Module 6.4: nag_lin_lsq

Linear Least-squares Problems

`nag_lin_lsq` provides procedures for solving linear least-squares problems, using either the singular value decomposition (SVD) or the QR factorization, or a combination of the two.

It also provides procedures for performing the QR factorization and related computational tasks.

Contents

Introduction

1 Notation and Background

The *linear least-squares problem* is to find x so as to

$$\text{minimize} \quad \|b - Ax\|_2 \tag{1}$$

where A is an $m \times n$ matrix, b is an m-element right-hand side vector, and x is an n-element solution vector.

Usually, $m \geq n$ and $\text{rank}(A) = n$ (that is, A has *full rank*), and in this case the solution x is unique; the problem is also referred to as finding the least-squares solution to an *over-determined* system of linear equations.

If $m < n$ and $\text{rank}(A) = m$ (again, A has full rank), there are an infinite number of solutions x which exactly satisfy $b - Ax = 0$. We can restore uniqueness by imposing the additional requirement of minimizing $\|x\|_2$. This problem is also referred to as finding the minimum norm solution to an *under-determined* system of linear equations.

In general, if $\text{rank}(A) < \min(m,n)$ (that is, A is *rank-deficient*), there is a unique *minimum norm* solution which minimizes both $\|b - Ax\|_2$ and $\|x\|_2$.

The minimum norm solution is not always the preferred solution to a rank-deficient problem or when $m < n$. An alternative solution, which is sometimes preferable, is a solution with at most $\text{rank}(A)$ non-zero components; this is known as a *basic* solution. A basic solution is not necessarily unique.

The theoretical remarks in this section assume that the rank of A is well defined. The next section discusses how to determine the rank in practical numerical work.

2 The SVD and the Numerical Rank of a Matrix

The most robust method for solving linear least-squares problems, allowing for the possibility that they may be rank-deficient, is the *Singular Value Decomposition* (SVD).

The SVD of an $m \times n$ real or complex matrix A is given by

$$A = U\Sigma V^H \quad \text{(with } V^H = V^T \text{ if } A \text{ is real).}$$

Here

Σ is an $m \times n$ diagonal matrix, whose $\min(m,n)$ diagonal elements are the *singular values* σ_i of A; they are real and non-negative, and arranged in descending order:

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)} \geq 0.$$

U is a real orthogonal or complex unitary matrix of order m; its leading $\min(m,n)$ columns are the *left singular vectors* of A.

V is a real orthogonal or complex unitary matrix of order n; its leading $\min(m,n)$ columns are the *right singular vectors of A*.

The largest singular value $\sigma_1(A)$ is the value of the 2-norm of A:

$$\|A\|_2 = \sigma_1(A)$$

and the ratio of the largest to the smallest singular value gives the condition number of A in the 2-norm:

$$\kappa_2(A) = \frac{\sigma_1(A)}{\sigma_{\min(m,n)}(A)}.$$

For more details about the SVD, see the module `nag_svd` (6.3).

In exact arithmetic, a matrix A has rank r if it has precisely r non-zero singular values; A is rank-deficient if $r < \min(m, n)$, or, equivalently, if $\kappa_2(A) = \infty$.

In practical work, because of uncertainties in the data and rounding errors due to computation in finite precision arithmetic, the *numerical rank* is defined to be the number of singular values which are greater than a specified threshold.

In this module, the threshold is defined as `tol` $\times \sigma_1$ ($=$ `tol` $\times \|A\|_2$), where `tol` is a tolerance supplied by the user; `tol` should normally be set to the relative accuracy of the data. For example, if the elements of the matrix A are correct to 4 significant figures, then a suitable value for `tol` is 5×10^{-4}. It follows that A is effectively rank-deficient if $\kappa_2(A) \geq 1/$`tol`.

Note that the solution to the least-squares problem depends on the determination of rank, and hence on the user-supplied tolerance `tol`; it may be desirable to experiment with different values of `tol`.

3 Solution Using the SVD

Given the SVD of $A = U\Sigma V^H$,

$$\|b - Ax\|_2 = \|c - \Sigma V^H x\|_2, \quad \text{where} \quad c = U^H b.$$

If the numerical rank of A is r, let

c_1 consist of the first r elements of c,

c_2 consist of the remaining elements of c,

Σ_1 be the leading $r \times r$ sub-matrix of Σ,

V_1 consist of the leading r columns of V.

Then the minimum norm solution to (1) is given by

$$x = V_1 \Sigma_1^{-1} c_1$$

and the residual sum of squares $\|b - Ax\|_2^2 = \|c_2\|_2^2$.

