NAG Toolbox for MATLAB

Chapter Introduction

F07 – Linear Equations (LAPACK)

Contents

1 Scope of the Chapter

This chapter provides functions for the solution of systems of simultaneous linear equations, and associated computations. It provides functions for

- matrix factorizations;
- solution of linear equations;
- estimating matrix condition numbers;
- computing error bounds for the solution of linear equations;
- matrix inversion;
- computing scaling factors to equilibrate a matrix.

Functions are provided for both *real* and *complex* data.

For a general introduction to the solution of systems of linear equations, you should turn first to the F04 Chapter Introduction. The decision trees, at the end of the F04 Chapter Introduction, direct you to the most appropriate functions in Chapters F04 or f07 for solving your particular problem. In particular, Chapters F04 and f07 contain *Black Box* (or *driver*) functions which enable some standard types of problem to be solved by a call to a single function. Where possible, functions in Chapter F04 call Chapter f07 functions to perform the necessary computational tasks.

There are two types of driver functions in this chapter: *simple drivers* which just return the solution to the linear equations; and *expert drivers* which also return condition and error estimates and, in many cases, also allow equilibration. The simple drivers for real matrices have names of the form f07_a and for complex matrices have names of the form f07_n. The expert drivers for real matrices have names of the form f07_b and for complex matrices have names of the form f07_p.

The functions in this chapter (F07) handle only *dense* and *band* matrices (not matrices with more specialized structures, or general sparse matrices).

The functions in this chapter have all been derived from the LAPACK project (see Anderson *et al.* 1999). They have been designed to be efficient on a wide range of high-performance computers, without compromising efficiency on conventional serial machines.

2 Background to the Problems

This section is only a brief introduction to the numerical solution of systems of linear equations. Consult a standard textbook, for example Golub and Van Loan 1996 for a more thorough discussion.

2.1 Notation

We use the standard notation for a system of simultaneous linear equations:

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

where A is the *coefficient matrix*, b is the *right-hand side*, and x is the *solution*. A is assumed to be a square matrix of order n.

If there are several right-hand sides, we write

$$AX = B \tag{2}$$

where the columns of B are the individual right-hand sides, and the columns of X are the corresponding solutions.

We also use the following notation, both here and in the function documents:

$\hat{x}$	a *computed* solution to $Ax = b$, (which usually differs from the exact solution x because of round-off error)
$r = b - A\hat{x}$	the *residual* corresponding to the computed solution $\hat{x}$
$\|x\|_\infty = \max_i \|x_i\|$	the ∞-norm of the vector x
$\|x\|_1 = \sum_{j=1}^{n} \|x_j\|$	the 1-norm of the vector x

$\|A\|_\infty = \max_i \sum_j \|a_{ij}\|$	the ∞-norm of the matrix A
$\|A\|_1 = \max_j \sum_{i=1}^{n} \|a_{ij}\|$	the 1-norm of the matrix A
$\|x\|$	the vector with elements $\|x_i\|$
$\|A\|$	the matrix with elements $\|a_{ij}\|$

Inequalities of the form $|A| \leq |B|$ are interpreted component-wise, that is $|a_{ij}| \leq |b_{ij}|$ for all i,j.

2.2 Matrix Factorizations

If A is upper or lower triangular, $Ax = b$ can be solved by a straightforward process of backward or forward substitution.

Otherwise, the solution is obtained after first factorizing A, as follows.

General matrices (*LU* factorization with partial pivoting)

$$A = PLU$$

where P is a permutation matrix, L is lower-triangular with diagonal elements equal to 1, and U is upper-triangular; the permutation matrix P (which represents row interchanges) is needed to ensure numerical stability.

Symmetric positive-definite matrices (Cholesky factorization)

$$A = U^{\mathrm{T}}U \qquad \text{or} \qquad A = LL^{\mathrm{T}}$$

where U is upper triangular and L is lower triangular.

