codes will significantly outperform comparable codes that are not column oriented.

There are two comments to be made about column orientation. First, the textbook examples of matrix algorithms are seldom column-oriented. For example, the classical recursive algorithm for solving the lower triangular system $Lx = b$ is the following:

```
for i := 1 to n loop
    x_i := b_i;
    for j := i+1 to n loop
        x_i := x_i - l_ij x_j;
    end loop;
    x_i := x_i / l_ii;
end loop;
```

On the other hand, the column-oriented algorithm is the following:

```
x_j := b_j  (j:=1,2,...,n );
for j := 1 to n loop
    x_j := x_j / l_jj;
    for i := j+1 to n loop
        x_i := x_i - l_ij x_j;
    end loop;
end loop;
```

From this example it is seen that translating from a row-oriented algorithm to a column-oriented one is not a mechanical procedure. An even more extreme example, cited in the discussion of matrix decompositions, is the algorithm for symmetric indefinite matrices, where column orientation requires that the underlying decomposition be modified.

The second comment is that LINPACK codes should not be translated into languages such as PL/I or PASCAL, where arrays are stored by rows. Instead, row-oriented subroutines with the same calling sequences should be produced. Unfortunately, doing this for all of LINPACK is a formidable task.

Another important factor affecting the efficiency of LINPACK is the use of the BLAS [Lawson *et al.*, 1979]. This set of subprograms performs basic

operations of linear algebra, such as computing an inner product or adding a multiple of one vector to another. In LINPACK the great majority of floating-point calculations are done within the BLAS. The reasons why the BLAS were used in LINPACK will be discussed in the next section. Here we are concerned with the effects of the BLAS on the efficiency of the programs.

The BLAS affect efficiency in three ways. First, the overhead entailed in calling the BLAS reduces the efficiency of the code. This reduction is negligible for large matrices, but it can be quite significant for small matrices. The point at which it becomes unimportant varies from system to system; for square matrices it is typically between $n = 25$ and $n = 100$. If this should seem like an unacceptably large overhead, remember that on many modern systems the solution of a system of order twenty five or less is itself a negligible calculation. Nonetheless, it cannot be denied that a person whose programs depend critically on solving small matrix problems in inner loops will be better off with BLAS-less versions of the LINPACK codes. Fortunately, the BLAS can be removed from the smaller, more frequently used programs in a short editing session.

The BLAS improve the efficiency of programs when they are run on nonoptimizing compilers. This is because doubly subscripted array references in the inner loop of the algorithm are replaced by singly subscripted array references in the appropriate BLAS. The effect can be seen in matrices of quite small order, and for large orders the savings are quite large.

Finally, improved efficiency can be achieved by coding a set of BLAS to take advantage of the special features of the computers on which LINPACK is being run. For most computers, this simply means producing machine-language versions. However, the code can also take advantage of more exotic architectural features, such as vector operations.

An important conclusion to be drawn from the foregoing discussion is that on today's computers efficiency is not portable. The modifications that make a program efficient on one system may make it inefficient on another, and would-be designers of portable software must be prepared to draw flak no matter what they do.

DESIGN AND IMPLEMENTATION

One of the aims of LINPACK was to provide easy-to-use software for solving linear equations and least squares problems. Although such software existed before LINPACK, the best codes were often difficult to obtain and not readily transportable across machines. The algorithms in LINPACK are built around four standard decompositions of a matrix and are not new. The

contribution of LINPACK has been in the directions of uniformity, portability, and efficiency.

The software in LINPACK owes its form to a set of decisions made early in the course of the project. Some of the decisions were determined by the nature of the package, but others were arbitrary in the sense that other ways of proceeding would have worked equally well.

Two of the major decisions were already discussed in the section on efficiency: namely, the use of column-oriented algorithms and the use of the BLAS. Against the former, one can argue that, in some algorithms, it prevents the user from obtaining greater accuracy by accumulating inner products in double precision. We felt that the sacrifice of this feature, which is not portable, was a small price to pay for the superior performance of LINPACK on systems with hierarchical memories. The decision to use the BLAS was more problematical. It was made in the absence of complete information, and the timings collected subsequently can be used to argue pro or con, depending on one's application. If considerations of efficiency are dropped, then the BLAS are a clear plus, since they reduce the amount of code while they improve clarity.

