# Module 5.3: nag_tri_lin_sys
# Triangular Systems of Linear Equations

`nag_tri_lin_sys` provides a procedure for solving real or complex triangular systems of linear equations with one or many right-hand sides:

$$Ax = b \text{ or } AX = B,$$

where $A$ is upper or lower triangular. It also provides a procedure for estimating the condition number of a triangular matrix and a procedure to evaluate the determinant of A in a form that avoids overflow or underflow.

## Contents

# Introduction

## 1 Notation

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 upper or lower triangular.

The procedure `nag_tri_lin_sol` solves the equations by a simple process of forward or backward substitution. It has options to solve alternative forms of the equations

$A^T x = b$, $A^T X = B$, $A^H x = b$ or $A^H X = B$.

(If $A$ is real, then $A^H = A^T$.)
The procedure `nag_tri_lin_sol` has options to return *forward* or *backward error bounds* on the computed solution.

The procedure `nag_tri_lin_cond` returns an estimate of the *condition number* of $A$, which is a measure of the sensitivity of the computed solution to perturbations of the original system or to rounding errors in the computation. For more details on error analysis, see the Chapter Introduction.

The procedure `nag_tri_mat_det` returns the *determinant* of $A$.

## 2 Storage of Matrices

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

### 2.1 Conventional Storage

`a` is a rank-2 array, of shape $(n,n)$. Matrix element $a_{ij}$ is stored in `a(`$i, j$`)`. If $A$ is upper triangular, only the elements of the upper triangle ($i \leq j$) need be stored; if $A$ is lower triangular, only the elements of the lower triangle ($i \geq j$) need be stored; 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.

### 2.2 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 | Triangular Matrix | Packed storage in array a |
|---|---|---|
| 'u' or 'U' | $\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$ | $a_{11}$ $\underbrace{a_{12}\,a_{22}}$ $\underbrace{a_{13}\,a_{23}\,a_{33}}$ $\underbrace{a_{14}\,a_{24}\,a_{34}\,a_{44}}$ |
| 'l' or 'L' | $\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$ | $\underbrace{a_{11}\,a_{21}\,a_{31}\,a_{41}}$ $\underbrace{a_{22}\,a_{32}\,a_{42}}$ $\underbrace{a_{33}\,a_{43}}$ $a_{44}$ |

## 2.3 Unit Triangular Matrices

A *unit* triangular matrix is a triangular matrix whose diagonal elements are known to be 1. The procedures in this module have an optional argument `unit_diag` which can be used to specify that the matrix is unit triangular, and then the diagonal elements do not need to be stored; the storage of the other elements of the matrix is not affected.

# Procedure: nag_tri_lin_sol

## 1 Description

nag_tri_lin_sol is a generic procedure which computes the solution of a real or complex triangular system of linear equations with one or many right-hand sides. It allows conventional or packed storage for $A$.

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 $B$);

where the matrix $A$ (the *coefficient matrix*) is upper or lower triangular.

Optionally, the procedure can solve alternative forms of the system of equations:

$A^T x = b$, $A^T X = B$, $A^H x = b$ or $A^H X = B$.

(If $A$ is real, then $A^H = A^T$.)

The procedure also has options to return *forward* and *backward error bounds* for the computed solution or solutions. See the Chapter Introduction for an explanation of these terms.

## 2 Usage

```
USE nag_tri_lin_sys
CALL nag_tri_lin_sol(uplo, a, b  [, optional arguments])
```

### 2.1 Interfaces

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

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 '**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

**uplo** — character(len=1), intent(in)

*Input:* specifies whether $A$ is upper or lower triangular.

If `uplo` = `'u'` or `'U'`, $A$ is upper triangular;

if `uplo` = `'l'` or `'L'`, $A$ is lower triangular.

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

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

*Input:* the triangular matrix $A$.

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

If `uplo` = `'u'`, $A$ is upper triangular, and elements below the diagonal need not be set;

if `uplo` = `'l'`, $A$ is lower triangular, and elements above the diagonal need not be set.

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

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

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

If `unit_diag` = `.true.`, the diagonal elements of $A$ are assumed to be 1; they need not be stored, and are not referenced by the procedure.

**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 have 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.

**unit_diag** — logical, intent(in), optional

*Input:* specifies whether $A$ has unit diagonal elements.

If `unit_diag` = `.false.`, the diagonal elements of $A$ must be explicitly stored;

if `unit_diag` = `.true.`, $A$ has unit diagonal elements: they need not be stored and are assumed to be 1.

*Default:* `unit_diag` = `.false.`.

**trans** — character(len=1), intent(in), optional

*Input:* specifies whether the equations involve $A$ or its transpose $A^T$ or its conjugate-transpose $A^H$ (= $A^T$ if $A$ is real).

If `trans` = `'n'` or `'N'`, the equations involve $A$ (i.e., $Ax = b$);

if `trans` = `'t'` or `'T'`, the equations involve $A^T$ (i.e., $A^T x = b$);

if `trans` = `'c'` or `'C'`, the equations involve $A^H$ (i.e., $A^H x = b$).

