

NAG Library Chapter Introduction

f07 – Linear Equations (LAPACK)

Contents

1 Scope of the Chapter	2
2 Background to the Problems	2
2.1 Notation	2
2.2 Matrix Factorizations	3
2.3 Solution of Systems of Equations	3
2.4 Sensitivity and Error Analysis	4
2.4.1 Normwise error bounds	4
2.4.2 Estimating condition numbers	4
2.4.3 Scaling and Equilibration	4
2.4.4 Componentwise error bounds	5
2.4.5 Iterative refinement of the solution	5
2.5 Matrix Inversion	6
2.6 Packed Storage Formats	6
2.7 Band and Tridiagonal Matrices	6
2.8 Block Partitioned Algorithms	6
2.9 Mixed Precision LAPACK Routines	7
3 Recommendations on Choice and Use of Available Functions	7
3.1 Available Functions	7
3.2 NAG Names and LAPACK Names	8
3.3 Matrix Storage Schemes	9
3.3.1 Conventional storage	9
3.3.2 Packed storage	10
3.3.3 Rectangular Full Packed (RFP) Storage	11
3.3.4 Band storage	13
3.3.5 Unit triangular matrices	14
3.3.6 Real diagonal elements of complex matrices	15
3.4 Parameter Conventions	15
3.4.1 Option arguments	15
3.4.2 Problem dimensions	15
3.5 Tables of Driver and Computational Functions	15
3.5.1 Real matrices	15
3.5.2 Complex matrices	16
4 Functionality Index	18
5 Auxiliary Functions Associated with Library Function Arguments	21
6 Functions Withdrawn or Scheduled for Withdrawal	21
7 References	21

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, in Section 4 in 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 nag_d..sv (f07.ac) and for complex matrices have names of the form nag_z..sv (f07.nc). The expert drivers for real matrices have names of the form nag_d..svx (f07.bc) and for complex matrices have names of the form nag_z..svx (f07.pc).

The functions in this chapter (Chapter f07) handle only *dense* and *band* matrices (not matrices with more specialised 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 <i>computed</i> solution to $Ax = b$, (which usually differs from the exact solution x because of round-off error)
$r = b - A\hat{x}$	the <i>residual</i> 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 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)

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

where U is upper triangular and L is lower triangular.

Symmetric positive semidefinite matrices (pivoted Cholesky factorization)

$$A = PU^T U P^T \quad \text{or} \quad A = PLL^T P^T$$

where P is a permutation matrix, U is upper triangular and L is lower triangular. The permutation matrix P (which represents row-and-column interchanges) is needed to ensure numerical stability and to reveal the numerical rank of A .

Symmetric indefinite matrices (Bunch–Kaufman factorization)

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

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 $A = UDU^T$ or $A = LDL^T$ 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)

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

Symmetric positive definite matrices (Cholesky factorization)

$$\begin{array}{l} U^T y = b \\ U x = y \end{array} \quad \text{or} \quad \begin{array}{l} L y = b \\ L^T x = y \end{array}$$

Symmetric indefinite matrices (Bunch–Kaufman factorization)

$$\begin{array}{l} P U D y = b \\ U^T P^T x = y \end{array} \quad \text{or} \quad \begin{array}{l} P L D y = b \\ L^T P^T x = y \end{array}$$

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

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

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

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

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 emphasized 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 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

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

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:

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

Care is taken when computing this bound to allow for rounding errors in computing r . The norm $\| |A^{-1}| \cdot |r| \|_\infty$ 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 $i = 0, 1, \dots$, 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 $|A| \cdot |x|$ 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.

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 Formats

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 $n(n+1)/2$ or a two-dimensional array with $n(n+1)/2$ elements; in other words, the storage is almost halved.

The one-dimensional array storage format is referred to as packed storage; it is described in Section 3.3.2. The two-dimensional array storage format is referred to as Rectangular Full Packed (RFP) format; it is described in Section 3.3.3. They may also be used for triangular matrices.

Functions designed for these packed storage formats perform the same number of arithmetic operations as functions which use conventional storage. Those using a packed one-dimensional array are usually less efficient, especially on high-performance computers, so there is then a trade-off between storage and efficiency. The RFP functions are as efficient as for conventional storage, although only a small subset of functions use this format.

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 3.3.4.

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 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 **block size** – 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 block size of 1), relying solely on calls to the Level 2 BLAS (see Chapter f16 again).

2.9 Mixed Precision LAPACK Routines

Some LAPACK routines use mixed precision arithmetic in an effort to solve problems more efficiently on modern hardware. They work by converting a double precision problem into an equivalent single precision problem, solving it and then using iterative refinement in double precision to find a full precision solution to the original problem. The method may fail if the problem is too ill-conditioned to allow the initial single precision solution, in which case the functions fall back to solve the original problem entirely in double precision. The vast majority of problems are not so ill-conditioned, and in those cases the technique can lead to significant gains in speed without loss of accuracy. This is particularly true on machines where double precision arithmetic is significantly slower than single precision.