Symmetric indefinite matrices (Bunch–Kaufman factorization)

$$A = PUDU^{\mathrm{T}}P^{\mathrm{T}} \qquad \text{or} \qquad A = PLDL^{\mathrm{T}}P^{\mathrm{T}}$$

where P is a permutation matrix, U is upper triangular, L is lower triangular, and D is a block diagonal matrix with diagonal blocks of order 1 or 2; U and L have diagonal elements equal to 1, and have 2 by 2 unit matrices on the diagonal corresponding to the 2 by 2 blocks of D. The permutation matrix P (which represents symmetric row-and-column interchanges) and the 2 by 2 blocks in D are needed to ensure numerical stability. If A is in fact positive-definite, no interchanges are needed and the factorization reduces to $A = UDU^{\mathrm{T}}$ or $A = LDL^{\mathrm{T}}$ with diagonal D, which is simply a variant form of the Cholesky factorization.

2.3 Solution of Systems of Equations

Given one of the above matrix factorizations, it is straightforward to compute a solution to $Ax = b$ by solving two subproblems, as shown below, first for y and then for x. Each subproblem consists essentially of solving a triangular system of equations by forward or backward substitution; the permutation matrix P and the block diagonal matrix D introduce only a little extra complication:

General matrices (*LU* factorization)

$$\begin{aligned} Ly &= P^{\mathrm{T}}b \\ Ux &= y \end{aligned}$$

Symmetric positive-definite matrices (Cholesky factorization)

$$\begin{aligned} U^{\mathrm{T}}y &= b \\ Ux &= y \end{aligned} \qquad \text{or} \qquad \begin{aligned} Ly &= b \\ L^{\mathrm{T}}x &= y \end{aligned}$$

Symmetric indefinite matrices (Bunch–Kaufman factorization)

$$\begin{aligned} PUDy &= b \\ U^{\mathrm{T}}P^{\mathrm{T}}x &= y \end{aligned} \qquad \text{or} \qquad \begin{aligned} PLDy &= b \\ L^{\mathrm{T}}P^{\mathrm{T}}x &= y \end{aligned}$$

2.4 Sensitivity and Error Analysis

2.4.1 Normwise error bounds

Frequently, in practical problems the data A and b are not known exactly, and it is then important to understand how uncertainties or perturbations in the data can affect the solution.

If x is the exact solution to $Ax = b$, and $x + \delta x$ is the exact solution to a perturbed problem $(A + \delta A)(x + \delta x) = (b + \delta b)$, then

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A)\left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|}\right) + \cdots (\text{second-order terms})$$

where $\kappa(A)$ is the *condition number* of A defined by

$$\kappa(A) = \|A\|.\left\|A^{-1}\right\|. \tag{3}$$

In other words, relative errors in A or b may be amplified in x by a factor $\kappa(A)$. Section 2.4.2 discusses how to compute or estimate $\kappa(A)$.

Similar considerations apply when we study the effects of *rounding errors* introduced by computation in finite precision. The effects of rounding errors can be shown to be equivalent to perturbations in the original data, such that $\frac{\|\delta A\|}{\|A\|}$ and $\frac{\|\delta b\|}{\|b\|}$ are usually at most $p(n)\epsilon$, where ϵ is the ***machine precision*** and $p(n)$ is an increasing function of n which is seldom larger than $10n$ (although in theory it can be as large as 2^{n-1}).

In other words, the computed solution $\hat{x}$ is the exact solution of a linear system $(A + \delta A)\hat{x} = b + \delta b$ which is close to the original system in a normwise sense.

2.4.2 Estimating condition numbers

The previous section has emphasised the usefulness of the quantity $\kappa(A)$ in understanding the sensitivity of the solution of $Ax = b$. To compute the value of $\kappa(A)$ from equation (3) is more expensive than solving $Ax = b$ in the first place. Hence it is standard practice to *estimate* $\kappa(A)$, in either the 1-norm or the ∞-norm, by a method which only requires $O\left(n^2\right)$ additional operations, assuming that a suitable factorization of A is available.