4 Solution Using QR Factorization

A cheaper route to solve problem (1) is via the QR factorization, with or without column pivoting.

In this section we assume $m \geq n$ (the more frequent case) for simplicity, although the procedures also handle problems with $m < n$.

4.1 QR Factorization Without Column Pivoting

The QR factorization (without column pivoting) of the $m \times n$ real or complex matrix A is:

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix},$$

where

Q is an $m \times m$ real orthogonal or complex unitary matrix,

R is an $n \times n$ upper triangular matrix.

If A has *full rank* n, the QR factorization can be used to solve the least-squares problem (1), since:

$$\|b - Ax\|_2 = \|c - Q^H Ax\|_2 = \left\| \begin{pmatrix} c_1 - Rx \\ c_2 \end{pmatrix} \right\|_2,$$

where

$c = Q^H b$,

c_1 consists of the first n elements of c,

c_2 consists of the remaining elements of c.

Then the unique solution x is given by:

$$x = R^{-1}c_1,$$

and the residual sum of squares $\|b - Ax\|_2^2 = \|c_2\|_2^2$.

To check that the problem is indeed of full rank, we can check the condition number of R, since $\kappa_2(R) = \kappa_2(A)$.

4.2 QR Factorization with Column Pivoting

Rank-deficient linear least-squares problems can often be solved using the QR factorization *with column pivoting*. This involves interchanging columns of A, so that the factorization takes the form:

$$AP = Q \begin{pmatrix} R \\ 0 \end{pmatrix},$$

where P is a *permutation* matrix. The column interchanges are chosen so that

$$|r_{11}| \geq |r_{22}| \geq |r_{33}| \ldots,$$

and, moreover, for each k

$$|r_{kk}| \geq \|R_{k:j,j}\|_2 \text{ for } j = k+1, \ldots, \min(m,n).$$

In exact arithmetic, if $\text{rank}(A) = r$, then

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

where R_{11} is the leading $r \times r$ sub-matrix, and $R_{22} = 0$. In numerical computation, we aim to determine an index r such that R_{11} is well conditioned, and R_{22} is negligible. Note that this *is not always possible*, even though the matrix is numerically rank-deficient.

If such a partition of R is possible, then a solution to the linear least-squares problem (1) is given by:

$$x = P \begin{pmatrix} R_{11}^{-1}\hat{c}_{11} \\ 0 \end{pmatrix}$$

where $\hat{c}_1$ consists of the first r elements of $c = Q^H b$. This is not a minimum-norm solution, but a *basic* solution with at most r non-zero components.

5 Solution Using QR Factorization and SVD

To obtain the most reliable numerical determination of rank, and to compute a minimum-norm solution to a rank-deficient problem, it is necessary to use the SVD, as was described in Section 3. However, if a QR factorization of A has already been computed, this can be combined with the SVD of the $n \times n$ upper triangular matrix R, to give the SVD of A. (If $m \gg n$, this can in fact be a more efficient method for computing the SVD of A.)

Therefore the following approach is often effective:

1. Compute the QR factorization of A, with or without column pivoting.
2. If A is numerically of full rank, then compute the unique solution using the QR factorization, as described in Section 4.1.
3. If the numerical rank of A can be clearly determined from the matrix R, and if a basic solution is acceptable, then compute a basic solution, as described in Section 4.2.
4. Otherwise, compute the SVD of R, and use the resulting SVD of A to determine the numerical rank and compute a solution as described in Section 3.

6 Choice of Procedures

The procedure `nag_lin_lsq_sol` is the simplest procedure to use. It solves a linear least-squares problem in a single call, by computing the SVD of A. By default, it computes the minimum norm solution, but it has an option to return a basic solution.

It may be followed by calls to `nag_lin_lsq_sol_svd` in order to try the effect of varying the tolerance used to determine the numerical rank, or to compare a minimum norm solution with a basic solution. `nag_lin_lsq_sol_svd` may also be used

- to solve a problem with the same matrix A but a different right-hand side b, without recomputing the SVD of A;
- to solve a linear least-squares problem after the SVD of A has been computed by the procedure `nag_gen_svd` in the module `nag_svd` (6.3) (or calls to lower-level procedures in that module).

The remaining procedures enable a solution to be obtained using the *QR factorization*. Two or more procedures must be called in succession, and care must be taken in the numerical determination of rank.

`nag_qr_fac` computes a QR factorization, optionally with column pivoting, and `nag_qr_orth` performs related computational tasks (but is not strictly necessary for solving linear least-squares problems); `nag_qr_fac` has an optional argument `rcond` which can be used to check for rank-deficiency.