3 Recommendations on Choice and Use of Available Functions

3.1 Available Functions

Tables 1 and 8 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 8 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 (RFP 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 (RFP storage)
- triangular band

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

- (a) (except for RFP matrices) solve a system of linear equations (driver functions);
- (b) (except for RFP matrices) 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) (except for RFP matrices) 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) (except for RFP matrices) 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).

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 8 show the LAPACK routine 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 routine names follow a simple scheme. Most names have 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
- exceptionally, the mixed precision LAPACK routines described in Section 2.9 replace the initial first letter by a pair of letters, as:
 - ds – double precision function using single precision internally
 - zc – double complex function using single precision complex internally
- the 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
 - pf – symmetric or Hermitian positive definite (rectangular full packed (RFP) storage)
 - pp – symmetric or Hermitian positive definite (packed storage)
 - pb – symmetric or Hermitian positive definite band
 - sy – symmetric indefinite
 - sf – symmetric indefinite (rectangular full packed (RFP) storage)
 - sp – symmetric indefinite (packed storage)
 - he – (complex) Hermitian indefinite
 - hf – (complex) Hermitian indefinite (rectangular full packed (RFP) storage)

- hp – (complex) Hermitian indefinite (packed storage)
- gt – general tridiagonal
- pt – symmetric or Hermitian positive definite tridiagonal
- tr – triangular
- tf – triangular (rectangular full packed (RFP) storage)
- tp – triangular (packed storage)
- tb – triangular band
- the last two or three letters **zz** or **zzz** indicate the computation performed. Examples are:
 - 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 `nag_dgetrf` performs a triangular factorization of a real general matrix in double precision; the corresponding function for a complex general matrix is `nag_zgetrf`.

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;
- rectangular full packed (RFP) 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.

3.3.1 Conventional storage

Matrices may be stored column-wise or row-wise as described in Section 2.3.1.4 in How to Use the NAG Library and its Documentation: a matrix A is stored in a one-dimensional array **a**, with matrix element $a_{i,j}$ stored column-wise in array element **a**[($j - 1$) \times **pda** + $i - 1$] or row-wise in array element **a**[($i - 1$) \times **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 * or \square in the examples below.

For example, when $n = 3$:

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

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	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ \bar{a}_{12} & a_{22} & a_{23} \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} \end{pmatrix}$	$a_{11} \sqcup a_{12} a_{22} \sqcup a_{13} a_{23} a_{33}$
Nag_RowMajor	Nag_Upper	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ \bar{a}_{12} & a_{22} & a_{23} \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} \end{pmatrix}$	$a_{11} a_{12} a_{13} \sqcup a_{22} a_{23} \sqcup a_{33}$
Nag_ColMajor	Nag_Lower	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} \\ a_{21} & a_{22} & \bar{a}_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11} a_{21} a_{31} \sqcup a_{22} a_{32} \sqcup a_{33}$
Nag_RowMajor	Nag_Lower	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} \\ a_{21} & a_{22} & \bar{a}_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$a_{11} \sqcup a_{21} a_{22} \sqcup a_{31} a_{32} a_{33}$

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 in a one-dimensional array. In this chapter, as in Chapters f08 and f16, arrays which hold matrices in packed storage, have names ending in P. For a matrix of order n , the array must have at least $n(n+1)/2$ elements. So:

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

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

For example:

	Triangle of matrix A	Packed storage in array \mathbf{ap}
uplo = Nag_Upper	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$a_{11} \underbrace{a_{12} a_{22}} \underbrace{a_{13} a_{23} a_{33}} \underbrace{a_{14} a_{24} a_{34} a_{44}}$
uplo = Nag_Lower	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11} a_{21} a_{31} a_{41}} \underbrace{a_{22} a_{32} a_{42}} \underbrace{a_{33} a_{43}} a_{44}$

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.)

3.3.3 Rectangular Full Packed (RFP) Storage

The rectangular full packed (RFP) storage format offers the same savings in storage as the packed storage format (described in Section 3.3.2), but is likely to be much more efficient in general since the block structure of the matrix is maintained. This structure can be exploited using block partition algorithms (see Section 2.8) in a similar way to matrices that use conventional storage.

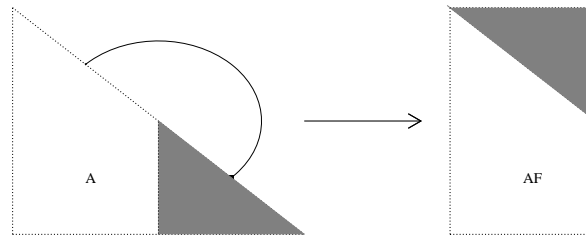


Figure 1 gives a graphical representation of the key idea of RFP for the particular case of a lower triangular matrix of even dimensions. In all cases the original triangular matrix of stored elements is separated into a trapezoidal part and a triangular part. The number of columns in these two parts is equal when the dimension of the matrix is even, $n = 2k$, while the trapezoidal part has $k + 1$ columns when $n = 2k + 1$. The smaller part is then transposed and fitted onto the trapezoidal part forming a rectangle. The rectangle has dimensions $2k + 1$ and q , where $q = k$ when n is even and $q = k + 1$ when n is odd.

