NAG C Library 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.

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_AF (D__SV) and for complex matrices have names of the form F07_NF (Z__SV). The expert drivers for real matrices have names of the form F07_BF (D__SVX) and for complex matrices have names of the form F07_PF (Z__SVX).

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

$$\|A\|_\infty = \max_i \sum_j |a_{ij}| \qquad \text{the } \infty\text{-norm of the matrix } A$$

$$\|A\|_1 = \max_j \sum_{i=1}^n |a_{ij}| \qquad \text{the 1-norm of the matrix } A$$

$|x|$ the vector with elements $|x_i|$

$|A|$ the matrix with elements $|a_{ij}|$

Inequalities of the form $|A| \le |B|$ are interpreted component-wise, that is $|a_{ij}| \le |b_{ij}|$ 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)

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

where U is upper triangular and L is lower triangular.

Symmetric indefinite matrices (Bunch–Kaufman factorization)

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

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)

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

Symmetric indefinite matrices (Bunch–Kaufman factorization)

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

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

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right) + \cdots \text{(2nd-order terms)}$$

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

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

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 $O(n^2)$ 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 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 $\kappa(A)$.

2.4.4 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 3.3.2. 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 non-zero 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 3.3.3.

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 non-zero 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 f16), 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 argument, 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 f16 again).

3   Recommendations on Choice and Use of Available Functions

3.1   Available Functions

Tables 1 and 2 in Section 3.5 show the functions which are provided for performing different computations on different types of matrices. Table 1 shows functions for real matrices; Table 2 shows functions for complex matrices. Each entry in the table gives the NAG function name and the LAPACK double precision name (see Section 3.2).

Functions are provided for the following types of matrix:

> general
>
> general band
>
> 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

tridiagonal

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 f16;

(f) refine the solution and compute forward and backward error bounds (see Sections 2.4.3 and 2.4.4); 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).

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   NAG Names and LAPACK Names

As well as the NAG function name (beginning f07-), Tables 1 and 2 show the LAPACK function names in double precision.

The functions may be called either by their NAG short names or by their NAG long names. The NAG long names for a function is simply the LAPACK name (in lower case) prepended by nag_, for example, nag_dpotrf is the long name for f07fdc.

References to Chapter f07 functions in the manual normally include the LAPACK double precision names, for example, nag_dgetrf (f07adc).

The LAPACK function names follow a simple scheme (which is similar to that used for the BLAS in Chapter f16). Each name has the structure **XYYZZZ**, where the components have the following meanings:

– the initial letter **X** indicates the data type (real or complex) and precision:

　S – real, single precision

　D – real, double precision

　C – complex, single precision

　Z – complex, double precision

– the 2nd and 3rd letters **YY** indicate the type of the matrix A (and in some cases its storage scheme):

　GE – general

　GB – general band

　PO – symmetric or Hermitian positive-definite

　PP – symmetric or Hermitian positive-definite (packed storage)

　PB – symmetric or Hermitian positive-definite band

　SY – symmetric indefinite

　SP – symmetric indefinite (packed storage)

　HE – (complex) Hermitian indefinite

    HP – (complex) Hermitian indefinite (packed storage)

    GT – general tridiagonal

    PT – symmetric or Hermitian positive-definite tridiagonal

    TR – triangular

    TP – triangular (packed storage)

    TB – triangular band

– the last 3 letters **ZZZ** indicate the computation performed:

    TRF – triangular factorization

    TRS – solution of linear equations, using the factorization

    CON – estimate condition number

    RFS – refine solution and compute error bounds

    TRI – compute inverse, using the factorization

Thus the function SGETRF performs a triangular factorization of a real general matrix in a single precision implementation; the corresponding function in a double precision implementation is DGETRF.

3.3 Matrix Storage Schemes

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

    – conventional storage;

    – packed storage for symmetric, Hermitian or triangular matrices;

    – band storage for band matrices.