`nag_lin_lsq_sol_qr` solves a linear least-squares problem, assuming that a QR factorization has already been performed by `nag_qr_fac`, and that you have determined the numerical rank of the problem.

`nag_lin_lsq_sol_qr_svd` solves a linear least-squares problem using the SVD, assuming that a QR factorization of A has already been performed by `nag_qr_fac`.

Thus `nag_qr_fac` and `nag_lin_lsq_sol_qr_svd` together can perform the same tasks as the single procedure `nag_lin_lsq_sol` (and may be more efficient if $m \gg n$). They also may be followed by calls to `nag_lin_lsq_sol_svd` to try the effect of varying the tolerance, and so on.

All the relevant procedures can handle many right-hand side vectors b_i and their corresponding solution vectors x_i in a single call, storing them as columns of matrices B and X respectively. Note however that the linear least-squares problem is solved for each right-hand side vector *independently*; this is *not* the same as finding a matrix X which minimizes $\|B - AX\|_2$.

Procedure: nag_lin_lsq_sol

1 Description

nag_lin_lsq_sol is a generic procedure which solves a real or complex linear least-squares problem.

Notation: the problem is to find x so as to

$$\text{minimize} \quad \|b - Ax\|_2$$

where A is an $m \times n$ matrix, b is an m-element right-hand side vector, and x is an n-element solution vector.

In the most usual case, $m \geq n$ and $\text{rank}(A) = n$ (that is, A has *full rank*); the solution is then unique.

For other types of problems (when $m < n$ or A is *rank-deficient*), the solution is not unique. By default the (unique) *minimum norm* solution is returned; however, the procedure has options to return a *basic* solution. See the Module Introduction for more details.

The procedure uses a method based on computing the SVD of A; see the Module Introduction for definition of the SVD. It has options to return the relevant parts of the SVD, so that they can subsequently be passed to the lower-level procedure nag_lin_lsq_sol_svd to solve additional problems without recomputing the SVD.

