Module 5.2: nag_sym_lin_sys
Symmetric Systems of Linear Equations

`nag_sym_lin_sys` provides a procedure for solving real or complex, symmetric or Hermitian systems of linear equations with one or many right-hand sides:

$Ax = b$ or $AX = B$.

It also provides procedures for factorizing A and solving a system of equations when the matrix A has already been factorized. Positive definite matrices are treated as a special case.

Contents

Introduction

1 Notation and Background

We use the following notation for a system of linear equations:

$Ax = b$, if there is one right-hand side b;

$AX = B$, if there are many right-hand sides (the columns of the matrix B).

In this module, the matrix A (the *coefficient matrix*) is assumed to be *real symmetric*, *complex Hermitian* or *complex symmetric*. The procedures take advantage of this in order to economize on the work and storage required. If A is real symmetric or complex Hermitian, it may also be *positive definite*, and the procedures can take advantage of this property if it is known, to make further savings in work and to achieve greater reliability.

The module provides options to return *forward* or *backward error bounds* on the computed solution. It also provides options to evaluate the *determinant* of A and to estimate the *condition number* of A, which is a measure of the sensitivity of the computed solution to perturbations of the original data or to rounding errors in the computation. For more details on error analysis, see the Chapter Introduction.

To solve the system of equations, the first step is to factorize A, using

the *Cholesky* factorization if A is known to be positive definite;

the *Bunch–Kaufman* factorization otherwise.

The system of equations can then be solved by forward and backward substitution.

2 Choice of Procedures

The procedure `nag_sym_lin_sol` should be suitable for most purposes; it performs the factorization of A and solves the system of equations in a single call. It also has options to estimate the condition number of A, and to return forward and backward error bounds on the computed solution.

The module also provides lower-level procedures which perform the two computational steps in the solution process:

`nag_sym_lin_fac` computes a factorization of A, with options to evaluate the determinant and to estimate the condition number;

`nag_sym_lin_sol_fac` solves the system of equations, assuming that A has already been factorized by a call to `nag_sym_lin_fac`. It has options to return forward and backward error bounds on the solution.

These lower-level procedures are intended for more experienced users. For example, they enable a factorization computed by `nag_sym_lin_fac` to be reused several times in repeated calls to `nag_sym_lin_sol_fac`.

3 Storage of Matrices

The procedures in this module allow a choice of storage schemes for the symmetric or Hermitian matrix A: conventional storage or packed storage. The choice is determined by the rank of the corresponding argument `a`.

3.1 Conventional Storage

`a` is a rank-2 array, of shape (n,n). Matrix element a_{ij} is stored in `a(`i,j`)`. Only the elements of either the upper or the lower triangle need be stored, as specified by the argument `uplo`; the remaining elements of `a` need not be set.

This storage scheme is more straightforward and carries less risk of user error than packed storage; on some machines it may result in more efficient execution. It requires almost twice as much memory as packed storage, although the other triangle of `a` may be used to store other data.

3.2 Packed Storage

`a` is a rank-1 array of shape $(n(n+1)/2)$. The elements of either the upper or the lower triangle of A, as specified by `uplo`, are packed by columns into contiguous elements of `a`.

Packed storage is more economical in use of memory than conventional storage, but may result in less efficient execution on some machines.

The details of packed storage are as follows:

- if `uplo` = `'u'` or `'U'`, a_{ij} is stored in `a(`$i+j(j-1)/2$`)`, for $i \leq j$;
- if `uplo` = `'l'` or `'L'`, a_{ij} is stored in `a(`$i+(2n-j)(j-1)/2$`)`, for $i \geq j$.

For example

`uplo`	Hermitian Matrix	Packed storage in array `a`
`'u'` or `'U'`	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \overline{a}_{12} & a_{22} & a_{23} & a_{24} \\ \overline{a}_{13} & \overline{a}_{23} & a_{33} & a_{34} \\ \overline{a}_{14} & \overline{a}_{24} & \overline{a}_{34} & a_{44} \end{pmatrix}$	a_{11} $\underbrace{a_{12}\,a_{22}}$ $\underbrace{a_{13}\,a_{23}\,a_{33}}$ $\underbrace{a_{14}\,a_{24}\,a_{34}\,a_{44}}$
`'l'` or `'L'`	$\begin{pmatrix} a_{11} & \overline{a}_{21} & \overline{a}_{31} & \overline{a}_{41} \\ a_{21} & a_{22} & \overline{a}_{32} & \overline{a}_{42} \\ a_{31} & a_{32} & a_{33} & \overline{a}_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}\,a_{21}\,a_{31}\,a_{41}}$ $\underbrace{a_{22}\,a_{32}\,a_{42}}$ $\underbrace{a_{33}\,a_{43}}$ a_{44}

Note that for 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 Hermitian matrices, packing the upper triangle by columns is equivalent to packing the conjugate of the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the conjugate of the upper triangle by rows.

Procedure: nag_sym_lin_sol

1 Description

nag_sym_lin_sol is a generic procedure which computes the solution of a system of linear equations with one or many right-hand sides, where the matrix of coefficients may be

real symmetric indefinite,

complex Hermitian indefinite,

complex symmetric,

real symmetric positive definite, or

complex Hermitian positive definite.

Here the term *indefinite* means a matrix that is not *known* to be positive definite, although it may in fact be so.

We write:

$Ax = b$, if there is one right-hand side b;

$AX = B$, if there are many right-hand sides (the columns of the matrix B).

The procedure allows conventional or packed storage for A.

The procedure also has options to return an estimate of the *condition number* of A, and *forward* and *backward error bounds* for the computed solution or solutions. See the Chapter Introduction for an explanation of these terms. If error bounds are requested, the procedure performs iterative refinement of the computed solution in order to guarantee a small backward error.