For functions using RFP there is the option of storing the rectangle as described above (**transr** = Nag_RFP_Normal) or its transpose (**transr** = Nag_RFP_Trans, for real **a**) or its conjugate transpose (**transr** = Nag_RFP_ConjTrans, for complex **a**). It is useful to note that the storage ordering for Nag_RowMajor is the same as that for Nag_ColMajor with the value of **transr** = Nag_RFP_Normal switched to **transr** = Nag_RFP_Trans or vice versa.

As an example, we first consider RFP for the case $n = 2k$ with $k = 3$.

If **transr** = Nag_RFP_Normal, then **ar** holds **a** as follows:

For **uplo** = Nag_Upper the upper trapezoid **AR**(1 : 6, 1 : 3) consists of the last three columns of **a** upper. The lower triangle **AR**(5 : 7, 1 : 3) consists of the transpose of the first three columns of **a** upper.

For **uplo** = Nag_Lower the lower trapezoid **AR**(2 : 7, 1 : 3) consists of the first three columns of **a** lower. The upper triangle **AR**(1 : 3, 1 : 3) consists of the transpose of the last three columns of **a** lower.

If **transr** = Nag_RFP_Trans, then **ar** in both **uplo** cases is just the transpose of **ar** as defined when **transr** = Nag_RFP_Normal.

uplo	Triangle of matrix A	Rectangular Full Packed matrix AR	
		transr = Nag_RFP_Normal	transr = Nag_RFP_Trans
Nag_Upper	$\begin{pmatrix} 00 & 01 & 02 & 03 & 04 & 05 \\ & 11 & 12 & 13 & 14 & 15 \\ & & 22 & 23 & 24 & 25 \\ & & & 33 & 34 & 35 \\ & & & & 44 & 45 \\ & & & & & 55 \end{pmatrix}$	03 04 05 13 14 15 23 24 25 33 34 35 00 44 45 01 11 55 02 12 22	03 13 23 33 00 01 02 04 14 24 34 44 11 12 05 15 25 35 45 55 22
Nag_Lower	$\begin{pmatrix} 00 & & & & & \\ 10 & 11 & & & & \\ 20 & 21 & 22 & & & \\ 30 & 31 & 32 & 33 & & \\ 40 & 41 & 42 & 43 & 44 & \\ 50 & 51 & 52 & 53 & 54 & 55 \end{pmatrix}$	33 43 53 00 44 54 10 11 55 20 21 22 30 31 32 40 41 42 50 51 52	33 00 10 20 30 40 50 43 44 11 21 31 41 51 53 54 55 22 32 42 52

Now we consider RFP for the case $n = 2k + 1$ and $k = 2$.

If **transr** = Nag_RFP_Normal. **ar** holds **a** as follows:

if **uplo** = Nag_Upper the upper trapezoid **AR**(1 : 5, 1 : 3) consists of the last three columns of **a** upper. The lower triangle **AR**(4 : 5, 1 : 2) consists of the transpose of the first two columns of **a** upper;

if **uplo** = Nag_Lower the lower trapezoid **AR**(1 : 5, 1 : 3) consists of the first three columns of **a** lower. The upper triangle **AR**(1 : 2, 2 : 3) consists of the transpose of the last two columns of **a** lower.

If **transr** = Nag_RFP_Trans. **ar** in both **uplo** cases is just the transpose of **ar** as defined when **transr** = Nag_RFP_Normal.

uplo	Triangle of matrix A	Rectangular Full Packed matrix AR	
		transr = Nag_RFP_Normal	transr = Nag_RFP_Trans
Nag_Upper	$\begin{pmatrix} 00 & 01 & 02 & 03 & 04 \\ & 11 & 12 & 13 & 14 \\ & & 22 & 23 & 24 \\ & & & 33 & 34 \\ & & & & 44 \end{pmatrix}$	02 03 04 12 13 14 22 23 24 00 33 34 01 11 44	02 12 22 00 01 03 13 23 33 11 04 14 24 34 44
Nag_Lower	$\begin{pmatrix} 00 & & & & \\ 10 & 11 & & & \\ 20 & 21 & 22 & & \\ 30 & 31 & 32 & 33 & \\ 40 & 41 & 42 & 43 & 44 \end{pmatrix}$	00 33 43 10 11 44 20 21 22 30 31 32 40 41 42	00 10 20 30 40 50 33 11 21 31 41 51 43 44 22 32 42 52