The method used in this chapter is Higham's modification of Hager's method (see Higham 1988). It yields an estimate which is never larger than the true value, but which seldom falls short by more than a factor of 3 (although artificial examples can be constructed where it is much smaller). This is acceptable since it is the order of magnitude of $\kappa(A)$ which is important rather than its precise value.

Because $\kappa(A)$ is infinite if A is singular, the functions in this chapter actually return the *reciprocal* of $\kappa(A)$.

2.4.3 Scaling and Equilibration

The condition of a matrix and hence the accuracy of the computed solution, may be improved by scaling; thus if D_1 and D_2 are diagonal matrices with positive diagonal elements, then

$$B = D_1 A D_2$$

is the scaled matrix. A general matrix is said to be *equilibrated* if it is scaled so that the lengths of its rows and columns have approximately equal magnitude. Similarly a general matrix is said to be *row-equilibrated (column-equilibrated)* if it is scaled so that the lengths of its rows (columns) have approximately equal magnitude. Note that row scaling can affect the choice of pivot when partial pivoting is used in the factorization of A.

A symmetric or Hermitian positive-definite matrix is said to be equilibrated if the diagonal elements are all approximately equal to unity.

For further information on scaling and equilibration see Section 3.5.2 of Golub and Van Loan 1996, Section 7.2, 7.3 and 9.8 of Higham 1988 and Section 5 of Chapter 4 of Wilkinson 1965.

Functions are provided to return the scaling factors that equilibrate a matrix for general, general band, symmetric and Hermitian positive-definite and symmetric and Hermitian positive-definite band matrices.

2.4.4 Componentwise error bounds

A disadvantage of normwise error bounds is that they do not reflect any special structure in the data A and b – that is, a pattern of elements which are known to be zero – and the bounds are dominated by the largest elements in the data.

Componentwise error bounds overcome these limitations. Instead of the normwise relative error, we can bound the relative error in *each component* of A and b:

$$\max_{ijk}\left(\frac{|\delta a_{ij}|}{|a_{ij}|},\frac{|\delta b_k|}{|b_k|}\right)\leq\omega$$

where the *component-wise backward error bound* ω is given by

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

Functions are provided in this chapter which compute ω, and also compute a *forward error bound* which is sometimes much sharper than the normwise bound given earlier:

$$\frac{\|x-\hat{x}\|_\infty}{\|x\|_\infty}\leq\frac{\left\||A^{-1}|.|r|\right\|_\infty}{\|x\|_\infty}.$$

Care is taken when computing this bound to allow for rounding errors in computing r. The norm $\left\||A^{-1}|.|r|\right\|_\infty$ is estimated cheaply (without computing A^{-1}) by a modification of the method used to estimate $\kappa(A)$.

2.4.5 Iterative refinement of the solution

If $\hat{x}$ is an approximate computed solution to $Ax=b$, and r is the corresponding residual, then a procedure for *iterative refinement* of $\hat{x}$ can be defined as follows, starting with $x_0=\hat{x}$:

for $i=0,1,\ldots,$ until convergence

compute $r_i=b-Ax_i$
solve $Ad_i=r_i$
compute $x_{i+1}=x_i+d_i$

In Chapter F04, functions are provided which perform this procedure using ***additional precision*** to compute r, and are thus able to reduce the *forward error* to the level of ***machine precision***.

The functions in this chapter do *not* use ***additional precision*** to compute r, and cannot guarantee a small forward error, but can guarantee a *small backward error* (except in rare cases when A is very ill-conditioned, or when A and x are sparse in such a way that $|A|.|x|$ has a zero or very small component). The iterations continue until the backward error has been reduced as much as possible; usually only one iteration is needed, and at most five iterations are allowed.

2.5 Matrix Inversion

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do *not* attempt to solve $Ax=b$ by first computing A^{-1} and then forming the matrix-vector product $x=A^{-1}b$; the procedure described in Section 2.3 is more efficient and more accurate.