2 Usage

```
USE nag_sym_lin_sys
CALL nag_sym_lin_sol(nag_key, uplo, a, b  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the 24 combinations of the following cases:

Symmetric indefinite / Hermitian indefinite / positive definite matrix

Symmetric indefinite:	nag_key = nag_key_sym.
Hermitian indefinite:	nag_key = nag_key_herm; for real matrices this is equivalent to nag_key_sym.
Positive definite:	nag_key = nag_key_pos.

Real / complex data

Real data:	a and b are of type real(kind=*wp*).
Complex data:	a and b are of type complex(kind=*wp*).

One / many right-hand sides

One r.h.s.:	b is a rank-1 array, and the optional arguments bwd_err and fwd_err are scalars.
Many r.h.s.:	b is a rank-2 array, and the optional arguments bwd_err and fwd_err are rank-1 arrays.

Conventional / packed storage (see the Module Introduction)

Conventional: `a` is a rank-2 array.

Packed: `a` is a rank-1 array.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

n — the order of the matrix A

r — the number of right-hand sides

3.1 Mandatory Arguments

nag_key — a "key" argument, intent(in)

Input: must have one of the following values (which are named constants, each of a different derived type, defined by the Library, and accessible from this module).

`nag_key_sym`: if the matrix A is real symmetric indefinite or complex symmetric;

`nag_key_herm`: if the matrix A is real or complex Hermitian indefinite;

`nag_key_pos`: if the matrix A is real symmetric positive definite or complex Hermitian positive definite.

For further explanation of "key" arguments, see the Essential Introduction.

Note: for real matrices, `nag_key_herm` is equivalent to `nag_key_sym`.

uplo — character(len=1), intent(in)

Input: specifies whether the upper or lower triangle of A is supplied, and whether the factorization involves an upper triangular matrix U or a lower triangular matrix L.

If `uplo` = `'u'` or `'U'`, the upper triangle is supplied, and is overwritten by an upper triangular factor U;

if `uplo` = `'l'` or `'L'`, the lower triangle is supplied, and is overwritten by a lower triangular factor L.

Constraints: `uplo` = `'u'`, `'U'`, `'l'` or `'L'`.

a(n,n) **/ a**$(n(n+1)/2)$ — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the matrix A.

Conventional storage (`a` has shape (n,n))

If `uplo` = `'u'`, the upper triangle of A must be stored, and elements below the diagonal need not be set;

if `uplo` = `'l'`, the lower triangle of A must be stored, and elements above the diagonal need not be set.

Packed storage (`a` has shape $(n(n+1)/2)$)

If `uplo` = `'u'`, the upper triangle of A must be stored, packed by columns, with a_{ij} in `a(`$i+j(j-1)/2$`)` for $i \leq j$;

if `uplo` = `'l'`, the lower triangle of A must be stored, packed by columns, with a_{ij} in `a(`$i+(2n-j)(j-1)/2$`)` for $i \geq j$.

Output: the supplied triangle of A is overwritten by details of the factorization; the other elements of `a` are unchanged.

Constraints: if A is complex Hermitian, its diagonal elements must have zero imaginary parts.

b(n) / **b**(n, r) — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the right-hand side vector b or matrix B.

Output: overwritten on exit by the solution vector x or matrix X.

Constraints: `b` must be of the same type as `a`.

Note: if optional error bounds are requested then the solution returned is that computed by iterative refinement.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

bwd_err / bwd_err(r) — real(kind=*wp*), intent(out), optional

Output: if `bwd_err` is a scalar, it returns the component wise backward error bound for the single solution vector x. Otherwise, `bwd_err(`i`)` returns the component wise backward error bound for the ith solution vector, returned in the ith column of `b`, for $i = 1, 2, \ldots, r$.

Constraints: if `b` has rank 1, `bwd_err` must be a scalar; if `b` has rank 2, `bwd_err` must be a rank-1 array.

fwd_err / fwd_err(r) — real(kind=*wp*), intent(out), optional

Output: if `fwd_err` is a scalar, it returns an estimated bound for the forward error in the single solution vector x. Otherwise, `fwd_err(`i`)` returns an estimated bound for the forward error in the ith solution vector, returned in the ith column of `b`, for $i = 1, 2, \ldots, r$.

Constraints: if `b` has rank 1, `fwd_err` must be a scalar; if `b` has rank 2, `fwd_err` must be a rank-1 array.

rcond — real(kind=*wp*), intent(out), optional

Output: an estimate of the reciprocal of the condition number of A, $\kappa_\infty(A) (= \kappa_1(A)$ for A symmetric or Hermitian). `rcond` is set to zero if exact singularity is detected or the estimate underflows. If `rcond` is less than `EPSILON(1.0_`*wp*`)`, then A is singular to working precision.

pivot(n) — integer, intent(out), optional

Output: the pivot indices used in the Bunch–Kaufman factorization; see `nag_sym_lin_fac` for details. If `nag_key` = `nag_key_pos` (Cholesky factorization), `pivot` is not needed but, if it is present, it is set to the vector $(1, 2, \ldots, n)$.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Singular matrix. This error can only occur if nag_key = nag_key_sym or nag_key_herm. The Bunch–Kaufman factorization has been completed, but the factor D has a zero diagonal block of order 1, and so is exactly singular. No solutions or error bounds are computed.
202	Matrix not positive definite. This error can only occur if nag_key = nag_key_pos. The Cholesky factorization cannot be completed. Either A is close to singularity, or it has at least one negative eigenvalue. No solutions or error bounds are computed.

Warnings (error%level = 1):

error%code	Description
101	Approximately singular matrix. The estimate of the reciprocal condition number (returned in rcond if present) is less than or equal to EPSILON(1.0_*wp*). The matrix is singular to working precision, and it is likely that the computed solution returned in b has no accuracy at all. You should examine the forward error bounds returned in fwd_err, if present.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 2 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The procedure first calls nag_sym_lin_fac to factorize A, and to estimate the condition number. It then calls nag_sym_lin_sol_fac to compute the solution to the system of equations, and, if required, the error bounds. See the documents for those procedures for more details, and Chapter 4 of Golub and Van Loan [2] for background. The algorithms are derived from LAPACK (see Anderson *et al.* [1]).

6.2 Accuracy

The accuracy of the computed solution is given by the forward and backward error bounds which are returned in the optional arguments fwd_err and bwd_err.

The backward error bound bwd_err is rigorous; the forward error bound fwd_err is an estimate, but is almost always satisfied.

The condition number $\kappa_\infty(A)$ gives a general measure of the *sensitivity* of the solution of $Ax = b$, either to uncertainties in the data or to rounding errors in the computation. An estimate of the reciprocal of $\kappa_\infty(A)$ is returned in the optional argument rcond. However, forward error bounds derived using this condition number may be more pessimistic than the bounds returned in fwd_err, if present.

6.3 Timing

The time taken is roughly proportional to n^3, and, if there are only a few right-hand sides, is roughly half that taken by the procedure nag_gen_lin_sol in the module nag_gen_lin_sys (5.1) which does not take advantage of symmetry. The time taken for complex data is about 4 times as long as that for real data.

The procedure is somewhat faster, especially on high-performance computers, when nag_key is set to nag_key_pos (assuming that A is indeed positive definite).

Procedure: nag_sym_lin_fac

1 Description

nag_sym_lin_fac is a generic procedure which factorizes a real or complex, symmetric or Hermitian matrix A of order n.

If A is *indefinite* (that is, not known to be positive definite), the procedure computes a Bunch–Kaufman factorization:

$A = PUDU^TP^T$ or $A = PLDL^TP^T$, if A is real or complex symmetric;
$A = PUDU^HP^T$ or $A = PLDL^HP^T$, if A is complex Hermitian;

where U is upper triangular, L is lower triangular, P is a permutation matrix, and D is a symmetric or Hermitian block diagonal matrix, with diagonal blocks of order 1 or 2.

If A is real symmetric or complex Hermitian and also *positive definite*, the procedure computes a *Cholesky* factorization (which is simpler and somewhat more efficient than the Bunch–Kaufman):

$A = U^TU$ or $A = LL^T$, if A is real symmetric;
$A = U^HU$ or $A = LL^H$, if A is complex Hermitian;

where U is upper triangular and L is lower triangular.

This procedure can also return the determinant of A and an estimate of the condition number $\kappa_\infty(A)$ $(= \kappa_1(A))$.

2 Usage