Explicitly, in the real matrix case, **ar** is a one-dimensional array of length $n(n + 1)/2$ and, when Nag_ColMajor, contains the elements of **a** as follows:

for **uplo** = Nag_Upper and **transr** = Nag_RFP_Normal,

a_{ij} is stored in **ar**[(2k + 1)(i - 1) + j + k], for $1 \leq j \leq k$ and $1 \leq i \leq j$, and
 a_{ij} is stored in **ar**[(2k + 1)(j - k - 1) + i - 1], for $k < j \leq n$ and $1 \leq i \leq j$;

for **uplo** = Nag_Upper and **transr** = Nag_RFP_Trans,

a_{ij} is stored in **ar**[q(j + k) + i - 1], for $1 \leq j \leq k$ and $1 \leq i \leq j$, and
 a_{ij} is stored in **ar**[q(i - 1) + j - k - 1], for $k < j \leq n$ and $1 \leq i \leq j$;

for **uplo** = Nag_Lower and **transr** = Nag_RFP_Normal,

a_{ij} is stored in **ar**[(2k + 1)(j - 1) + i + k - q], for $1 \leq j \leq q$ and $j \leq i \leq n$, and
 a_{ij} is stored in **ar**[(2k + 1)(i - k - 1) + j - q - 1], for $q < j \leq n$ and $j \leq i \leq n$;

for **uplo** = Nag_Lower and **transr** = Nag_RFP_Trans,

a_{ij} is stored in **ar**[q(i + k - q) + j - 1], for $1 \leq j \leq q$ and $1 \leq i \leq n$, and
 a_{ij} is stored in **ar**[q(j - 1 - q) + i - k - 1], for $q < j \leq n$ and $1 \leq i \leq n$.

When `Nag_RowMajor` the above storage formulae can be used by looking up the opposite case for **transr**, i.e., when **transr** = `Nag_RFP_Trans` look up the storage order above for the cases when **transr** = `Nag_RFP_Normal` and vice versa.

In the case of complex matrices, the assumption is that the full matrix, if it existed, would be Hermitian. Thus, when **transr** = `Nag_RFP_Normal`, the triangular portion of **a** that is, in the real case, transposed into the notional $(2k+1)$ by q RFP matrix is also conjugated. When **transr** = `Nag_RFP_ConjTrans` the notional q by $(2k+1)$ RFP matrix is the conjugate transpose of the corresponding **transr** = `Nag_RFP_Normal` RFP matrix. Explicitly, for complex **a**, the array **ar** contains the elements (or conjugated elements) of **a** as follows:

for **uplo** = `Nag_Upper` and **transr** = `Nag_RFP_Normal`,

\bar{a}_{ij} is stored in **ar** $[(2k+1)(i-1) + j + k]$, for $1 \leq j \leq k$ and $1 \leq i \leq j$, and

a_{ij} is stored in **ar** $[(2k+1)(j-k-1) + i - 1]$, for $k < j \leq n$ and $1 \leq i \leq j$;

for **uplo** = `Nag_Upper` and **transr** = `Nag_RFP_ConjTrans`,

a_{ij} is stored in **ar** $[q(j+k) + i - 1]$, for $1 \leq j \leq k$ and $1 \leq i \leq j$, and

\bar{a}_{ij} is stored in **ar** $[q(i-1) + j - k - 1]$, for $k < j \leq n$ and $1 \leq i \leq j$;

for **uplo** = `Nag_Lower` and **transr** = `Nag_RFP_Normal`,

a_{ij} is stored in **ar** $[(2k+1)(j-1) + i + k - q]$, for $1 \leq j \leq q$ and $j \leq i \leq n$, and

\bar{a}_{ij} is stored in **ar** $[(2k+1)(i-k-1) + j - q - 1]$, for $q < j \leq n$ and $j \leq i \leq n$;

for **uplo** = `Nag_Lower` and **transr** = `Nag_RFP_ConjTrans`,

\bar{a}_{ij} is stored in **ar** $[q(i+k-q) + j - 1]$, for $1 \leq j \leq q$ and $1 \leq i \leq n$, and

a_{ij} is stored in **ar** $[q(j-1-q) + i - k - 1]$, for $q < j \leq n$ and $1 \leq i \leq n$.

When `Nag_RowMajor` the above storage formulae can be used by looking up the opposite case for **transr**, i.e., when **transr** = `Nag_RFP_ConjTrans` look up the storage order above for the cases when **transr** = `Nag_RFP_Normal` and vice versa.