However, functions are provided for the rare occasions when an inverse is needed, using one of the factorizations described in Section 2.2.

2.6 Packed Storage

Functions which handle symmetric matrices are usually designed so that they use either the upper or lower triangle of the matrix; it is not necessary to store the whole matrix. If the upper or lower triangle is stored conventionally in the upper or lower triangle of a two-dimensional array, the remaining elements of the array can be used to store other useful data. However, that is not always convenient, and if it is important to economize on storage, the upper or lower triangle can be stored in a one-dimensional array of length $n(n+1)/2$; in other words, the storage is almost halved.

This storage format is referred to as *packed storage;* it is described in Section 3.2.2. It may also be used for triangular matrices.

Functions designed for packed storage perform the same number of arithmetic operations as functions which use conventional storage, but they are usually less efficient, especially on high-performance computers, so there is then a trade-off between storage and efficiency.

2.7 Band and Tridiagonal Matrices

A *band* matrix is one whose nonzero elements are confined to a relatively small number of subdiagonals or superdiagonals on either side of the main diagonal. A *tridiagonal* matrix is a special case of a band matrix with just one subdiagonal and one superdiagonal. Algorithms can take advantage of bandedness to reduce the amount of work and storage required. The storage scheme used for band matrices is described in Section 3.2.3.

The LU factorization for general matrices, and the Cholesky factorization for symmetric and Hermitian positive-definite matrices both preserve bandedness. Hence functions are provided which take advantage of the band structure when solving systems of linear equations.

The Cholesky factorization preserves bandedness in a very precise sense: the factor U or L has the same number of superdiagonals or subdiagonals as the original matrix. In the LU factorization, the row-interchanges modify the band structure: if A has k_l subdiagonals and k_u superdiagonals, then L is not a band matrix but still has at most k_l nonzero elements below the diagonal in each column; and U has at most $k_l + k_u$ superdiagonals.

The Bunch–Kaufman factorization does not preserve bandedness, because of the need for symmetric row-and-column permutations; hence no functions are provided for symmetric indefinite band matrices.

The inverse of a band matrix does not in general have a band structure, so no functions are provided for computing inverses of band matrices.

2.8 Block Partitioned Algorithms

Many of the functions in this chapter use what is termed a *block partitioned algorithm*. This means that at each major step of the algorithm a *block* of rows or columns is updated, and most of the computation is performed by matrix-matrix operations on these blocks. The matrix-matrix operations are performed by calls to the Level 3 BLAS (see Chapter F06), which are the key to achieving high performance on many modern computers. See Golub and Van Loan 1996 or Anderson *et al.* 1999 for more about block partitioned algorithms.

The performance of a block partitioned algorithm varies to some extent with the ***blocksize*** – that is, the number of rows or columns per block. This is a machine-dependent parameter, which is set to a suitable value when the library is implemented on each range of machines. You do not normally need to be aware of what value is being used. Different block sizes may be used for different functions. Values in the range 16 to 64 are typical.

On some machines there may be no advantage from using a block partitioned algorithm, and then the functions use an *unblocked* algorithm (effectively a blocksize of 1), relying solely on calls to the Level 2 BLAS (see Chapter F06 again).

The only situation in which you need some awareness of the block size is when it affects the amount of workspace to be supplied to a particular function. This is discussed in Section missing entity f07recomm_43.

3 Recommendations on Choice and Use of Available Functions

3.1 Available Functions

Tables 1 to 8 in Section 3.4 show the functions which are provided for performing different computations on different types of matrices. Tables 1 to 4 show functions for real matrices; Tables 5 to 8 show functions for complex matrices. Each entry in the table gives the NAG function name and the LAPACK double precision name .