```
USE nag_sym_lin_sys
CALL nag_sym_lin_fac(nag_key, uplo, a, pivot  [, optional arguments])
```

or for positive definite matrices only:

```
CALL nag_sym_lin_fac(nag_key, uplo, a  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the 16 combinations of the following cases:

Symmetric indefinite / Hermitian indefinite / positive definite matrix

For positive definite matrices, two forms of the interface are provided: the first *includes* pivot as a mandatory argument for compatibility with the interface for indefinite matrices; the second *omits* pivot since it is not needed for Cholesky factorization.

Symmetric indefinite: nag_key = nag_key_sym.

Hermitian indefinite: nag_key = nag_key_herm; for real matrices this is equivalent to nag_key_sym.

positive definite (1): nag_key = nag_key_pos, with pivot as a mandatory argument.

positive definite (2): nag_key = nag_key_pos, with pivot not in the argument list.

Real / complex data

Real data: a is of type real(kind=*wp*).

Complex data: a is of type complex(kind=*wp*).

Conventional / packed storage (see the Module Introduction)

Conventional: `a` is a rank-2 array.

Packed: `a` is a rank-1 array.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

n — the order of the matrix A

3.1 Mandatory Arguments

nag_key — a "key" argument, intent(in)

Input: must have one of the following values (which are named constants, each of a different derived type, defined by the Library, and accessible from this module).

`nag_key_sym`: if the matrix A is real symmetric indefinite or complex symmetric;

`nag_key_herm`: if the matrix A is real symmetric indefinite or complex Hermitian indefinite;

`nag_key_pos`: if the matrix A is real symmetric positive definite or complex Hermitian positive definite.

For further explanation of "key" arguments, see the Essential Introduction.

Note: for real matrices, `nag_key_herm` is equivalent to `nag_key_sym`.

uplo — character(len=1), intent(in)

Input: specifies whether the upper or lower triangle of A is supplied, and whether the factorization involves an upper triangular matrix U or a lower triangular matrix L.

If `uplo` = `'u'` or `'U'`, the upper triangle is supplied, and is overwritten by an upper triangular factor U;

if `uplo` = `'l'` or `'L'`, the lower triangle is supplied, and is overwritten by a lower triangular factor L.

Constraints: `uplo` = `'u'`, `'U'`, `'l'` or `'L'`.

a(n,n) **/ a**$(n(n+1)/2)$ — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the matrix A.

Conventional storage (`a` has shape (n,n))

If `uplo` = `'u'`, the upper triangle of A must be stored, and elements below the diagonal need not be set;

if `uplo` = `'l'`, the lower triangle of A must be stored, and elements above the diagonal need not be set.

Packed storage (`a` has shape $(n(n+1)/2)$)

If `uplo` = `'u'`, the upper triangle of A must be stored, packed by columns, with a_{ij} in `a(`$i+j(j-1)/2$`)` for $i \leq j$;

if `uplo` = `'l'`, the lower triangle of A must be stored, packed by columns, with a_{ij} in `a(`$i+(2n-j)(j-1)/2$`)` for $i \geq j$.

Output: the supplied triangle of A is overwritten by details of the factorization; the other elements of `a` are unchanged.

Constraints: if A is complex Hermitian, its diagonal elements must have zero imaginary parts.

pivot(n) — integer, intent(out)

Output: the pivot indices used in the Bunch–Kaufman factorization. See Section 6.1 for details.

Note: if `nag_key` = `nag_key_pos` (Cholesky factorization), `pivot` need not be included in the argument list, but if it is included, it is set to the vector $(1, 2, \ldots, n)$.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

rcond — real(kind=*wp*), intent(out), optional

Output: an estimate of the reciprocal of the condition number of A, $\kappa_\infty(A)(= \kappa_1(A)$ for A symmetric or Hermitian). `rcond` is set to zero if exact singularity is detected or the estimate underflows. If `rcond` is less than `EPSILON(1.0_`*wp*`)`, then A is singular to working precision.

det_frac — real(kind=*wp*) / complex(kind=*wp*), intent(out), optional

det_exp — integer, intent(out), optional

Output: `det_frac` returns the fractional part f, and `det_exp` returns the exponent e, of the determinant of A expressed as $f.b^e$, where b is the base of the representation of the floating point numbers (given by `RADIX(1.0_`*wp*`)`), or as `SCALE` (`det_frac`,`det_exp`). The determinant is returned in this form to avoid the risk of overflow or underflow.