3.3.4 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** $[(j-1) \times \text{pdab} + k_u + i - j]$;

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

where $\text{pdab} \geq k_l + k_u + 1$ 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				
a_{11} a_{12}	*	a_{12}	a_{23}	a_{34}	a_{45}	*	*	a_{11}	a_{12}	
a_{21} a_{22} a_{23}	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	*	a_{21}	a_{22}	a_{23}	
a_{31} a_{32} a_{33} a_{34}	a_{21}	a_{32}	a_{43}	a_{54}	*	a_{31}	a_{32}	a_{33}	a_{34}	
a_{42} a_{43} a_{44} a_{45}	a_{31}	a_{42}	a_{53}	*	*	a_{42}	a_{43}	a_{44}	a_{45}	
a_{53} a_{54} a_{55}						a_{53}	a_{54}	a_{55}	*	

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 **ldab** takes the minimum value of 4, then **ab**[0] need not be set, **ab**[1] = a_{11} , **ab**[2] = a_{21} , ..., **ab**[17] = a_{55} . On the other hand, if **order** = Nag_RowMajor (**ldab** = 4), then **ab**[0] and **ab**[1] need not be set, **ab**[2] = a_{11} , **ab**[3] = a_{12} , ..., **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 **ldab** $\geq 2k_l + k_u + 1$.

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	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} & \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} & a_{35} \\ & \bar{a}_{24} & \bar{a}_{34} & a_{44} & a_{45} \\ & & \bar{a}_{35} & \bar{a}_{45} & a_{55} \end{pmatrix}$	$\begin{pmatrix} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \end{pmatrix}$	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ a_{22} & a_{23} & a_{24} & & \\ a_{33} & a_{34} & a_{35} & & \\ a_{44} & a_{45} & * & & \\ a_{55} & * & * & & \end{pmatrix}$
Nag_Lower	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & & \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} & \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} & \bar{a}_{53} \\ & a_{42} & a_{43} & a_{44} & \bar{a}_{54} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{pmatrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{pmatrix}$	$\begin{pmatrix} * & * & a_{11} & & \\ * & a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & & \\ a_{42} & a_{43} & a_{44} & & \\ a_{53} & a_{54} & a_{55} & & \end{pmatrix}$

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, **ab**[2] = a_{11} , **ab**[4] = a_{12} , ..., **ab**[14] = a_{55} ; while for **uplo** = Nag_Lower, **ab**[0] = a_{11} , **ab**[1] = a_{21} , ..., **ab**[12] = a_{55} . If **order** = Nag_RowMajor (**pdab** = 3), then for **uplo** = Nag_Upper, **ab**[0] = a_{11} , **ab**[1] = a_{12} , ..., **ab**[12] = a_{55} ; while for **uplo** = Nag_Lower, **ab**[2] = a_{11} , **ab**[4] = a_{21} , ..., **ab**[14] = a_{55} .

3.3.5 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.