These storage schemes are compatible with those used in Chapter f16 (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. The examples illustrate only the relevant part of the arrays; array arguments may of course have additional rows or columns, according to the usual rules for passing array arguments in Fortran 77.

3.3.1 Conventional storage

Matrices may be stored column-wise or row-wise as described in Section 2.2.1.4 of the Essential Introduction: a matrix A is stored in a one-dimensional array **a**, with matrix element $a_{i,j}$ stored column-wise in array element $\mathbf{a}[(j-1) \times \mathbf{pda} + i - 1]$ or row-wise in array element $\mathbf{a}[(i-1) \times \mathbf{pda} + j - 1]$ where **pda** is the principle dimension of the array (i.e., the stride separating row or column elements of the matrix respectively). Most functions in this chapter contain the **order** argument which can be set to **Nag_ColMajor** for column-wise storage or **Nag_RowMajor** for row-wise storage of matrices. Where groups of functions are intended to be used together, the value of the **order** argument passed must be consistent throughout.

If a matrix is **triangular** (upper or lower, as specified by the argument **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 $n = 3$:

order	uplo	Triangular matrix A	Storage in array a
Nag_ColMajor	**Nag_Upper**	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ & a_{22} & a_{23} \\ & & a_{33} \end{pmatrix}$	$a_{11} * * a_{12} a_{22} * a_{13} a_{23} a_{33}$
Nag_RowMajor	**Nag_Upper**	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ & a_{22} & a_{23} \\ & & a_{33} \end{pmatrix}$	$a_{11} a_{12} a_{13} * a_{22} a_{23} * * a_{33}$
Nag_ColMajor	**Nag_Lower**	$\begin{pmatrix} a_{11} & & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11} a_{21} a_{31} * a_{22} a_{32} * * a_{33}$
Nag_RowMajor	**Nag_Lower**	$\begin{pmatrix} a_{11} & & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11} * * a_{21} a_{22} * a_{31} a_{32} a_{33}$

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 $n = 3$:

order	uplo	Hermitian matrix A	Storage in array a
Nag_ColMajor	**Nag_Upper**	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ \bar{a}_{12} & a_{22} & a_{23} \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} \end{pmatrix}$	$a_{11} * * a_{12} a_{22} * a_{13} a_{23} a_{33}$
Nag_RowMajor	**Nag_Upper**	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ \bar{a}_{12} & a_{22} & a_{23} \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} \end{pmatrix}$	$a_{11} a_{12} a_{13} * a_{22} a_{23} * * a_{33}$
Nag_ColMajor	**Nag_Lower**	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} \\ a_{21} & a_{22} & \bar{a}_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11} a_{21} a_{31} * a_{22} a_{32} * * a_{33}$
Nag_RowMajor	**Nag_Lower**	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} \\ a_{21} & a_{22} & \bar{a}_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11} * * a_{21} a_{22} * a_{31} a_{32} a_{33}$

3.3.2 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 or rows 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** = **Nag_Upper** then

if **order** = **Nag_ColMajor**, a_{ij} is stored in **ap**$[(i-1)+j(j-1)/2]$ for $i \leq j$;
if **order** = **Nag_RowMajor**, a_{ij} is stored in **ap**$[(j-1)+(2n-i)(i-1)/2]$ for $i \leq j$.

if **uplo** = **Nag_Lower** then

if **order** = **Nag_ColMajor**, a_{ij} is stored in **ap**$[(i-1)+(2n-j)(j-1)/2]$ for $j \leq i$;
if **order** = **Nag_RowMajor**, a_{ij} is stored in **ap**$[(j-1)+i(i-1)/2]$ for $j \leq i$.