Constraints: `det_frac` *must be* of type complex(kind=*wp*) if `a` is of type complex(kind=*wp*) and `nag_key` is set to `nag_key_sym`, otherwise `det_frac` is of type real(kind=*wp*). If either `det_frac` or `det_exp` is present the other must also be present.

inertia(3) — integer, intent(out), optional

Output: `inertia` returns the inertia of the matrix A. The inertia of a real symmetric or complex Hermitian matrix is defined by the number of positive, negative and zero eigenvalues of the matrix. The three elements of `inertia` are:

`inertia` (1) contains the number of positive eigenvalues of `a`;

`inertia` (2) contains the number of negative eigenvalues of `a`;

`inertia` (3) contains the number of zero eigenvalues of `a`.

Note: the inertia of a complex symmetric matrix is not defined. For such a matrix all three elements of `inertia` are set to 0.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
305	Invalid absence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Singular matrix. This error can only occur if nag_key = nag_key_sym or nag_key_herm. The Bunch–Kaufman factorization has been completed, but the factor D has a zero diagonal block of order 1, and so is exactly singular. If the factorization is used to solve a system of linear equations, an error will occur.
202	Matrix not positive definite. This error can only occur if nag_key = nag_key_pos. The Cholesky factorization cannot be completed. Either A is close to singularity, or it has at least one negative eigenvalue. If the factorization is used to solve a system of linear equations, an error will occur.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 3 of this module document.

6 Further Comments

6.1 Algorithmic Detail

If nag_key = nag_key_pos (A is positive definite), the procedure performs a Cholesky factorization of A:

$A = U^H U$, with U upper triangular, if uplo = 'u';

$A = LL^H$, with L lower triangular, if uplo = 'l'.

See Section 4.2 of Golub and Van Loan [2].

Otherwise, it performs a Bunch–Kaufman factorization with diagonal pivoting:

$A = PUDU^T P^T$ (or $PUDU^H P^T$ if A is Hermitian), with U unit upper triangular, if uplo = 'u';

$A = PLDL^T P^T$ (or $PLDL^H P^T$ if A is Hermitian), with L unit lower triangular, if uplo = 'l'.

P is a permutation matrix, and D is a symmetric or Hermitian block diagonal matrix with diagonal blocks of order 1 or 2; U or L has unit diagonal blocks of order 2 corresponding to the 2×2 blocks of D. See Section 4.4 of Golub and Van Loan [2].

If the Bunch–Kaufman factorization is performed on a matrix which is in fact positive definite, no interchanges are performed, and no diagonal blocks of order 2 occur in D; thus, A is factorized as $U^H DU$ or LDL^H, with D being a simple diagonal matrix with positive diagonal elements.

In the Bunch–Kaufman factorization, the argument pivot is used to record details of the interchanges and the structure of D, as follows.

If pivot(i) $= k > 0$, then d_{ii} is a 1×1 block, and the ith row and column were interchanged with the kth row and column.

If uplo = 'u', and pivot($i-1$) = pivot(i) $= -k < 0$, then the $(i-1)$th row and column were interchanged with the kth row and column, and D has a 2×2 block in rows and columns $i-1$ and i, of the form $\begin{pmatrix} d_{i-1,i-1} & d_{i-1,i} \\ d_{i-1,i} & d_{ii} \end{pmatrix}$ if symmetric, or $\begin{pmatrix} d_{i-1,i-1} & d_{i-1,i} \\ \overline{d}_{i-1,i} & d_{ii} \end{pmatrix}$ if Hermitian. The elements of the upper triangle of D overwrite the corresponding elements of A; the corresponding elements of U are either 1 or 0, and are not stored.

If uplo = 'l', and pivot(i) = pivot($i+1$) $= -k < 0$, then the $(i+1)$th row and column were interchanged with the kth row and column, and D has a 2×2 block in rows and columns i and

$i+1$, of the form $\begin{pmatrix} d_{ii} & d_{i+1,i} \\ d_{i+1,i} & d_{i+1,i+1} \end{pmatrix}$ if symmetric, or $\begin{pmatrix} d_{ii} & \overline{d}_{i+1,i} \\ d_{i+1,i} & d_{i+1,i+1} \end{pmatrix}$ if Hermitian. The elements of the lower triangle of D overwrite the corresponding elements of A; the corresponding elements of L are either 1 or 0, and are not stored.

To give a simple example, suppose $n = 4$, `uplo = 'u'`, A is Hermitian, and D has a 2×2 block in rows 2 and 3: then U and D have the forms

$$U = \begin{pmatrix} 1 & u_{12} & u_{13} & u_{14} \\ & 1 & 0 & u_{24} \\ & & 1 & u_{34} \\ & & & 1 \end{pmatrix} \quad D = \begin{pmatrix} d_{11} & & & \\ & d_{22} & d_{23} & \\ & \overline{d}_{23} & d_{33} & \\ & & & d_{44} \end{pmatrix};$$

on exit from this procedure, `pivot(1)` > 0, `pivot(2)` $=$ `pivot(3)` < 0, and `pivot(4)` > 0; if `a` is a rank-2 array, its upper triangle holds:

$$\begin{matrix} d_{11} & u_{12} & u_{13} & u_{14} \\ & d_{22} & d_{23} & u_{24} \\ & & d_{33} & u_{34} \\ & & & d_{44} \end{matrix}$$

To estimate the condition number $\kappa_\infty(A)$ $(= \kappa_1(A) = \|A\|_1\|A^{-1}\|_1)$, the procedure first computes $\|A\|_1$ directly, and then uses Higham's modification of Hager's method (see Higham [3]) to estimate $\|A^{-1}\|_1$. The procedure returns the reciprocal $\rho = 1/\kappa_\infty(A)$, rather than $\kappa_\infty(A)$ itself.

The algorithms are derived from LAPACK (see Anderson *et al.* [1]).

6.2 Accuracy

If a Cholesky factorization is performed with `uplo = 'u'`, the computed factor U is the exact factor of a perturbed matrix $A + E$, such that

$$|E| \leq c(n)\epsilon |U^H||U|,$$

where $c(n)$ is a modest linear function of n, and ϵ = `EPSILON(1.0_wp)`. If `uplo = 'l'`, a similar statement holds for the computed factor L. It follows that in both cases $|e_{ij}| \leq c(n)\epsilon\sqrt{a_{ii}a_{jj}}$.