Functions are provided for the following types of matrix:

general

general band

general tridiagonal

symmetric or Hermitian positive-definite

symmetric or Hermitian positive-definite (packed storage)

symmetric or Hermitian positive-definite band

symmetric or Hermitian positive-definite tridiagonal

symmetric or Hermitian indefinite

symmetric or Hermitian indefinite (packed storage)

triangular

triangular (packed storage)

triangular band

For each of the above types of matrix (except where indicated), functions are provided to perform the following computations:

(a) solve a system of linear equations (driver functions);

(b) solve a system of linear equations with condition and error estimation (expert drivers);

(c) (except for triangular matrices) factorize the matrix (see Section 2.2);

(d) solve a system of linear equations, using the factorization (see Section 2.3);

(e) estimate the condition number of the matrix, using the factorization (see Section 2.4.2); these functions also require the norm of the original matrix (except when the matrix is triangular) which may be computed by a function in Chapter F06;

(f) refine the solution and compute forward and backward error bounds (see Sections 2.4.4 and 2.4.5); these functions require the original matrix and right-hand side, as well as the factorization returned from (a) and the solution returned from (b);

(g) (except for band and tridiagonal matrices) invert the matrix, using the factorization (see Section 2.5);

(h) (except for tridiagonal, symmetric indefinite and triangular matrices) compute scale factors to equilibrate the matrix (see Section 2.4.3).

Thus, to solve a particular problem, it is usually only necessary to call a single driver function, but alternatively two or more functions may be called in succession. This is illustrated in the example programs in the function documents.

3.2 Matrix Storage Schemes

In this chapter the following different storage schemes are used for matrices:

– conventional storage in a two-dimensional array;

– packed storage for symmetric, Hermitian or triangular matrices;

– band storage for band matrices.

These storage schemes are compatible with those used in Chapter F06 (especially in the BLAS) and Chapter F08, but different schemes for packed or band storage are used in a few older functions in Chapters F01, F02, F03 and F04.

In the examples below, $*$ indicates an array element which need not be set and is not referenced by the functions. .

3.2.1 Conventional storage

The default scheme for storing matrices is the obvious one: a matrix A is stored in a two-dimensional array **a**, with matrix element a_{ij} stored in array element $\mathbf{a}(i,j)$.

If a matrix is **triangular** (upper or lower, as specified by the parameter **uplo**), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set. Such elements are indicated by * in the examples below.

For example, when $n = 4$:

uplo	**Triangular matrix A**	**Storage in array A**
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{matrix}$
'L'	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$

Functions which handle **symmetric** or **Hermitian** matrices allow for either the upper or lower triangle of the matrix (as specified by **uplo**) to be stored in the corresponding elements of the array; the remaining elements of the array need not be set.

For example, when $n = 4$:

uplo	**Hermitian matrix A**	**Storage in array a**
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} \\ \bar{a}_{14} & \bar{a}_{24} & \bar{a}_{34} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{matrix}$
'L'	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & \bar{a}_{41} \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$

3.2.2 Packed storage

Symmetric, Hermitian or triangular matrices may be stored more compactly, if the relevant triangle (again as specified by **uplo**) is packed by columns in a one-dimensional array. In Chapters f07 and F08, arrays which hold matrices in packed storage have names ending in P. The storage of matrix elements a_{ij} in the packed array **ap** is as follows:

if **uplo** = 'U', a_{ij} is stored in $\mathbf{ap}(i + j(j-1)/2)$ for $i \leq j$;
if **uplo** = 'L', a_{ij} is stored in $\mathbf{ap}(i + (2n-j)(j-1)/2)$ for $j \leq i$.

For example:

uplo	**Triangle of matrix A**	**Packed storage in array ap**
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$a_{11}\,\underbrace{a_{12}a_{22}}\,\underbrace{a_{13}a_{23}a_{33}}\,\underbrace{a_{14}a_{24}a_{34}a_{44}}$

'L'	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}a_{21}a_{31}a_{41}}\,\underbrace{a_{22}a_{32}a_{42}}\,\underbrace{a_{33}a_{43}}\,a_{44}$

Note that for real symmetric matrices, packing the upper triangle by columns is equivalent to packing the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the upper triangle by rows. (For complex Hermitian matrices, the only difference is that the off-diagonal elements are conjugated.)