For example:

order	uplo	Triangle of matrix A	Packed storage in array ap
Nag_ColMajor	**Nag_Upper**	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ & a_{22} & a_{23} \\ & & a_{33} \end{pmatrix}$	$a_{11}\underbrace{a_{12}a_{22}}\underbrace{a_{13}a_{23}a_{33}}$
Nag_RowMajor	**Nag_Upper**	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ & a_{22} & a_{23} \\ & & a_{33} \end{pmatrix}$	$\underbrace{a_{11}a_{12}a_{13}}\underbrace{a_{22}a_{23}}a_{33}$
Nag_ColMajor	**Nag_Lower**	$\begin{pmatrix} a_{11} & & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$\underbrace{a_{11}a_{21}a_{31}}\underbrace{a_{22}a_{32}}a_{33}$
Nag_RowMajor	**Nag_Lower**	$\begin{pmatrix} a_{11} & & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11}\underbrace{a_{21}a_{22}}\underbrace{a_{31}a_{32}a_{33}}$

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.)

3.3.3 Band storage

A band matrix with k_l subdiagonals and k_u superdiagonals may be stored compactly in a notional two-dimensional array with $k_l + k_u + 1$ rows and n columns if stored column-wise or n rows and $k_l + k_u + 1$ columns if stored row-wise. In column-major order, elements of a column of the matrix are stored contiguously in the array, and elements of the diagonals of the matrix are stored with constant stride (i.e., in a row of the two-dimensional array). In row-major order, elements of a row of the matrix are stored contiguously in the array, and elements of a diagonal of the matrix are stored with constant stride (i.e., in a column of the two-dimensional array). These storage schemes should only be used in practice 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:

if **order** = **Nag_ColMajor**, a_{ij} is stored in **ab**$[(j-1) \times \mathbf{pdab} + k_u + i - j]$;
if **order** = **Nag_RowMajor**, a_{ij} is stored in **ab**$[(i-1) \times \mathbf{pdab} + k_l + j - i]$,

where **pdab** $\geq k_l + k_u + 1$ is the stride between diagonal elements and where $\max(1, i - k_l) \leq j \leq \min(n, i + k_u)$.

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

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

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. In this example, if **order** = **Nag_ColMajor** and **pdab** takes the minimum value of 4, then **ab**$[0]$ need not be set, **ab**$[1] = a_{11}$, **ab**$[2] = a_{21}, \ldots,$ **ab**$[17] = a_{55}$. On the other hand, if **order** = **Nag_RowMajor** (**pdab** = 4), then **ab**$[0]$ and **ab**$[1]$ need not be set, **ab**$[2] = a_{11}$, **ab**$[3] = a_{12}, \ldots,$ **ab**$[18] = a_{55}$.

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; it also means that the principal dimension has the constraint **pdab** $\geq 2k_l + k_u + 1$.

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** = **Nag_Upper** then
>
> > if **order** = **Nag_ColMajor**, a_{ij} is stored in **ab**$[(j - 1) \times$ **pdab** $+ k + i - j]$;
> > if **order** = **Nag_RowMajor**, a_{ij} is stored in **ab**$[(i - 1) \times$ **pdab** $+ j - i]$.
>
> for $\max(1, j - k) \leq i \leq j$;
>
> if **uplo** = **Nag_Lower** then
>
> > if **order** = **Nag_ColMajor**, a_{ij} is stored in **ab**$[(j - 1) \times$ **pdab** $+ i - j]$;
> > if **order** = **Nag_RowMajor**, a_{ij} is stored in **ab**$[(i - 1) \times$ **pdab** $+ k + j - i]$.
>
> for $j \leq i \leq \min(n, j + k)$,

where **pdab** $\geq k + 1$ is the stride separating diagonal matrix elements in the array **ab**.