*Default:* `trans` = `'n'`.

*Constraints:* `trans` = `'n'`, `'N'`, `'t'`, `'T'`, `'c'` or `'C'`.

**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 $i$th solution vector, returned in the $i$th 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 $i$th solution vector, returned in the $i$th 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.

**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.<br>$A$ has a zero diagonal element, and so is exactly singular. No solutions or error bounds are computed. |

# 5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

# 6 Further Comments

## 6.1 Algorithmic Detail

The solution $x$ is computed by forward or backward substitution.

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 algorithm is 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 solutions of triangular systems of equations are usually computed to high accuracy. See Higham [3].

For each right-hand side $b$, the computed solution $\hat{x}$ is the exact solution of a perturbed system of equations $(A + E)\hat{x} = b$, such that

$$|E| \leq c(n)\epsilon|A|,$$

where $c(n)$ is a modest linear function of $n$, and $\epsilon =$ `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. If the system has one of the alternative forms $A^T x = b$ or $A^H x = b$, the appropriate condition number is $\kappa_1(A)$ $(= \kappa_\infty(A^T) = \kappa_\infty(A^H))$. An estimate of the reciprocal of $\kappa_\infty(A)$ or $\kappa_1(A)$ is returned by the function `nag_tri_lin_cond`. However, forward error bounds derived using these condition numbers 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 matrix 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 $n^2 r$ if $A$ is real, and $4n^2 r$ if $A$ is complex.

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

# Procedure: nag_tri_lin_cond

## 1 Description

nag_tri_lin_cond is a generic procedure which estimates the condition number of a real or complex triangular matrix $A$ of order $n$. It allows either conventional or packed storage for $A$.

The procedure can estimate the condition number in either the infinity-norm (the default), defined as

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty,$$

or in the 1-norm, defined as

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1.$$

Note that $\kappa_1(A) = \kappa_\infty(A^T) = \kappa_\infty(A^H)$.

If $A$ is singular, the condition number is infinite. Therefore, to avoid the possibility of overflow, the procedure returns the *reciprocal* of the condition number. If the reciprocal of the condition number is less than EPSILON(1.0_*wp*), then $A$ is singular to working precision.

## 2 Usage

```
USE nag_tri_lin_sys
[value =] nag_tri_lin_cond(uplo, a  [, optional arguments])
```

The function result is a scalar of type real(kind=*wp*).

### 2.1 Interfaces

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

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 '**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

**uplo** — character(len=1), intent(in)

*Input:* specifies whether $A$ is upper or lower triangular.

If uplo = 'u' or 'U', $A$ is upper triangular;

if uplo = 'l' or 'L', $A$ is lower triangular.

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

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

*Input:* the triangular matrix $A$.

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

If `uplo` = `'u'`, $A$ is upper triangular, and elements below the diagonal need not be set;

if `uplo` = `'l'`, $A$ is lower triangular, and elements above the diagonal need not be set.

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

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

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

If `unit_diag` = `.true.`, the diagonal elements of $A$ are assumed to be 1; and are not referenced by the procedure.

## 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.

**unit_diag** — logical, intent(in), optional

*Input:* specifies whether $A$ has unit diagonal elements.

If `unit_diag` = `.false.`, the diagonal elements of $A$ must be explicitly stored;

if `unit_diag` = `.true.`, $A$ has unit diagonal elements: they need not be stored and are assumed to be 1.

*Default:* `unit_diag` = `.false.`.

**one_norm** — logical, intent(in), optional

*Input:* specifies whether the condition number of $A$ is to be estimated in the infinity-norm or the 1-norm.

If `one_norm` = `.false.`, the procedure estimates $\kappa_\infty(A)$;

if `one_norm` = `.true.`, the procedure estimates $\kappa_1(A)$ $(= \kappa_\infty(A^T) = \kappa_\infty(A^H))$.

*Default:* `one_norm` = `.false.`.

**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. |
| **320** | The procedure was unable to allocate enough memory. |

# 5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

# 6 Further Comments

## 6.1 Algorithmic Detail

To estimate $\kappa_\infty(A)$ $(= \|A\|_\infty \|A^{-1}\|_\infty)$, the procedure first computes $\|A\|_\infty$ directly, and then uses Higham's modification of Hager's method (see Higham [2]) to estimate $\|A^{-1}\|_\infty$. The procedure returns the reciprocal $\rho = 1/\kappa_\infty(A)$, rather than $\kappa_\infty(A)$ itself.

A similar approach is used to estimate $\kappa_1(A)$.

The algorithm is derived from LAPACK (see Anderson *et al.* [1]).

## 6.2 Accuracy

The computed estimate is never less than the true value $\rho$, and in practice is nearly always less than $10\rho$ (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 method 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 $n^2$ floating-point operations if $A$ is real, or $4n^2$ 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_tri_mat_det

## 1 Description

nag_tri_mat_det is a generic procedure which evaluates the determinant of a real or complex triangular matrix $A$ of order $n$. It allows either conventional or packed storage for $A$.