3.3.6 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 Chapters f07 and f08, 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 Parameter 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,

```
nag_dpotrff(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 Driver and Computational Functions

3.5.1 Real matrices

Each entry gives:

the NAG function short name

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

Operation	Type of matrix and storage scheme		
	general	general band	general tridiagonal
driver	nag_dgesv (f07aac)	nag_dgbv (f07bac)	nag_dgtv (f07cac)
expert driver	nag_dgesvx (f07abc)	nag_dgbv (f07bbc)	nag_dgtvx (f07cbc)
mixed precision driver	nag_dsgesv (f07acc)		
factorize	nag_dgetrf (f07adc)	nag_dgbtrf (f07bdc)	nag_dgttrf (f07cdc)
solve	nag_dgetrs (f07aec)	nag_dgbtrs (f07bec)	nag_dgttrs (f07cec)
scaling factors	nag_dgeequ (f07afc)	nag_dgbequ (f07bfc)	
condition number	nag_dgecon (f07agc)	nag_dgbcon (f07bgc)	nag_dgtcon (f07cgc)
error estimate	nag_dgerfs (f07ahc)	nag_dgbrfs (f07bhc)	nag_dgtfrs (f07chc)
invert	nag_dgetri (f07ajc)		

Table 1
Functions for real general matrices

	Type of matrix and storage scheme			
Operation	symmetric positive definite	symmetric positive definite (packed storage)	symmetric positive definite band	symmetric positive definite tridiagonal
driver	nag_dposv (f07fac)	nag_dppsv (f07gac)	nag_dpbsv (f07hac)	nag_dptsv (f07jac)
expert driver	nag_dposvx (f07fbc)	nag_dppsvx (f07gbc)	nag_dpbsvx (f07hbc)	nag_dptsvx (f07jbc)
factorize	nag_dpotrf (f07fdc)	nag_dpptrf (f07gdc)	nag_dpbrf (f07hdc)	nag_dpstrf (f07jdc)
solve	nag_dpotrs (f07fec)	nag_dpptrs (f07gec)	nag_dpbrs (f07hec)	nag_dpstrs (f07jec)
scaling factors	nag_dpoequ (f07ffc)	nag_dppequ (f07gfc)	nag_dpbequ (f07hfc)	
condition number	nag_dpocon (f07fgc)	nag_dppcon (f07ggc)	nag_dpbcn (f07hgc)	nag_dpcon (f07jgc)
error estimate	nag_dporfs (f07fhc)	nag_dpprfs (f07ghc)	nag_dpbrfs (f07hhc)	nag_dptrfs (f07jhc)
invert	nag_dpstri (f07fjc)	nag_dpptri (f07gjc)		

Table 2

Functions for real symmetric positive definite matrices

	Type of matrix and storage scheme	
Operation	symmetric indefinite	symmetric indefinite (packed storage)
driver	nag_dsylv (f07mac)	nag_dspsv (f07pac)
expert driver	nag_dsylvx (f07mbc)	nag_dspsvx (f07pbc)
factorize	nag_dsytrf (f07mdc)	nag_dsptrf (f07pdc)
solve	nag_dsytrs (f07mec)	nag_dsptrs (f07pec)
condition number	nag_dsycon (f07mgc)	nag_dspcon (f07pgc)
error estimate	nag_dsyrrfs (f07mhc)	nag_dsprfs (f07phc)
invert	nag_dsytri (f07mjc)	nag_dsptri (f07pjc)

Table 3

Functions for real symmetric indefinite matrices

	Type of matrix and storage scheme		
Operation	triangular	triangular (packed storage)	triangular band
solve	nag_dtrtrs (f07tec)	nag_dtptrs (f07uec)	nag_dtbtrs (f07vec)
condition number	nag_dtrcon (f07tgc)	nag_dtpcon (f07ugc)	nag_dtbcon (f07vgc)
error estimate	nag_dtrrfs (f07thc)	nag_dtpfrfs (f07uhc)	nag_dtbfrfs (f07vhc)
invert	nag_dtrtri (f07tjc)	nag_dtptri (f07ujc)	

Table 4

Functions for real triangular matrices

3.5.2 Complex matrices

Each entry gives:

the NAG function short name

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

	Type of matrix and storage scheme		
Operation	general	general band	general tridiagonal
driver	nag_zgesv (f07anc)	nag_zgbv (f07bnc)	nag_zgtsv (f07cnc)
expert driver	nag_zgesvx (f07apc)	nag_zgbvsvx (f07bpc)	nag_zgtsvx (f07cpc)
mixed precision driver	nag_zcgesv (f07aqc)		
factorize	nag_zgetrf (f07arc)	nag_zgbtrf (f07brc)	nag_zgttrf (f07crc)
solve	nag_zgetrs (f07asc)	nag_zgbtrs (f07bsc)	nag_zgttrs (f07csc)
scaling factors	nag_zgeequ (f07atc)	nag_zgbequ (f07btc)	
condition number	nag_zgecon (f07auc)	nag_zgbcon (f07buc)	nag_zgtcon (f07cuc)
error estimate	nag_zgerfs (f07avc)	nag_zgbrfs (f07bvc)	nag_zgtrfs (f07cvc)
invert	nag_zgetri (f07awc)		

Table 5
Functions for complex general matrices

	Type of matrix and storage scheme			
Operation	Hermitian positive definite	Hermitian positive definite (packed storage)	Hermitian positive definite band	Hermitian positive definite tridiagonal
driver	nag_zposv (f07fnc)	nag_zppsv (f07gnc)	nag_zpbsv (f07hnc)	nag_zptsv (f07jnc)
expert driver	nag_zposvx (f07fpc)	nag_zppsvx (f07gpc)	nag_zpbsvx (f07hpc)	nag_zptsvx (f07jpc)
factorize	nag_zpotrf (f07frc)	nag_zpptrf (f07grc)	nag_zpbtrf (f07hrc)	nag_zpttrf (f07jrc)
solve	nag_zpotrs (f07fsc)	nag_zpptrs (f07gsc)	nag_zpbtrs (f07hsc)	nag_zpttrs (f07jsc)
scaling factors	nag_zpoequ (f07ftc)	nag_zppequ (f07gtc)		