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

uplo	Hermitian band matrix A	Band storage in array ab	
		order = **Nag_ColMajor**	order = **Nag_RowMajor**
Nag_Upper	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} & \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} & a_{35} \\ & \bar{a}_{24} & \bar{a}_{34} & a_{44} & a_{45} \\ & & \bar{a}_{35} & \bar{a}_{45} & a_{55} \end{pmatrix}$	$\begin{array}{lllll} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \end{array}$	$\begin{array}{lll} a_{11} & a_{12} & a_{13} \\ a_{22} & a_{23} & a_{24} \\ a_{33} & a_{34} & a_{35} \\ a_{44} & a_{45} & * \\ a_{55} & * & * \end{array}$
Nag_Lower	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & & \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} & \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} & \bar{a}_{53} \\ & a_{42} & a_{43} & a_{44} & \bar{a}_{54} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{array}{lllll} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{array}$	$\begin{array}{lll} * & * & a_{11} \\ * & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} \\ a_{42} & a_{43} & a_{44} \\ a_{53} & a_{54} & a_{55} \end{array}$

Note that different storage schemes for band matrices are used by some functions in Chapters f01, f02, f03 and f04. In the above example, if **order** = **Nag_ColMajor** and **pdab** = 3, then for **uplo** = **Nag_Upper**, $\mathbf{ab}[2] = a_{11}, \mathbf{ab}[4] = a_{12}, \ldots, \mathbf{ab}[14] = a_{55}$; while for **uplo** = **Nag_Lower**, $\mathbf{ab}[0] = a_{11}, \mathbf{ab}[1] = a_{21}, \ldots, \mathbf{ab}[12] = a_{55}$. If **order** = **Nag_RowMajor** (**pdab** = 3), then for **uplo** = **Nag_Upper**, $\mathbf{ab}[0] = a_{11}, \mathbf{ab}[1] = a_{12}, \ldots, \mathbf{ab}[12] = a_{55}$; while for **uplo** = **Nag_Lower**, $\mathbf{ab}[2] = a_{11}, \mathbf{ab}[4] = a_{21}, \ldots, \mathbf{ab}[14] = a_{55}$.

3.3.4 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** = **Nag_UnitDiag** (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.

3.3.5 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.

3.4 Argument Conventions

3.4.1 Option arguments

In addition to the **order** argument of type **Nag_OrderType**, most functions in this Chapter have one or more option arguments of various types; only options of the correct type may be supplied.

For example,

```
f07fdc(Nag_RowMajor,Nag_Upper,...)
```

3.4.2 Problem dimensions

It is permissible for the problem dimensions (for example, **m** in nag_dgetrf (f07adc), **n** or **nrhs** in nag_dgetrs (f07aec)) to be passed as zero, in which case the computation (or part of it) is skipped. Negative dimensions are regarded as an error.

3.5 Tables of Available Computational Functions

It should be noted that all the LAPACK computational functions from Release 3 are included in the NAG C Library and can be called by their LAPACK name*, although not all of these functions are currently documented in Chapters f07 and f08.

Each entry gives:

    the NAG function short name
    the LAPACK function name from which the NAG function long name is derived by prepending nag_.

Type of matrix and storage scheme	factorize	solve	condition number	error estimate	invert
general	nag_dgetrf (f07adc)	nag_dgetrs (f07aec)	nag_dgecon (f07agc)	nag_dgerfs (f07ahc)	nag_dgetri (f07ajc)
general band	nag_dgbtrf (f07bdc)	nag_dgbtrs (f07bec)	nag_dgbcon (f07bgc)	nag_dgbrfs (f07bhc)	
symmetric positive-definite	nag_dpotrf (f07fdc)	nag_dpotrs (f07fec)	nag_dpocon (f07fgc)	nag_dporfs (f07fhc)	nag_dpotri (f07fjc)
symmetric positive-definite (packed storage)	nag_dpptrf (f07gdc)	nag_dpptrs (f07gec)	nag_dppcon (f07ggc)	nag_dpprfs (f07ghc)	nag_dpptri (f07gjc)
symmetric positive-definite band	nag_dpbtrf (f07hdc)	nag_dpbtrs (f07hec)	nag_dpbcon (f07hgc)	nag_dpbrfs (f07hhc)	
symmetric indefinite	nag_dsytrf (f07mdc)	nag_dsytrs (f07mec)	nag_dsycon (f07mgc)	nag_dsyrfs (f07mhc)	nag_dsytri (f07mjc)
symmetric indefinite (packed storage)	nag_dsptrf (f07pdc)	nag_dsptrs (f07pec)	nag_dspcon (f07pgc)	nag_dsprfs (f07phc)	nag_dsptri (f07pjc)
triangular		nag_dtrtrs (f07tec)	nag_dtrcon (f07tgc)	nag_dtrrfs (f07thc)	nag_dtrtri (f07tjc)
triangular (packed storage)		nag_dtptrs (f07uec)	nag_dtpcon (f07ugc)	nag_dtprfs (f07uhc)	nag_dtptri (f07ujc)
triangular band		nag_dtbtrs (f07vec)	nag_dtbcon (f07vgc)	nag_dtbrfs (f07vhc)	