If a Bunch–Kaufman factorization is performed with `uplo = 'u'`, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, such that

$$|E| \leq c(n)\epsilon P|U||D||U^T|P^T,$$

where $c(n)$ is a modest linear function of n, and ϵ = `EPSILON(1.0_wp)`. If `uplo = 'l'`, a similar statement holds for the computed factors L and D.

The computed estimate `rcond` is never less than the true value ρ, and in practice is nearly always less than 10ρ (although examples can be constructed where the computed estimate is much larger).

Since $\rho = 1/\kappa(A)$, this means that the procedure never overestimates the condition number, and hardly ever underestimates it by more than a factor of 10.

6.3 Timing

The total number of floating-point operations required for either the Cholesky or the Bunch–Kaufman factorization is roughly $(1/3)n^3$ for real A, and $(4/3)n^3$ for complex A. The Cholesky factorization is somewhat more efficient, especially on high-performance computers.

Estimating the condition number involves solving a number of systems of linear equations with A or A^T as the coefficient matrix; the number is usually 4 or 5 and never more than 11. Each solution involves approximately $2n^2$ floating-point operations if A is real, or $8n^2$ if A is complex. Thus, for large n, the cost is much less than that of directly computing A^{-1} and its norm, which would require $O(n^3)$ operations.

Procedure: nag_sym_lin_sol_fac

1 Description

nag_sym_lin_sol_fac is a generic procedure which computes the solution of a real or complex, symmetric or Hermitian system of linear equations with one or many right-hand sides, assuming that the coefficient matrix has already been factorized by nag_sym_lin_fac.

We write:

$Ax = b$, if there is one right-hand side b;

$AX = B$, if there are many right-hand sides (the columns of the matrix B).

The matrix A (the *coefficient matrix*) may be:

real symmetric indefinite,

complex Hermitian indefinite,

complex symmetric,

real symmetric positive definite, or

complex Hermitian positive definite,

Here the term *indefinite* means a matrix that is not *known* to be positive definite, although it may in fact be so.

The procedure also has options to return *forward* and *backward error bounds* for the computed solution or solutions.