condition number	nag_zpocon (f07fuc)	nag_zppcon (f07guc)	nag_zpbcon (f07huc)	nag_zptcon (f07juc)
error estimate	nag_zporfs (f07fvc)	nag_zpprfs (f07gvc)	nag_zpbrfs (f07hvc)	nag_zptrfs (f07jvc)
invert	nag_zpotri (f07fwc)	nag_zpptri (f07gwc)		

Table 6
Functions for complex Hermitian positive definite matrices

	Type of matrix and storage scheme			
Operation	Hermitian indefinite	symmetric indefinite (packed storage)	Hermitian indefinite band	symmetric indefinite tridiagonal
driver	nag_zhesv (f07mnc)	nag_zsysv (f07nnc)	nag_zhpsv (f07pnc)	nag_zspsv (f07qnc)
expert driver	nag_zhesvx (f07mpc)	nag_zsysvx (f07npc)	nag_zhpsvx (f07ppc)	nag_zspsvx (f07qpc)
factorize	nag_zhetrf (f07mrc)	nag_zsytrf (f07nrc)	nag_zhptrf (f07prc)	nag_zsptf (f07qrc)
solve	nag_zhetrs (f07msc)	nag_zsytrs (f07nsc)	nag_zhptrs (f07psc)	nag_zspters (f07qsc)
condition number	nag_zhecon (f07muc)	nag_zsycon (f07nuc)	nag_zhpcon (f07puc)	nag_zspcon (f07quc)
error estimate	nag_zherfs (f07mvc)	nag_zsyfrs (f07nvc)	nag_zhprfs (f07pvc)	nag_zsprfs (f07qvc)
invert	nag_zhetri (f07mwc)	nag_zsytri (f07nwc)	nag_zhpri (f07pwc)	nag_zsptri (f07qwc)

Table 7
Functions for complex Hermitian and symmetric indefinite matrices

Operation	Type of matrix and storage scheme		
	triangular	triangular (packed storage)	triangular band
solve	nag_ztrtrs (f07tsc)	nag_ztptrs (f07usc)	nag_ztbtrs (f07vsc)
condition number	nag_ztrcon (f07tuc)	nag_ztpcon (f07uuc)	nag_ztbcon (f07vuc)
error estimate	nag_ztrrfs (f07tvc)	nag_ztparfs (f07uvc)	nag_ztbrfs (f07vvc)
invert	nag_ztrtri (f07twc)	nag_ztptri (f07uwc)	

Table 8
Functions for complex triangular matrices

4 Functionality 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 Hermitian positive definite tridiagonal matrix nag_zptrfs (f07jvc)
 complex matrix nag_zgerfs (f07avc)
 complex symmetric indefinite matrix nag_zsyrrfs (f07nvc)
 complex symmetric indefinite matrix, packed storage nag_zsprfs (f07qvc)
 complex tridiagonal matrix nag_zgtrfs (f07cvc)
 real band matrix nag_dgbrfs (f07bhc)
 real matrix nag_dgerfs (f07ahc)
 real symmetric indefinite matrix nag_dsyrrfs (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)
 real symmetric positive definite tridiagonal matrix nag_dptrfs (f07jhc)
 real tridiagonal matrix nag_dgtrfs (f07chc)

Compute error estimates,

complex triangular band matrix nag_ztbrfs (f07vvc)
 complex triangular matrix nag_ztrrfs (f07tvc)
 complex triangular matrix, packed storage nag_ztparfs (f07uvc)
 real triangular band matrix nag_dtbrfs (f07vhc)
 real triangular matrix nag_dtrrfs (f07thc)
 real triangular matrix, packed storage nag_dtparfs (f07uhc)

Compute row and column scalings,

complex band matrix nag_zgbequ (f07btc)
 complex Hermitian positive definite band matrix nag_zpbequ (f07htc)
 complex Hermitian positive definite matrix nag_zpoequ (f07ftc)
 complex Hermitian positive definite matrix, packed storage nag_zppequ (f07gtc)
 complex matrix nag_zgeequ (f07atc)
 real band matrix nag_dgbequ (f07bfc)
 real matrix nag_dgeequ (f07afc)
 real symmetric positive definite band matrix nag_dpbequ (f07hfc)
 real symmetric positive definite matrix nag_dpoequ (f07ffc)
 real symmetric positive definite matrix, packed storage nag_dppequ (f07gfc)

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 Hermitian positive definite tridiagonal matrix	nag_zptcon (f07juc)
complex matrix	nag_zgecon (f07auc)
complex symmetric indefinite matrix	nag_zsycon (f07nuc)
complex symmetric indefinite matrix, packed storage	nag_zspcon (f07quc)
complex tridiagonal matrix	nag_zgtcon (f07cuc)
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)
real symmetric positive definite tridiagonal matrix	nag_dptcon (f07jgc)
real tridiagonal matrix	nag_dgtcon (f07cgc)
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)

LDL^T factorization,

complex Hermitian positive definite tridiagonal matrix	nag_zpttrf (f07jrc)
real symmetric positive definite tridiagonal matrix	nag_dpttrf (f07jdc)

LL^T or *U^TU* 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)
complex Hermitian positive definite matrix, RFP storage	nag_zpftf (f07wrc)
complex Hermitian positive semidefinite matrix	nag_zpstf (f07krc)
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)
real symmetric positive definite matrix, RFP storage	nag_dpftf (f07wdc)
real symmetric positive semidefinite matrix	nag_dpstf (f07kdc)

LU factorization,

complex band matrix	nag_zgbtrf (f07brc)
complex matrix	nag_zgetrf (f07arc)
complex tridiagonal matrix	nag_zgttrf (f07crc)