Each entry gives:

    the NAG function short name
    the LAPACK function name from which the NAG function long name is derived by prepending nag_.

Type of matrix and storage scheme	factorize	solve	condition number	error estimate	invert
general	nag_zgetrf (f07arc)	nag_zgetrs (f07asc)	nag_zgecon (f07auc)	nag_zgerfs (f07avc)	nag_zgetri (f07awc)
general band	nag_zgbtrf (f07brc)	nag_zgbtrs (f07bsc)	nag_zgbcon (f07buc)	nag_zgbrfs (f07bvc)	
Hermitian positive-definite	nag_zpotrf (f07frc)	nag_zpotrs (f07fsc)	nag_zpocon (f07fuc)	nag_zporfs (f07fvc)	nag_zpotri (f07fwc)
Hermitian positive-definite (packed storage)	nag_zpptrf (f07grc)	nag_zpptrs (f07gsc)	nag_zppcon (f07guc)	nag_zpprfs (f07gvc)	nag_zpptri (f07gwc)
Hermitian positive-definite band	nag_zpbtrf (f07hrc)	nag_zpbtrs (f07hsc)	nag_zpbcon (f07huc)	nag_zpbrfs (f07hvc)	
Hermitian indefinite	nag_zhetrf (f07mrc)	nag_zhetrs (f07msc)	nag_zhecon (f07muc)	nag_zherfs (f07mvc)	nag_zhetri (f07mwc)
symmetric indefinite	nag_zsytrf (f07nrc)	nag_zsytrs (f07nsc)	nag_zsycon (f07nuc)	nag_zsyrfs (f07nvc)	nag_zsytri (f07nwc)
Hermitian indefinite (packed storage)	nag_zhptrf (f07prc)	nag_zhptrs (f07psc)	nag_zhpcon (f07puc)	nag_zhprfs (f07pvc)	nag_zhptri (f07pwc)
symmetric indefinite (packed storage)	nag_zsptrf (f07qrc)	nag_zsptrs (f07qsc)	nag_zspcon (f07quc)	nag_zsprfs (f07qvc)	nag_zsptri (f07qwc)
triangular		nag_ztrtrs (f07tsc)	nag_ztrcon (f07tuc)	nag_ztrrfs (f07tvc)	nag_ztrtri (f07twc)
triangular (packed storage)		nag_ztptrs (f07usc)	nag_ztpcon (f07uuc)	nag_ztprfs (f07uvc)	nag_ztptri (f07uwc)
triangular band		nag_ztbtrs (f07vsc)	nag_ztbcon (f07vuc)	nag_ztbrfs (f07vvc)	