3.2.3 Band storage

A band matrix with k_l subdiagonals and k_u superdiagonals may be stored compactly in a two-dimensional array with $k_l + k_u + 1$ rows and n columns. Columns of the matrix are stored in corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. This storage scheme should be used in practice only if k_l, $k_u \ll n$, although the functions in Chapters f07 and F08 work correctly for all values of k_l and k_u. In Chapters f07 and F08 arrays which hold matrices in band storage have names ending in B.

To be precise, elements of matrix elements a_{ij} are stored as follows:

a_{ij} is stored in $\mathbf{ab}(k_u + 1 + i - j, j)$ for $\max(1, j - k_u) \le i \le \min(n, j + k_l)$.

For example, when $n = 5$, $k_l = 2$ and $k_u = 1$:

Band matrix A	**Band storage in array ab**
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

The elements marked $*$ in the upper left and lower right corners of the array **ab** need not be set, and are not referenced by the functions.

Note: when a general band matrix is supplied for LU factorization, space must be allowed to store an additional k_l superdiagonals, generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with $k_l + k_u$ superdiagonals.

Triangular band matrices are stored in the same format, with either $k_l = 0$ if upper triangular, or $k_u = 0$ if lower triangular.

For symmetric or Hermitian band matrices with k subdiagonals or superdiagonals, only the upper or lower triangle (as specified by **uplo**) need be stored:

if **uplo** = 'U', a_{ij} is stored in $\mathbf{ab}(k + 1 + i - j, j)$ for $\max(1, j - k) \le i \le j$;
if **uplo** = 'L', a_{ij} is stored in $\mathbf{ab}(1 + i - j, j)$ for $j \le i \le \min(n, j + k)$.

For example, when $n = 5$ and $k = 2$:

uplo	**Hermitian band matrix *A***	**Band storage in array a**
'U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} & \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} & a_{35} \\ & \bar{a}_{24} & \bar{a}_{34} & a_{44} & a_{45} \\ & & \bar{a}_{35} & \bar{a}_{45} & a_{55} \end{pmatrix}$	$\begin{matrix} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \end{matrix}$

'L'	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & & \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} & \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} & \bar{a}_{53} \\ & a_{42} & a_{43} & a_{44} & \bar{a}_{54} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

Note that different storage schemes for band matrices are used by some functions in Chapters F01, F02, F03 and F04.

3.2.4 Unit triangular matrices

Some functions in this chapter have an option to handle unit triangular matrices (that is, triangular matrices with diagonal elements $= 1$). This option is specified by an argument **diag**. If **diag** = 'U' (Unit triangular), the diagonal elements of the matrix need not be stored, and the corresponding array elements are not referenced by the functions. The storage scheme for the rest of the matrix (whether conventional, packed or band) remains unchanged.

3.2.5 Real diagonal elements of complex matrices

Complex Hermitian matrices have diagonal elements that are by definition purely real. In addition, complex triangular matrices which arise in Cholesky factorization are defined by the algorithm to have real diagonal elements.

If such matrices are supplied as input to functions in this chapter, the imaginary parts of the diagonal elements are not referenced, but are assumed to be zero. If such matrices are returned as output by the functions, the computed imaginary parts are explicitly set to zero.

3.3 Parameter Conventions

3.3.1 Option parameters

Most functions in this chapter have one or more option parameters, of type string. The descriptions in Section 5 of the function documents refer only to upper-case values (for example **uplo** = 'U' or 'L'); however, in every case, the corresponding lower-case characters may be supplied (with the same meaning). Any other value is illegal.

A longer character string can be passed as the actual parameter, making the calling program more readable, but only the first character is significant. For example:

```
[b, info] = f07ae('Transpose', a, ipiv, b);
```

3.3.2 Problem dimensions

It is permissible for the problem dimensions (for example, **m** in f07ad, **n** or **nrhs_p** in f07ae) to be passed as zero, in which case the computation (or part of it) is skipped. Negative dimensions are regarded as an error.