2 Usage

```
USE nag_sym_lin_sys
CALL nag_sym_lin_sol_fac(nag_key, uplo, a_fac, pivot, b  [, optional arguments])
```

or for positive definite matrices only:

```
CALL nag_sym_lin_sol_fac(nag_key, uplo, a_fac, b  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for each of the 32 combinations of the following cases:

Symmetric indefinite / Hermitian indefinite / positive definite matrix

For positive definite matrices, two forms of the interface are provided: the first *includes* pivot as a mandatory argument for compatibility with the interface for indefinite matrices; the second *omits* pivot since it is not needed for Cholesky factorization.

Symmetric indefinite: nag_key = nag_key_sym.
Hermitian indefinite: nag_key = nag_key_herm; for real matrices this is equivalent to nag_key_sym.
positive definite (1): nag_key = nag_key_pos, with pivot as a mandatory argument.
positive definite (2): nag_key = nag_key_pos, with pivot not in the argument list.

Real / complex data

Real data: a_fac, b and the optional argument a are of type real(kind=*wp*).

Complex data: a_fac, b and the optional argument a are of type complex(kind=*wp*).

One / many right-hand sides

One r.h.s.: b is a rank-1 array, and the optional arguments bwd_err and fwd_err are scalars.

Many r.h.s.: b is a rank-2 array, and the optional arguments bwd_err and fwd_err are rank-1 arrays.

Conventional / packed storage (see the Module Introduction)

Conventional: a_fac and the optional argument a are rank-2 arrays.

Packed: a_fac and the optional argument a are rank-1 arrays.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

n — the order of the matrix A

r — the number of right-hand sides

3.1 Mandatory Arguments

nag_key — a "key" argument, intent(in)

Input: must have one of the following values (which are named constants, each of a different derived type, defined by the Library, and accessible from this module).

nag_key_sym: if the matrix A is real symmetric indefinite or complex symmetric;

nag_key_herm: if the matrix A is real or complex Hermitian indefinite;

nag_key_pos: if the matrix A is real symmetric positive definite or complex Hermitian positive definite.

For further explanation of "key" arguments, see the Essential Introduction.

Note: for real matrices, nag_key_herm is equivalent to nag_key_sym.

uplo — character(len=1), intent(in)

Input: specifies whether the upper or lower triangle of A was supplied to nag_sym_lin_fac, and whether the factorization involves an upper triangular matrix U or a lower triangular matrix L.

If uplo = 'u' or 'U', the upper triangle was supplied, and was overwritten by an upper triangular factor U;

if uplo = 'l' or 'L', the lower triangle was supplied, and was overwritten by a lower triangular factor L.

Constraints: uplo = 'u', 'U', 'l' or 'L'.

Note: the value of uplo must be the same as in the preceding call to nag_sym_lin_fac.

a_fac(n, n) **/ a_fac**($n(n+1)/2$) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the factorization of A, as returned by nag_sym_lin_fac.

pivot(n) — integer, intent(in)

Input: the pivot indices, as returned by nag_sym_lin_fac.

Note: if nag_key = nag_key_pos, pivot need not be included in the argument list.

b(n) / **b**(n, r) — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the right-hand side vector b or matrix B.

Output: overwritten on exit by the solution vector x or matrix X.

Constraints: `b` must be of the same type as `a_fac`.

Note: if optional error bounds are requested then the solution returned is that computed by iterative refinement.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

bwd_err / **bwd_err**(r) — real(kind=*wp*), intent(out), optional

Output: if `bwd_err` is a scalar, it returns the component wise backward error bound for the single solution vector x. Otherwise, `bwd_err(`i`)` returns the component wise backward error bound for the ith solution vector, returned in the ith column of `b`, for $i = 1, 2, \ldots, r$.

Constraints: if `bwd_err` is present, the original matrix A must be supplied in `a`; if `b` has rank 1, `bwd_err` must be a scalar; if `b` has rank 2, `bwd_err` must be a rank-1 array.

fwd_err / **fwd_err**(r) — real(kind=*wp*), intent(out), optional

Output: if `fwd_err` is a scalar, it returns an estimated bound for the forward error in the single solution vector x. Otherwise, `fwd_err(`i`)` returns an estimated bound for the forward error in the ith solution vector, returned in the ith column of `b`, for $i = 1, 2, \ldots, r$.

Constraints: if `fwd_err` is present, the original matrix A must be supplied in `a`; if `b` has rank 1, `fwd_err` must be a scalar; if `b` has rank 2, `fwd_err` must be a rank-1 array.

a(n, n) / **a**($n(n+1)/2$) — real(kind=*wp*) / complex(kind=*wp*), intent(in), optional

Input: the original coefficient matrix A, as supplied to `nag_sym_lin_fac`.

Constraints: `a` must be present if either `bwd_err` or `fwd_err` is present; `a` must be of the same type and rank as `a_fac`.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
305	Invalid absence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Singular matrix. This error can only occur if nag_key = nag_key_sym or nag_key_herm. In the Bunch–Kaufman factorization supplied in a_fac, the factor D has a zero diagonal block of order 1, and so is exactly singular. No solutions or error bounds are computed.
202	Matrix not positive definite. This error can only occur if nag_key = nag_key_pos. The supplied array a_fac does not contain a valid Cholesky factorization, indicating that the original matrix A was not positive definite. No solutions or error bounds are computed.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 3 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The solution x is computed by forward and backward substitution. Assuming uplo = 'u':

if nag_key = nag_key_pos (Cholesky factorization), $U^H y = b$ is solved for y, and then $Ux = b$ is solved for x;

otherwise (Bunch–Kaufman factorization), $PUDy = b$ is solved for y, and then $U^T P^T x = y$ is solved for x if nag_key = nag_key_sym, or $U^H P^T x = y$ if nag_key = nag_key_herm.

A similar method is used if uplo = 'l'.

If error bounds are requested (that is, fwd_err or bwd_err is present), iterative refinement of the solution is performed (in working precision), to reduce the backward error as far as possible.

The algorithms are derived from LAPACK (see Anderson *et al.* [1]).

6.2 Accuracy

The accuracy of the computed solution is given by the forward and backward error bounds which are returned in the optional arguments fwd_err and bwd_err.

The backward error bound bwd_err is rigorous; the forward error bound fwd_err is an estimate, but is almost always satisfied.

For each right-hand side b, the computed solution $\hat{x}$ is the exact solution of a perturbed system of equations $(A + E)\hat{x} = b$. Assuming uplo = 'u':

with a Cholesky factorization

$$|E| \leq c(n)\epsilon |U^H||U|$$

with a Bunch–Kaufman factorization

$$|E| \leq c(n)\epsilon P|U||D||U^H|P^T$$

where $c(n)$ is a modest linear function of n, and ϵ = EPSILON(1.0_wp).

The condition number $\kappa_\infty(A)$ gives a general measure of the *sensitivity* of the solution of $Ax = b$, either to uncertainties in the data or to rounding errors in the computation. An estimate of the reciprocal of $\kappa_\infty(A)$ is returned by nag_sym_lin_fac in its optional argument rcond. However, forward error bounds

derived using this condition number may be more pessimistic than the bounds returned in `fwd_err`, if present.

If the reciprocal of the condition number $\leq$ `EPSILON(1.0_`*wp*`)`, then A is singular to working precision; if the factorization is used to solve a system of linear equations, the computed solution may have no meaningful accuracy and should be treated with great caution.

6.3 Timing

The number of real floating-point operations required to compute the solutions is roughly $2n^2r$ if A is real, and $8n^2r$ if A is complex.

To compute the error bounds `fwd_err` and `bwd_err` usually requires about 5 times as much work.

Example 1: Solution of a real symmetric indefinite system of linear equations

Solve a real symmetric system of linear equations with one right-hand side $Ax = b$, also estimating the condition number of A, and forward and backward error bounds on the computed solutions. A is not known to be positive definite. This example calls the single procedure `nag_sym_lin_sol`, using conventional storage for A.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_sym_lin_sys_ex01

  ! Example Program Text for nag_sym_lin_sys
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sym_lin_sys, ONLY : nag_key_sym, nag_sym_lin_sol
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n
  REAL (wp) :: bwd_err, fwd_err, rcond
  CHARACTER (1) :: uplo
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: a(:,:), b(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_sym_lin_sys_ex01'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n
  READ (nag_std_in,*) uplo

  ALLOCATE (a(n,n),b(n))       ! Allocate storage

  SELECT CASE (uplo)
  CASE ('L','l')
    READ (nag_std_in,*) (a(i,:i),i=1,n)
  CASE ('U','u')
    READ (nag_std_in,*) (a(i,i:),i=1,n)
  END SELECT

  READ (nag_std_in,*) b

  ! Solve the system of equations

  CALL nag_sym_lin_sol(nag_key_sym,uplo,a,b,bwd_err=bwd_err, &
   fwd_err=fwd_err,rcond=rcond)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,'(1X,''kappa(A) (1/rcond)''/2X,ES11.2)') 1/rcond
  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Solution'
```

```
      WRITE (nag_std_out,'(4X,F9.4)') b
      WRITE (nag_std_out,*)
      WRITE (nag_std_out,*) 'Backward error bound'
      WRITE (nag_std_out,'(2X,ES11.2)') bwd_err
      WRITE (nag_std_out,*)
      WRITE (nag_std_out,*) 'Forward error bound (estimate)'
      WRITE (nag_std_out,'(2X,ES11.2)') fwd_err

      DEALLOCATE (a,b)             ! Deallocate storage

    END PROGRAM nag_sym_lin_sys_ex01
```

2 Program Data

```
Example Program Data for nag_sym_lin_sys_ex01
  4                          : Value of n
  'U'                        : Value of uplo
  2.07   3.87   4.20  -1.15
        -0.21   1.87   0.63
                1.15   2.06
                      -1.81 : End of Matrix A (upper triangle)
 -9.50
 -8.38
 -6.07
 -0.96                       : End of right-hand side vector b
```

3 Program Results

```
 Example Program Results for nag_sym_lin_sys_ex01

 kappa(A) (1/rcond)
     7.57E+01

 Solution
      -4.0000
      -1.0000
       2.0000
       5.0000

 Backward error bound
     1.84E-16

 Forward error bound (estimate)
     4.66E-14
```