Two other major decisions are rather controversial. The first concerns the absence of driver programs to coordinate the LINPACK subroutines. Most problems will require the use of two LINPACK subroutines, one to process the coefficient matrix and one to process a particular right-hand side. This modularity results in significant savings in computer time when there is a sequence of problems involving the same matrix but different right-hand sides. Such a situation is so common and the savings so important that no provision has been made for solving a single system with just one subroutine. Actually, this should cause few problems for the user, since there is nothing in LINPACK as complicated as the EISPACK [Smith *et al.*, 1976; Garbow *et al.*, 1977] codes, where many routines are needed to solve a given problem. Another reason for not providing driver programs is to keep the size of the package manageable. Given the way the package is structured, an extra driver for each matrix structure and decomposition would have significantly increased the number of routines.

The second controversial decision concerns error checking. No checks are made on quantities such as the order of the matrix or the leading dimension of the array. The reason is that, except in the simplest programs, the number of things to check is very large and the checking would add significantly to the length of the code. Moreover, an elaborate data structure would have to be devised to report errors, so elaborate as to be beyond most casual users. Although the LINPACK programs will inform the user if a computed decomposition is exactly singular, no attempt is made to check for near singularities, much less to recover from them. The reason is that what constitutes a near singularity

depends on how the problem has been scaled, a process that is imperfectly understood at this time. However, the user may monitor the condition number and do something if it is too large.

The naming conventions and data structures for the package have already been discussed. By and large, the decisions were easy, either because they were necessary or because it did not make much difference as long as something was decided.

Each of the LINPACK routines has a standard prologue:

Subroutine statement
Declaration of subroutine parameters
Brief description of the routine
List of the input arguments, their types and dimensions, and role in the algorithm
List of the output arguments, their types and dimensions
Date and author of the routine
External user routines needed, such as the BLAS
Fortran functions used
Declaration of variables for internal subroutine usage

The subroutine arguments are arranged in a specific order. The first three parameters are usually the matrix, **A**; the leading dimension of the matrix, **LDA**; and the order of the matrix, **N**. As has been pointed out, **LDA** and **N** should not be confused. Information about additional matrices, if any, is entered in the same way. The last two parameters are the **JOB** parameter, which tells the subroutine what to do, and the **INFO** parameter, which tells the user what the subroutine has done.

The parameters for a subroutine are declared in the order **INTEGER**, **LOGICAL**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, and **COMPLEX*16**. They are arranged in order of appearance within each classification.

After a description of the subroutine's purpose there follows a description of the parameters passed to and from each subroutine. This section is divided into two parts: information the routine needs for execution, and information generated and returned from the routine. These two parts are headed ON ENTRY and ON RETURN. The parameters listed are typed; dimensioning information is supplied, as well as a brief description of the function of the parameters. This way of specifying the input and output works well when the calling sequence is short. However, when the parameter list is long and complicated, as in the singular value routine, the convention is not so clean.

After the parameter information, the author's name appears with a date. This date shows when the routine was last updated and is very important in maintaining the package.

Next is a list of routines used by the documented routine. This list has the following order: LINPACK routines used, BLAS used, and Fortran functions used. Finally, the declarations for the internal variables are given.

Four versions of LINPACK exist, corresponding to the data types single precision, double precision, complex, and double precision complex. Because the algorithms are essentially the same for all data types, we decided to write code for the complex version only. This complex master version was then processed by an automatic programming-editing system called TAMPR [Boyle, 1980; Boyle and Dritz, 1974] to produce the other three types of programs. This automatic processing of the complex version reduced the coding and debugging time while contributing to the integrity of the package as a whole.