The determinant is returned in a form which avoids overflow or underflow.

## 2 Usage

```
USE nag_tri_lin_sys
CALL nag_tri_mat_det(uplo, a, det_frac, det_exp  [, optional arguments])
```

### 2.1 Interfaces

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

Real / complex data

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

**Complex data:** a and det_frac are 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 '**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

**uplo** — character(len=1), intent(in)

*Input:* specifies whether $A$ is upper or lower triangular.

If uplo = 'u' or 'U', $A$ is upper triangular;

if uplo = 'l' or 'L', $A$ is lower triangular.

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

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

*Input:* the triangular matrix $A$.

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

If uplo = 'u', $A$ is upper triangular, and elements below the diagonal need not be set;

if uplo = 'l', $A$ is lower triangular, and elements above the diagonal need not be set.

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

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

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

If unit_diag = .true., the diagonal elements of $A$ are assumed to be 1; and are not referenced by the procedure.

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

**det_exp** — integer, intent(out)

*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 the same type as a.

## 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.

**unit_diag** — logical, intent(in), optional

*Input:* specifies whether $A$ has unit diagonal elements.

If unit_diag = .false., the diagonal elements of $A$ must be explicitly stored;

if unit_diag = .true., $A$ has unit diagonal elements: they need not be stored and are assumed to be 1.

*Default:* unit_diag = .false..

*Note:* this argument has been added for consistency with other procedures in the module.

**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. |
| **320** | The procedure was unable to allocate enough memory. |

# 5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

# Example 1: Solution of a real triangular system of linear equations

Solve a real triangular 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 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_tri_lin_sys_ex01

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_tri_lin_sys, ONLY : nag_tri_lin_sol, nag_tri_lin_cond, &
   nag_tri_mat_det
  USE nag_write_mat, ONLY : nag_write_gen_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) :: det_frac, rcond_inf
  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_tri_lin_sys_ex01'

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

  ALLOCATE (a(n,n),b(n,nrhs),bwd_err(nrhs), &
   fwd_err(nrhs))              ! 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(i,:),i=1,n)

  ! Compute the determinant

  CALL nag_tri_mat_det(uplo,a,det_frac,det_exp)

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

```
      ! Compute the condition number

      rcond_inf = nag_tri_lin_cond(uplo,a)

      WRITE (nag_std_out,*)
      WRITE (nag_std_out,'(1X,''kappa(A) (1/rcond_inf)''/2X,ES11.2)') 1/ &
       rcond_inf

      IF (rcond_inf<=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_tri_lin_sol(uplo,a,b,bwd_err=bwd_err,fwd_err=fwd_err)

      WRITE (nag_std_out,*)

      CALL nag_write_gen_mat(b,int_col_labels=.TRUE., &
       title='Solution vectors (one vector 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_tri_lin_sys_ex01
```

# 2 Program Data

```
Example Program Data for nag_tri_lin_sys_ex01
  4 2                          : Values of n, nrhs
  'L'                          : Value of uplo
  4.30
 -3.96  -4.87
  0.40   0.31  -8.02
 -0.27   0.07  -5.95   0.12   : End of Matrix A (lower triangle)
-12.90 -21.50
 16.75  14.93
-17.55   6.33
-11.04   8.09                 : End of right-hand sides (one rhs per column)
```

# 3 Program Results

```
Example Program Results for nag_tri_lin_sys_ex01

determinant = SCALE(det_frac,det_exp) =    2.015E+01

kappa(A) (1/rcond_inf)
    1.38E+02

Solution vectors (one vector per column)
           1          2
     -3.0000    -5.0000
     -1.0000     1.0000
      2.0000    -1.0000
      1.0000     6.0000

Backward error bounds
    6.89E-17   0.00E+00

Forward error bounds (estimates)
    1.66E-13   5.27E-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_tri_lin_sys_ex02`
: Solution of a complex triangular system of linear equations with many right-hand sides, using conventional storage.

`nag_tri_lin_sys_ex03`
: Solution of a real triangular system of linear equations with many right-hand sides, using packed storage.

`nag_tri_lin_sys_ex04`
: Solution of a complex triangular system of linear equations with many right-hand sides, using packed storage.

`nag_tri_lin_sys_ex05`
: Solution of a real triangular system of linear equations with one right-hand side, using conventional storage.

`nag_tri_lin_sys_ex06`
: Solution of a complex triangular system of linear equations with one right-hand side, using conventional storage.

`nag_tri_lin_sys_ex07`
: Solution of a real triangular system of linear equations with one right-hand side, using packed storage.

`nag_tri_lin_sys_ex08`
: Solution of a complex triangular system of linear equations with one right-hand side, 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] 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

[3] Higham N J (1989) The accuracy of solutions to triangular systems *SIAM J. Numer. Anal.* **26** 1252–1265