Example 2: Solution of a real symmetric positive definite system of linear equations

Solve a real symmetric positive definite system of linear equations with many right-hand sides $AX = B$, also estimating the condition number of A, and forward and backward error bounds on the computed solutions. This example calls the single procedure `nag_sym_lin_sol`, using packed storage for A.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_sym_lin_sys_ex02

  ! Example Program Text for nag_sym_lin_sys
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sym_lin_sys, ONLY : nag_key_pos, nag_sym_lin_sol
  USE nag_write_mat, ONLY : nag_write_gen_mat
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, j, n, nrhs
  REAL (wp) :: rcond
  CHARACTER (1) :: uplo
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: a(:), b(:,:), bwd_err(:), fwd_err(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_sym_lin_sys_ex02'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, nrhs
  READ (nag_std_in,*) uplo

  ALLOCATE (a((n*(n+1))/2),b(n,nrhs),bwd_err(nrhs), &
   fwd_err(nrhs))              ! Allocate storage

  SELECT CASE (uplo)
  CASE ('L','l')
    DO i = 1, n
      READ (nag_std_in,*) (a(i+((2*n-j)*(j-1))/2),j=1,i)
    END DO
  CASE ('U','u')
    DO i = 1, n
      READ (nag_std_in,*) (a(i+(j*(j-1))/2),j=i,n)
    END DO
  END SELECT

  READ (nag_std_in,*) (b(i,:),i=1,n)

  ! Solve the system of equations

  CALL nag_sym_lin_sol(nag_key_pos,uplo,a,b,bwd_err=bwd_err, &
   fwd_err=fwd_err,rcond=rcond)
```

```
      WRITE (nag_std_out,*)
      WRITE (nag_std_out,'(1X,''kappa(A) (1/rcond)''/2X,ES11.2)') 1/rcond
      WRITE (nag_std_out,*)

      CALL nag_write_gen_mat(b,int_col_labels=.TRUE., &
       title='Solutions (one solution per column)')

      WRITE (nag_std_out,*)
      WRITE (nag_std_out,*) 'Backward error bounds'
      WRITE (nag_std_out,'(2X,4ES11.2)') bwd_err
      WRITE (nag_std_out,*)
      WRITE (nag_std_out,*) 'Forward error bounds (estimates)'
      WRITE (nag_std_out,'(2X,4ES11.2)') fwd_err

      DEALLOCATE (a,b,bwd_err,fwd_err) ! Deallocate storage

    END PROGRAM nag_sym_lin_sys_ex02
```

2 Program Data

```
Example Program Data for nag_sym_lin_sys_ex02
  4 2                       : Values of n, nrhs
  'U'                       : Value of uplo
  4.16  -3.12   0.56  -0.10
        5.03  -0.83   1.18
              0.76   0.34
                     1.18 : End of Matrix A (upper triangle)
  8.70   8.30
-13.35   2.13
  1.89   1.61
 -4.14   5.00               : End of right-hand sides (one rhs per column)
```

3 Program Results

```
 Example Program Results for nag_sym_lin_sys_ex02

 kappa(A) (1/rcond)
     9.73E+01

 Solutions (one solution per column)
            1          2
       1.0000     4.0000
      -1.0000     3.0000
       2.0000     2.0000
      -3.0000     1.0000

 Backward error bounds
     7.63E-17   8.22E-17

 Forward error bounds (estimates)
     4.45E-14   4.53E-14
```

Example 3: Factorization of a complex symmetric matrix and use of the factorization to solve a system of linear equations

Solve a complex symmetric system of linear equations with many right-hand sides $AX = B$, also returning forward and backward error bounds on the computed solution. This example calls `nag_sym_lin_fac` to factorize A, and then `nag_sym_lin_sol_fac` to solve the equations using the factorization. The program uses conventional storage for A.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_sym_lin_sys_ex03

  ! Example Program Text for nag_sym_lin_sys
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sym_lin_sys, ONLY : nag_key_sym, nag_sym_lin_fac, &
   nag_sym_lin_sol_fac
  USE nag_write_mat, ONLY : nag_write_gen_mat, nag_write_tri_mat
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EPSILON, KIND, SCALE
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: det_exp, i, n, nrhs
  REAL (wp) :: rcond
  COMPLEX (wp) :: det_frac
  CHARACTER (1) :: uplo
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: pivot(:)
  REAL (wp), ALLOCATABLE :: bwd_err(:), fwd_err(:)
  COMPLEX (wp), ALLOCATABLE :: a(:,:), a_fac(:,:), b(:,:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_sym_lin_sys_ex03'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, nrhs
  READ (nag_std_in,*) uplo

  ALLOCATE (a(n,n),a_fac(n,n),b(n,nrhs),bwd_err(nrhs),fwd_err(nrhs), &
   pivot(n))                   ! Allocate storage

  a = 0.0_wp
  SELECT CASE (uplo)
  CASE ('L','l')
    READ (nag_std_in,*) (a(i,:i),i=1,n)
  CASE ('U','u')
    READ (nag_std_in,*) (a(i,i:),i=1,n)
  END SELECT
  a_fac = a

  READ (nag_std_in,*) (b(i,:),i=1,n)

  ! Carry out the Bunch-Kaufman factorization
```

```
  CALL nag_sym_lin_fac(nag_key_sym,uplo,a_fac,pivot,rcond=rcond, &
   det_frac=det_frac,det_exp=det_exp)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Results of the Bunch-Kaufman factorization'
  WRITE (nag_std_out,*)

  CALL nag_write_tri_mat(uplo,a_fac,format='(F7.4)', &
   title='Details of the Bunch-Kaufman factorization')

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Pivotal sequence (pivot)'
  WRITE (nag_std_out,'(2X,10I4:)') pivot

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,'(1X,''determinant = det_frac*SCALE(1.0_wp,det_exp) &
   &='',2X,"(",ES11.3,",",ES11.3,")")') det_frac*SCALE(1.0_wp,det_exp)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,'(1X,''kappa(A) (1/rcond)''/9X,ES11.2)') 1/rcond

  IF (rcond<=EPSILON(1.0_wp)) THEN
    WRITE (nag_std_out,*)
    WRITE (nag_std_out,*) ' ** WARNING ** '
    WRITE (nag_std_out,*) &
     'The matrix is almost singular: the solution may have no accuracy.'
    WRITE (nag_std_out,*) &
     'Examine the forward error bounds estimates returned in fwd_err.'
  END IF

  ! Solve the system of equations

  CALL nag_sym_lin_sol_fac(nag_key_sym,uplo,a_fac,pivot,b,a=a, &
   bwd_err=bwd_err,fwd_err=fwd_err)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) &
   'Results of the solution of the simultaneous equations'
  WRITE (nag_std_out,*)

  CALL nag_write_gen_mat(b,int_col_labels=.TRUE.,format='(F7.4)', &
   title='Solutions (one solution per column)')

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Backward error bounds'
  WRITE (nag_std_out,'(2X,4(7X,ES11.2:))') bwd_err
  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Forward error bounds (estimates)'
  WRITE (nag_std_out,'(2X,4(7X,ES11.2:))') fwd_err

  DEALLOCATE (a,a_fac,b,bwd_err,fwd_err,pivot) ! Deallocate storage

END PROGRAM nag_sym_lin_sys_ex03
```