3.4 Tables of Available Computational Functions

3.4.1 Real matrices

Type of matrix and storage scheme	general	general band	general tridiagonal
factorize	f07ad	f07bd	f07cd
solve	f07ae	f07be	f07ce
scaling factors	f07af	f07bf	
condition number	f07ag	f07bg	f07cg
error estimate	f07ah	f07bh	f07ch

invert	f07aj		

Table 1
Functions for real general matrices

Type of matrix and storage scheme	symmetric positive-definite	symmetric positive-definite (packed storage)	symmetric positive-definite band	symmetric positive-definite tridiagonal
factorize	f07fd	f07gd	f07hd	f07jd
solve	f07fe	f07ge	f07he	f07je
scaling factors	f07ff	f07gf	f07hf	
condition number	f07fg	f07gg	f07hg	f07jg
error estimate	f07fh	f07gh	f07hh	f07jh
invert	f07fj	f07gj		

Table 2
Functions for real symmetric positive-definite matrices

Type of matrix and storage scheme	symmetric indefinite	symmetric indefinite (packed storage)
factorize	f07md	f07pd
solve	f07me	f07pe
condition number	f07mg	f07pg
error estimate	f07mh	f07ph
invert	f07mj	f07pj

Table 3
Functions for real symmetric indefinite matrices

Type of matrix and storage scheme	triangular	triangular (packed storage)	triangular band
solve	f07te	f07ue	f07ve
condition number	f07tg	f07ug	f07vg
error estimate	f07th	f07uh	f07vh
invert	f07tj	f07uj	

Table 4
Functions for real triangular matrices

3.4.2 Complex matrices

Type of matrix and storage scheme	general	general band	general tridiagonal
factorize	f07ar	f07br	f07cr
solve	f07as	f07bs	f07cs
scaling factors	f07at	f07bt	
condition number	f07au	f07bu	f07cu
error estimate	f07av	f07bv	f07cv
invert	f07aw		

Table 5
Functions for complex general matrices

Type of matrix and storage scheme	Hermitian positive-definite	Hermitian positive-definite (packed storage)	Hermitian positive-definite band	Hermitian positive-definite tridiagonal
factorize	f07fr	f07gr	f07hr	f07jr
solve	f07fs	f07gs	f07hs	f07js
scaling factors	f07ft	f07gt		
condition number	f07fu	f07gu	f07hu	f07ju
error estimate	f07fv	f07gv	f07hv	f07jv
invert	f07fw	f07gw		

Table 6
Functions for complex Hermitian positive-definite matrices

Type of matrix and storage scheme	Hermitian indefinite	symmetric indefinite (packed storage)	Hermitian indefinite band	symmetric indefinite tridiagonal
factorize	f07mr	f07nr	f07pr	f07qr
solve	f07ms	f07ns	f07ps	f07qs
condition number	f07mu	f07nu	f07pu	f07qu
error estimate	f07mv	f07nv	f07pv	f07qv
invert	f07mw	f07nw	f07pw	f07qw

Table 7
Functions for complex Hermitian and symmetric indefinite matrices

Type of matrix and storage scheme	triangular	triangular (packed storage)	triangular band
solve	f07ts	f07us	f07vs
condition number	f07tu	f07uu	f07vu
error estimate	f07tv	f07uv	f07vv
invert	f07tw	f07uw	

Table 8
Functions for complex triangular matrices

4 Index

5 References

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J J, Du Croz J J, Greenbaum A, Hammarling S, McKenney A and Sorensen D 1999 *LAPACK Users' Guide* (3rd Edition) SIAM, Philadelphia

Golub G H and Van Loan C F 1996 *Matrix Computations* (3rd Edition) Johns Hopkins University Press, Baltimore

Higham N J 1988 Algorithm 674: Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation *ACM Trans. Math. Software* **14** 381–396

Wilkinson J H 1965 *The Algebraic Eigenvalue Problem* Oxford University Press, Oxford