It was decided that the LINPACK programs should follow the canons of the structured programming school; that is, they should use only simple control structures, such as *if—then—else*. Since these structures do not exist in Fortran IV, it was necessary to simulate them. This process was greatly aided by the TAMPR system, which has the ability to recognize structure and generate appropriately indented code.*

The programs in LINPACK conform to the ANSI Fortran 66 Standard [ANSI, 1966]. We adopted Fortran because it is the language most widely used in scientific computations. Actually, the codes are restricted to a subset of the Standard that excludes **COMMON**, **EXTERNAL**, **EQUIVALENCE**, and any input or output statements. While the use of these excluded forms does not necessarily lead to unreadable, nonportable programs, it was felt that they should be avoided if possible [Smith, 1976].

The documentation of LINPACK is designed to serve both the casual user and the person who must know the technical details of the programs. The *LINPACK Users' Guide* treats, by chapter, the various types of matrix problems solved by LINPACK. In addition, appendices give related information such as the BLAS, timing data, and program listings.

Each chapter is self-contained and usually consists of seven sections:

Overview
Usage
Examples
Algorithmic Details
Programming Details

* At one point the participants considered publishing the coding conventions by which the structures were implemented, but the appearance of Fortran 77 made it pointless to do so.

Performance
Notes and References

The first three sections of each chapter contain the user-oriented material. Basic information on how to use the routines and some examples of common usage are presented. The technical material is contained in the remaining sections. These give detailed descriptions of the algorithms and their implementations, as well as operation counts and a discussion of the effects of rounding error. The final section gives historical information and references for further reading.

TESTING

Only too often software is produced with little testing and evaluation. In the development of LINPACK, considerable time and effort were spent in designing and implementing a test package. In some cases, the test programs were harder to design than the programs they tested. The chief goal was to ensure the numerical stability and the portability of the programs.

We have already observed that no program can be expected to solve ill-conditioned problems accurately. Yet ill-conditioned problems must be included in any test package, since these problems often cause an algorithm to fail catastrophically. This creates the problem of how to judge the solution of an ill-conditioned problem. The answer adopted for the LINPACK tests was to demand that the solution be stable — that is, that it be the exact solution of a slightly perturbed problem. For example, in testing the program for computing the QR factorization of a matrix X, it was required of the computed Q and R that QR reproduce X to within a modest multiple of the rounding unit. Similarly, the computed solution $\bar{x}$ of $b - Ax$ was required to satisfy

$$\frac{||b - A\bar{x}||}{||A||\,||\bar{x}||} \leqslant \epsilon,$$

where ϵ is near the rounding unit. This is equivalent to testing for stability.

In problems such as matrix inversion, which are not done stably, the LINPACK programs were required at least to produce accurate solutions to well-conditioned problems.

The programs were also tested at "edge of the machine"; that is, the programs were given problems with entries near the underflow point or near the overflow point of the machine. Our purpose was to test how close LINPACK came to being free of overflow and underflow problems. An interesting fact that emerged from these tests is that a 64-bit floating-point word with an 8-bit binary exponent is a handicap in serious computations.

An important, though technically unachievable, LINPACK goal was to produce a completely portable package. Completely portable means that *no* changes need to be made to the software to run on any system. The testing program was critical in approximating this goal. As the results came back, it was found that this or that system had unexpected, perverse features. As each of these problems was circumvented, LINPACK came nearer and nearer to complete portability.

The test programs that were used are distributed with the package. They will help spot major flaws in the installation of the LINPACK routines, although they are not designed to test the codes exhaustively. It must be admitted that the quality of the test programs is not as high as LINPACK itself, although every effort was made to ensure their portability.

Timing information was collected on a wide range of computer-compiler combinations. In the multiprogramming environment of modern computers, it is often very difficult to measure the execution time of a program reliably. Significant variations can occur, depending on the load of the machine, the amount of I/O interference, and the resolution of the timing program. The timing data were gathered by a number of people in quite different environments. Our experience with gathering timing data indicates that we can expect a variation of 10 to 15 per cent if the timings are repeated.