real band matrix	nag_dgbtrf (f07bdc)
real matrix	nag_dgetrf (f07adc)
real tridiagonal matrix	nag_dgttrf (f07cdc)

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 Hermitian positive definite matrix, RFP storage	nag_zpftri (f07wwc)
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)
real symmetric positive definite matrix, RFP storage	nag_dpftri (f07wjc)
complex triangular matrix	nag_ztrtri (f07twc)
complex triangular matrix, packed storage	nag_ztptri (f07uwc)
complex triangular matrix, RFP storage, expert driver	nag_ztftri (f07wxc)
real triangular matrix	nag_dtrtri (f07tjc)
real triangular matrix, packed storage	nag_dtptri (f07ujc)
real triangular matrix, RFP storage, expert driver	nag_dtftri (f07wkc)
<i>PLDL^TP^T or PUDU^TP^T factorization,</i>	
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 Hermitian positive definite matrix, RFP storage	nag_zpftrs (f07wsc)
complex Hermitian positive definite tridiagonal matrix	nag_zpttrs (f07jsc)
complex matrix	nag_zgetrs (f07asc)
complex symmetric indefinite matrix	nag_zsytrs (f07nsc)
complex symmetric indefinite matrix, packed storage	nag_zsptrs (f07qsc)
complex tridiagonal matrix	nag_zgttrs (f07csc)
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)
real symmetric positive definite matrix, RFP storage	nag_dpfttrs (f07wec)
real symmetric positive definite tridiagonal matrix	nag_dpttrs (f07jec)
real tridiagonal matrix	nag_dgttrs (f07cec)
expert drivers (with condition and error estimation):	
complex band matrix	nag_zgbsvx (f07bpc)
complex Hermitian indefinite matrix	nag_zhesvx (f07mpc)
complex Hermitian indefinite matrix, packed storage	nag_zhpsvx (f07ppc)
complex Hermitian positive definite band matrix	nag_zpbsvx (f07hpc)
complex Hermitian positive definite matrix	nag_zposvx (f07fpc)
complex Hermitian positive definite matrix, packed storage	nag_zppsvox (f07gpc)
complex Hermitian positive definite tridiagonal matrix	nag_zptsvox (f07jpc)
complex matrix	nag_zgesvx (f07apc)
complex symmetric indefinite matrix	nag_zsysvx (f07npc)
complex symmetric indefinite matrix, packed storage	nag_zspsvx (f07qpc)
complex tridiagonal matrix	nag_zgtsvx (f07cpc)

real band matrix	nag_dgbsvx (f07bbc)
real matrix	nag_dgesvx (f07abc)
real symmetric indefinite matrix	nag_dsylvx (f07mbc)
real symmetric indefinite matrix, packed storage	nag_dspsvx (f07pbc)
real symmetric positive definite band matrix	nag_dpbsvx (f07hbc)
real symmetric positive definite matrix	nag_dposvx (f07fbc)
real symmetric positive definite matrix, packed storage	nag_dpssvx (f07gbc)
real symmetric positive definite tridiagonal matrix	nag_dptsvx (f07jbc)
real tridiagonal matrix	nag_dgtsvx (f07cbc)
simple drivers,	
complex band matrix	nag_zgbsv (f07bnc)
complex Hermitian indefinite matrix	nag_zhesv (f07mnc)
complex Hermitian indefinite matrix, packed storage	nag_zhpsv (f07pnc)
complex Hermitian positive definite band matrix	nag_zpbsv (f07hnc)
complex Hermitian positive definite matrix	nag_zposv (f07fnc)
complex Hermitian positive definite matrix, packed storage	nag_zppsv (f07gnc)
complex Hermitian positive definite tridiagonal matrix	nag_zptsv (f07jnc)
complex matrix	nag_zgesv (f07anc)
complex matrix, using mixed precision	nag_zcgesv (f07aqc)
complex symmetric indefinite matrix	nag_zsysv (f07nnc)
complex symmetric indefinite matrix, packed storage	nag_zspsv (f07qnc)
complex triangular band matrix	nag_ztbtrs (f07vsc)
complex triangular matrix	nag_ztrtrs (f07tsc)
complex triangular matrix, packed storage	nag_ztptrs (f07usc)
complex tridiagonal matrix	nag_zgtsv (f07cnc)
real band matrix	nag_dgbsv (f07bac)
real matrix	nag_dgesv (f07aac)
real matrix, using mixed precision	nag_dsgesv (f07acc)
real symmetric indefinite matrix	nag_dsylv (f07mac)
real symmetric indefinite matrix, packed storage	nag_dspsv (f07pac)
real symmetric positive definite band matrix	nag_dpbsv (f07hac)
real symmetric positive definite matrix	nag_dposv (f07fac)
real symmetric positive definite matrix, packed storage	nag_dpssv (f07gac)
real symmetric positive definite tridiagonal matrix	nag_dptsv (f07jac)
real triangular band matrix	nag_dtbtrs (f07vec)
real triangular matrix	nag_dtrtrs (f07tec)
real triangular matrix, packed storage	nag_dtptrs (f07uec)
real tridiagonal matrix	nag_dgtsv (f07cac)

5 Auxiliary Functions Associated with Library Function Arguments

None.

6 Functions Withdrawn or Scheduled for Withdrawal

None.

7 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