4   Index

Apply iterative refinement to the solution and compute error estimates:
    after factorizing the matrix of coefficients:
        complex band matrix .... nag_zgbrfs (f07bvc)
        complex Hermitian indefinite matrix .... nag_zherfs (f07mvc)
        complex Hermitian indefinite matrix, packed storage .... nag_zhprfs (f07pvc)
        complex Hermitian positive-definite band matrix .... nag_zpbrfs (f07hvc)
        complex Hermitian positive-definite matrix .... nag_zporfs (f07fvc)
        complex Hermitian positive-definite matrix, packed storage .... nag_zpprfs (f07gvc)
        complex matrix .... nag_zgerfs (f07avc)
        complex symmetric indefinite matrix .... nag_zsyrfs (f07nvc)
        complex symmetric indefinite matrix, packed storage .... nag_zsprfs (f07qvc)
        real band matrix .... nag_dgbrfs (f07bhc)
        real matrix .... nag_dgerfs (f07ahc)
        real symmetric indefinite matrix .... nag_dsyrfs (f07mhc)
        real symmetric indefinite matrix, packed storage .... nag_dsprfs (f07phc)
        real symmetric positive-definite band matrix .... nag_dpbrfs (f07hhc)
        real symmetric positive-definite matrix .... nag_dporfs (f07fhc)
        real symmetric positive-definite matrix, packed storage .... nag_dpprfs (f07ghc)
Compute error estimates:
    complex triangular band matrix .... nag_ztbrfs (f07vvc)
    complex triangular matrix .... nag_ztrrfs (f07tvc)
    complex triangular matrix, packed storage .... nag_ztprfs (f07uvc)
    real triangular band matrix .... nag_dtbrfs (f07vhc)
    real triangular matrix .... nag_dtrrfs (f07thc)
    real triangular matrix, packed storage .... nag_dtprfs (f07uhc)
Condition number estimation:
    after factorizing the matrix of coefficients:
        complex band matrix .... nag_zgbcon (f07buc)
        complex Hermitian indefinite matrix .... nag_zhecon (f07muc)
        complex Hermitian indefinite matrix, packed storage .... nag_zhpcon (f07puc)
        complex Hermitian positive-definite band matrix .... nag_zpbcon (f07huc)
        complex Hermitian positive-definite matrix .... nag_zpocon (f07fuc)
        complex Hermitian positive-definite matrix, packed storage .... nag_zppcon (f07guc)
        complex matrix .... nag_zgecon (f07auc)
        complex symmetric indefinite matrix .... nag_zsycon (f07nuc)
        complex symmetric indefinite matrix, packed storage .... nag_zspcon (f07quc)
        real band matrix .... nag_dgbcon (f07bgc)
        real matrix .... nag_dgecon (f07agc)
        real symmetric indefinite matrix .... nag_dsycon (f07mgc)
        real symmetric indefinite matrix, packed storage .... nag_dspcon (f07pgc)
        real symmetric positive-definite band matrix .... nag_dpbcon (f07hgc)
        real symmetric positive-definite matrix .... nag_dpocon (f07fgc)
        real symmetric positive-definite matrix, packed storage .... nag_dppcon (f07ggc)
    complex triangular band matrix .... nag_ztbcon (f07vuc)
    complex triangular matrix .... nag_ztrcon (f07tuc)
    complex triangular matrix, packed storage .... nag_ztpcon (f07uuc)
    real triangular band matrix .... nag_dtbcon (f07vgc)
    real triangular matrix .... nag_dtrcon (f07tgc)
    real triangular matrix, packed storage .... nag_dtpcon (f07ugc)
LL^{T} or $U^{\mathrm{T}}U$ factorization:
    complex Hermitian positive-definite band matrix .... nag_zpbtrf (f07hrc)
    complex Hermitian positive-definite matrix .... nag_zpotrf (f07frc)
    complex Hermitian positive-definite matrix, packed storage .... nag_zpptrf (f07grc)
    real symmetric positive-definite band matrix .... nag_dpbtrf (f07hdc)
    real symmetric positive-definite matrix .... nag_dpotrf (f07fdc)
    real symmetric positive-definite matrix, packed storage .... nag_dpptrf (f07gdc)