Execution times also vary widely from computer to computer. It was found that this variability can be reduced by dividing the raw time for a process by the raw time for another process that involves the same amount of work. For example, the time required to execute **SGESL** might be compared to the time required to compute $A*x$. By the use of this scaling technique the authors of LINPACK were able to extract meaningful results from the mass of raw timing data collected during the testing. These are reported in the *LINPACK Users' Guide.*

LINPACK was tested by various people on many different compilers and operating systems. The authors of LINPACK performed the initial testing on their own computers to ensure that the programs were working correctly. Then, the package was sent out to the test sites for further testing. To say that LINPACK *would have been impossible without the help of the test sites is not an* exaggeration. They twice nursed our test programs through their systems, made the routines available to their user communities, commented on the documentation, and reported results. Listed below are the machines used in the testing:

Amdahl 470/V6	Honeywell 6030
Burroughs 6700	IBM 360/91
CDC Cyber 175	IBM 370/158
CDC 6600	IBM 360/165
CDC 7600	IBM 370/168
CRAY-1	IBM 370/195
DEC KL-20	Itel AS/5
DEC KA-10	Univac 1110
Data General Eclipse C330	

As it turned out, few errors were revealed by the testing; on the contrary most failures were due to errors in compilers and operating systems. The test package is being used by CRAY as one of their Fortran compiler tests. While the package does not exercise all of the Fortran language, it does provide a good check of the numerical environment.

As a result of the extensive testing and our efforts in writing the codes, LINPACK comes close to being a fully portable package. There are no machine-dependent parameters or constants. The routines run, without modification, on all Fortran-based systems we know of.

CONCLUSIONS

In this final section we should like to offer some subjective conclusions on the LINPACK project and its implementation. The reader should keep in mind that these conclusions are the opinions of the two authors of this paper and do not necessarily reflect those of the other LINPACK participants.

LINPACK was not funded as a software development project; rather the National Science Foundation regarded it as research into methods for producing mathematical software. Although many useful ideas emerged from the project, it is safe to say the authors were more interested in development of the package than in software research. The reason for the curious rationale is that the National Science Foundation is constrained to help "research," which excludes software development. We feel that this constraint is detrimental to scientific endeavors in all fields and should somehow be removed. At the very least, it could be recognized that software development, by its very nature, involves a great deal of unstructured research, and this is sufficient justification for supporting such projects.

The fact that the LINPACK authors were scattered across the country did not impede the project. This is important because such an arrangement is a way of getting senior people from universities and other institutions involved in

software development. In these days of computer networks, communications are not a problem. But the arrangement does require that the participants set aside two or three weeks a year to meet at a fixed location. There is no substitute for face-to-face contact.

The major difficulty with the way the project was organized was that there was no senior member with final authority to decide hard cases. Instead, each participant was responsible for a specific part of the package, and common matters — such as nomenclature, programming conventions, and documentation — were decided by consensus. As might be expected, the process frequently degenerated into bickering, usually about matters that did not seem very important a few months later. Agreements were always reached, and we do not feel that LINPACK suffered from the compromises. But we would advise anyone embarking on a project of this sort to set up a court of last resort, especially if more than three or four people are involved.

It goes without saying that LINPACK could not have succeeded without the support of the Applied Mathematics Division at Argonne National Laboratory. Not only did they provide tangible support in the form of offices, computer time, and secretarial assistance, but they also handled the many administrative details associated with the project. Most important of all, the members of the division treated the project participants with warm hospitality.

Only recently have people become aware of how greatly the development of mathematical software is aided by appropriate computer tools. We were fortunate in having the TAMPR system available to generate code from master complex programs and format it according to its structure. We also used the PFORT verifier to check our programs for portability. Although, strictly speaking, it is not a software tool, we found the WATFIV system with its extensive error checking useful in debugging our programs. We regret that we did not have one of the mathematical typesetting systems that are now appearing; if we had, we would undoubtedly have prepared the *LINPACK Users' Guide* on it.