2 Program Data

```
Example Program Data for nag_sym_lin_sys_ex03
  4 2                                                    : Values of n, nrhs
  'U'                                                    : Value of uplo
 (-0.39,-0.71) ( 5.14,-0.64) (-7.86,-2.96) ( 3.80, 0.92)
               ( 8.86, 1.81) (-3.52, 0.58) ( 5.32,-1.59)
                             (-2.83,-0.03) (-1.54,-2.86)
                                           (-0.56, 0.12) : End of Matrix A
 (-55.64, 41.22) (-19.09,-35.97)
 (-48.18, 66.00) (-12.08,-27.02)
 ( -0.49, -1.47) (  6.95, 20.49)
 ( -6.43, 19.24) ( -4.59,-35.53) : End of right-hand sides (one rhs per column)
```

3 Program Results

```
 Example Program Results for nag_sym_lin_sys_ex03

 Results of the Bunch-Kaufman factorization

 Details of the Bunch-Kaufman factorization
   (-2.0954,-2.2011) ( 0.6163, 0.3205) (-0.6361,-0.1468) ( 0.5427,-0.1831)
                     (-3.0624, 0.5785) (-6.0558,-3.9193) ( 0.5412,-0.2900)
                                       (-4.0456, 0.6792) (-0.3685, 0.1408)
                                                         ( 8.8600, 1.8100)

 Pivotal sequence (pivot)
     1  -1  -1   2

 determinant = det_frac*SCALE(1.0_wp,det_exp) =  ( -1.073E+03,  9.736E+02)

 kappa(A) (1/rcond)
            1.57E+01

 Results of the solution of the simultaneous equations

 Solutions (one solution per column)
                  1                 2
   ( 1.0000,-1.0000) (-2.0000,-1.0000)
   (-2.0000, 5.0000) ( 1.0000,-3.0000)
   ( 3.0000,-2.0000) ( 3.0000, 2.0000)
   (-4.0000, 3.0000) (-1.0000, 1.0000)

 Backward error bounds
            4.83E-17          1.27E-16

 Forward error bounds (estimates)
            2.27E-14          1.94E-14
```

Additional Examples

Not all example programs supplied with NAG *fl*90 appear in full in this module document. The following additional examples, associated with this module, are available.

`nag_sym_lin_sys_ex04`
: Solution of a real symmetric indefinite system of linear equations with one right-hand side, using packed storage.

`nag_sym_lin_sys_ex05`
: Solution of a real symmetric positive definite system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex06`
: Factorization of a complex symmetric matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex07`
: Solution of a real symmetric positive definite system of linear equations with one right-hand side, using conventional storage.

`nag_sym_lin_sys_ex08`
: Solution of a real symmetric positive definite system of linear equations with one right-hand side, using packed storage.

`nag_sym_lin_sys_ex09`
: Solution of a complex Hermitian positive definite system of linear equations with one right-hand side, using conventional storage.

`nag_sym_lin_sys_ex10`
: Solution of a complex Hermitian positive definite system of linear equations with one right-hand side, using packed storage.

`nag_sym_lin_sys_ex11`
: Solution of a complex Hermitian positive definite system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex12`
: Solution of a complex Hermitian positive definite system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex13`
: Factorization of a real symmetric positive definite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex14`
: Factorization of a real symmetric positive definite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex15`
: Factorization of a complex Hermitian positive definite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex16`
: Factorization of a complex Hermitian positive definite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex17`
: Solution of a complex Hermitian indefinite system of linear equations with one right-hand side, using conventional storage.

`nag_sym_lin_sys_ex18`
: Solution of a complex Hermitian indefinite system of linear equations with one right-hand side, using packed storage.

`nag_sym_lin_sys_ex19`

Solution of a real symmetric indefinite system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex20`

Solution of a real symmetric indefinite system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex21`

Solution of a complex Hermitian indefinite system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex22`

Solution of a complex Hermitian indefinite system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex23`

Factorization of a real symmetric indefinite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex24`

Factorization of a real symmetric indefinite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex25`

Factorization of a complex Hermitian indefinite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex26`

Factorization of a complex Hermitian indefinite matrix and use of the factorization to solve a system of linear equations with many right-hand sides, using packed storage.

`nag_sym_lin_sys_ex27`

Solution of a complex symmetric system of linear equations with one right-hand side, using conventional storage.

`nag_sym_lin_sys_ex28`

Solution of a complex symmetric system of linear equations with one right-hand side, using packed storage.

`nag_sym_lin_sys_ex29`

Solution of a complex symmetric system of linear equations with many right-hand sides, using conventional storage.

`nag_sym_lin_sys_ex30`

Solution of a complex symmetric system of linear equations with many right-hand sides, using packed storage.

References

[1] Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J J, Du Croz J J, Greenbaum A, Hammarling S, McKenney A, Ostrouchov S and Sorensen D (1995) *LAPACK Users' Guide* (2nd Edition) SIAM, Philadelphia

[2] Golub G H and Van Loan C F (1989) *Matrix Computations* Johns Hopkins University Press (2nd Edition)

[3] 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