LU factorization:
    complex band matrix .... nag_zgbtrf (f07brc)
    complex matrix .... nag_zgetrf (f07arc)
    real band matrix .... nag_dgbtrf (f07bdc)
    real matrix .... nag_dgetrf (f07adc)
Matrix inversion:
    after factorizing the matrix of coefficients:
        complex Hermitian indefinite matrix .... nag_zhetri (f07mwc)
        complex Hermitian indefinite matrix, packed storage .... nag_zhptri (f07pwc)
        complex Hermitian positive-definite matrix .... nag_zpotri (f07fwc)
        complex Hermitian positive-definite matrix, packed storage .... nag_zpptri (f07gwc)
        complex matrix .... nag_zgetri (f07awc)
        complex symmetric indefinite matrix .... nag_zsytri (f07nwc)
        complex symmetric indefinite matrix, packed storage .... nag_zsptri (f07qwc)
        real matrix .... nag_dgetri (f07ajc)
        real symmetric indefinite matrix .... nag_dsytri (f07mjc)
        real symmetric indefinite matrix, packed storage .... nag_dsptri (f07pjc)
        real symmetric positive-definite matrix .... nag_dpotri (f07fjc)
        real symmetric positive-definite matrix, packed storage .... nag_dpptri (f07gjc)
    complex triangular matrix .... nag_ztrtri (f07twc)
    complex triangular matrix, packed storage .... nag_ztptri (f07uwc)
    real triangular matrix .... nag_dtrtri (f07tjc)
    real triangular matrix, packed storage .... nag_dtptri (f07ujc)
$PLDL^{\mathrm{T}}P^{\mathrm{T}}$ or $PUDU^{\mathrm{T}}P^{\mathrm{T}}$ factorization:
    complex Hermitian indefinite matrix .... nag_zhetrf (f07mrc)
    complex Hermitian indefinite matrix, packed storage .... nag_zhptrf (f07prc)
    complex symmetric indefinite matrix .... nag_zsytrf (f07nrc)
    complex symmetric indefinite matrix, packed storage .... nag_zsptrf (f07qrc)
    real symmetric indefinite matrix .... nag_dsytrf (f07mdc)
    real symmetric indefinite matrix, packed storage .... nag_dsptrf (f07pdc)
Solution of simultaneous linear equations:
    after factorizing the matrix of coefficients:
        complex band matrix .... nag_zgbtrs (f07bsc)
        complex Hermitian indefinite matrix .... nag_zhetrs (f07msc)
        complex Hermitian indefinite matrix, packed storage .... nag_zhptrs (f07psc)
        complex Hermitian positive-definite band matrix .... nag_zpbtrs (f07hsc)
        complex Hermitian positive-definite matrix .... nag_zpotrs (f07fsc)
        complex Hermitian positive-definite matrix, packed storage .... nag_zpptrs (f07gsc)
        complex matrix .... nag_zgetrs (f07asc)
        complex symmetric indefinite matrix .... nag_zsytrs (f07nsc)
        complex symmetric indefinite matrix, packed storage .... nag_zsptrs (f07qsc)
        real band matrix .... nag_dgbtrs (f07bec)
        real matrix .... nag_dgetrs (f07aec)
        real symmetric indefinite matrix .... nag_dsytrs (f07mec)
        real symmetric indefinite matrix, packed storage .... nag_dsptrs (f07pec)
        real symmetric positive-definite band matrix .... nag_dpbtrs (f07hec)
        real symmetric positive-definite matrix .... nag_dpotrs (f07fec)
        real symmetric positive-definite matrix, packed storage .... nag_dpptrs (f07gec)
    complex triangular band matrix .... nag_ztbtrs (f07vsc)
    complex triangular matrix .... nag_ztrtrs (f07tsc)
    complex triangular matrix, packed storage .... nag_ztptrs (f07usc)
    real triangular band matrix .... nag_dtbtrs (f07vec)
    real triangular matrix .... nag_dtrtrs (f07tec)
    real triangular matrix, packed storage .... nag_dtptrs (f07uec)

5   Functions Withdrawn or Scheduled for Withdrawal

None.

6   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