At the beginning of the project, we decided to get as much advice from others as possible. To let people know what we were doing, we distributed informal reports under the title "LINPACK Working Notes." We also made early versions of the programs available to those who requested them. Although we had to spend a great deal of time justifying specific decisions to people who would have done things otherwise, the valuable suggestions we got more than compensated for the trouble.

We learned not to expect that tests, however extensive, would uncover all program bugs. By the time a well-written piece of mathematical software has been run on two or three systems, most of the obvious errors have been

detected, and the remaining errors are quite subtle. For example, the shift of origin in the singular value routine is calculated incorrectly. This was not discovered during testing because it had no dramatic effect on the convergence of the algorithm and no effect at all on the stability of the result. By no means do we intend to imply that extensive testing is pointless; we have already noted that the tests helped improve the portability of LINPACK. But we found that there is considerable truth to the inverse of Murphy's law: If something *must* go wrong, it won't.

LINPACK is by no means perfect, and each of the participants has his particular regrets. The condition estimator would have been more useful if it had been a 2-norm and null vector estimator. The package should contain programs for packed triangular matrices and for updating the *QR* decomposition. We could have spent more time polishing the test drivers.

But perfection is an elusive thing. We are convinced that LINPACK is a good package and do not regret the time we spent producing it. We are reminded of Darwin's advice to the would-be world traveler:

But I have too deeply enjoyed the voyage, not to recommend any naturalist ... to start. He may feel assured, he will meet with no difficulties or dangers, excepting in rare cases, nearly so bad as he beforehand anticipates. In a moral point of view, the effect ought to be, to teach him good-humoured patience, freedom from selfishness, the habit of acting for himself, and of making the best of every occurrence. ... Travelling ought also to teach him distrust; but at the same time he will discover, how many truly kind-hearted people there are, with whom he never before had, or ever again will have any further communication, who yet are ready to offer him the most disinterested assistance.

REFERENCES

ANSI [1966]. *FORTRAN.* ANS X3.9-1966, American National Standards Institute, New York.

Boyle, J. [1980]. "Software Adaptability and Program Transformation." *Software Engineering.* Eds. W. Freeman and P. M. Lewis. Academic Press, New York, pp. 75-90.

Boyle, J., and K. W. Dritz [1974]. "An Automated Programming System to Aid the Development of Quality Mathematical Software." *IFIP Proceedings,* North-Holland, Amsterdam, pp. 542-546.

Dongarra, J. J., J. R. Bunch, C. B. Moler, and G. W. Stewart [1979]. *LINPACK Users' Guide.* SIAM Publications, Philadelphia.

Forsythe and Moler [1967]. *Computer Solution of Linear Algebra Systems.* Prentice-Hall, Englewood Cliffs, New Jersey.

Garbow, B. S., *et al.* [1977]. *Matrix Eigensystem Routines — EISPACK Guide Extension.* Lecture Notes in Computer Science, Vol. 51. Springer-Verlag, Berlin.

Lawson, C., R. Hanson, D. Kincaid, and F. Krogh [1979]. "Basic linear algebra subprograms for Fortran usage." *ACM Trans. on Math. Soft.,* 5:308-323.

O'Leary [1980]. "Estimating matrix condition numbers." *SIAM Scientific and Statistical Computing,* 2:205-209.

Smith, B. T. [1976]. *Fortran Poisoning and Antidotes.* In Lecture Notes in Computer Science, Vol. 57. *Portability of Numerical Software.* Ed. W. Cowell, Springer-Verlag, Berlin.

Smith, B. T. *et al.* [1976]. *Matrix Eigensystem Routines — EISPACK Guide.* Lecture Notes in Computer Science, Vol. 6. 2nd ed. Springer-Verlag, Berlin.

Stewart, G. W. [1974]. *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 Numer. Anal.,* 17:403-409.

Wilkinson, J. H. [1963]. *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, New Jersey.

Wilkinson, J. H. [1965]. *The Algebraic Eigenvalue Problem.* Oxford University Press, London.