2 Usage

```
USE nag_lin_lsq
CALL nag_lin_lsq_sol(a, b, x  [, 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, b, x and the optional argument u are of type real(kind=*wp*).

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

One / many right-hand sides

One r.h.s.: b and x are rank-1 arrays, and the optional argument std_err is a scalar.

Many r.h.s.: b and x are rank-2 arrays, and the optional argument std_err 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.

m — the number of equations (the number of rows of A)

n — the number of unknowns (the number of columns of A)

k — the number of right-hand sides

n_U $(\min(m,n) \leq n_U \leq m)$ — the number of columns to be computed of the matrix U

3.1 Mandatory Arguments

a(m, n) — real(kind=wp) / complex(kind=wp), intent(inout)

Input: the matrix A.

Output: the leading $\min(m, n)$ rows of A are overwritten by the corresponding rows of V^H (the right singular vectors of A, stored row-wise — and conjugated if complex).

b(m) / **b**(m, k) — real(kind=wp) / complex(kind=wp), intent(inout)

Input: if `b` has rank 1, it holds the single right-hand side vector b. If `b` has rank 2, each of its columns holds a right-hand side vector.

Output: each right-hand side vector b is overwritten by $U^H b$.

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

x(n) / **x**(n, k) — real(kind=wp) / complex(kind=wp), intent(out)

Output: if `x` has rank 1, it holds the single solution vector x. If `x` has rank 2, then the ith column holds the solution vector corresponding to the right-hand side vector in the ith column of `b`.

Constraints: `x` must be of the same type and rank as `b`.

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.

solution — character(len=1), intent(in), optional

Input: specifies the type of solution required.

If `solution` = `'m'` or `'M'`, the minimum norm solution;

if `solution` = `'b'` or `'B'`, a basic solution.

Default: `solution` = `'m'`.

Constraints: `solution` = `'m'`, `'M'`, `'b'` or `'B'`.

tol — real(kind=wp), intent(in), optional

Input: the relative tolerance used to determine the rank of A. `tol` should be chosen as approximately the largest relative error in the elements of A. A singular value is considered negligible if it is less than or equal to `tol`$\times\sigma_1$ (= `tol`$\times\|A\|_2$).

Default: `tol` = `EPSILON(1.0_`wp`)`.

Constraints: $0.0 \leq$ `tol` ≤ 1.0.

rank — integer, intent(out), optional

Output: the effective rank r of the matrix A; it is the number of singular values which are *not* considered negligible (see `tol`).

std_err / **std_err**(k) — real(kind=wp), intent(out), optional

Output: if `std_err` is a scalar, it returns the standard error of the single solution vector x, defined as $\|Ax - b\|_2/\sqrt{m - r}$ if $m > r$, and zero if $m = r$, where r is the effective rank of A. If `std_err` is an array, then `std_err`(i) returns the standard error of the solution vector in the ith column of `x`.

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

sigma$(\min(m, n))$ — real(kind=wp), intent(out), optional

Output: the singular values of A, in descending order.

u(m, n_U) — real(kind=*wp*) / complex(kind=*wp*), intent(out), optional

Output: the first n_U columns of the matrix U. The most likely values of n_U are: $\min(m,n)$, giving the first $\min(m,n)$ columns of U (the left singular vectors); or m, giving the whole of U. U is needed if you wish to solve additional problems with the same matrix A but different right-hand sides, without recomputing the SVD of A.

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

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.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Failure to converge.
	(This error is not likely to occur.) The QR algorithm failed to compute all the singular values in the permitted number of iterations.

5 Examples of Usage

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

See also Section 5 for procedure document `nag_lin_lsq_sol_svd`.

6 Further Comments

6.1 Algorithmic Detail

The procedure first calls `nag_gen_svd` to compute the SVD of A; more precisely, `nag_gen_svd` returns the singular values, overwrites A with the first $\min(m,n)$ rows of V^H, and overwrites each right-hand side vector b with $U^H b$. The algorithm is derived from LAPACK (see Anderson *et al.* [1]).

This procedure then calls `nag_lin_lsq_sol_svd` to solve the linear least-squares problem. This procedure first determines the rank r of A, using the value of `tol` as described in the Module Introduction. It then computes either the minimum norm solution or a basic solution, as described in the document for `nag_lin_lsq_sol_svd`.

6.2 Accuracy

For a discussion of the sensitivity of the solution to uncertainties in the data, see Golub and Van Loan [2], Sections 5.3 (for full rank problems) and 5.5 (for rank-deficient problems).

6.3 Timing

The time taken is roughly proportional to mn^2 if $m \geq n$, or to $m^2 n$ if $m \leq n$.

Procedure: nag_lin_lsq_sol_svd

1 Description

nag_lin_lsq_sol_svd is a generic procedure which solves a real or complex linear least-squares problem, assuming that the relevant parts of the singular value decomposition (SVD) of the coefficient matrix have already been computed, usually by nag_lin_lsq_sol.

This procedure can also be used following a call to nag_lin_lsq_sol_qr_svd, or following a call to nag_gen_svd in the module nag_svd (6.3) (or calls to lower-level procedures in that module).

Notation: the problem is to find x so as to

$$\text{minimize} \quad \|b - Ax\|_2$$

where A is an $m \times n$ matrix, b is an m-element right-hand side vector, and x is an n-element solution vector.

Let the SVD of A be

$A = U\Sigma V^T$, with U and V orthogonal, if A is real;

$A = U\Sigma V^H$, with U and V unitary, if A is complex.

The problem is therefore equivalent to minimizing

$$\|c - \Sigma V^H x\|_2$$

where $c = U^H b$. This is the form in which the least-squares problem must be presented to this procedure, following a call to nag_gen_svd to compute Σ, V^H and $c = U^H b$.

In the most usual case, $m \geq n$ and $\text{rank}(A) = n$ (that is, A has *full rank*), the solution is unique.

If the problem is *rank-deficient* or $m < n$, the solution is not unique. By default the (unique) *minimum norm* solution is returned; however, the procedure has options to return a *basic* solution. See the Module Introduction for more details.

The procedure may be called repeatedly with different values of solution or tol, but with the other input arguments unchanged, in order to see the difference between the two types of solution, or the effect of changing tol.

2 Usage

```
USE nag_lin_lsq
CALL nag_lin_lsq_sol_svd(vh, sigma, c, x  [, optional arguments])
```

2.1 Interfaces

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

Real / complex data

Real data: vh, c and x are of type real(kind=*wp*).

Complex data: vh, c and x are of type complex(kind=*wp*).

One / many right-hand sides

One r.h.s.: c and x are rank-1 arrays, and the optional argument std_err is a scalar.

Many r.h.s.: c and x are rank-2 arrays, and the optional argument std_err 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 '$\mathtt{x}(n)$' 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.

m — the number of equations

n — the number of unknowns

k — the number of right-hand sides

3.1 Mandatory Arguments

vh($\min(m,n),n$) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the leading $\min(m,n)$ rows of the matrix V^H from the singular value decomposition of A — in other words, the right singular vectors of A, stored row-wise, as returned by nag_lin_lsq_sol, nag_lin_lsq_sol_qr_svd or nag_gen_svd.

sigma($\min(m,n)$) — real(kind=*wp*), intent(in)

Input: the singular values of A in descending order, as returned by nag_lin_lsq_sol, nag_lin_lsq_sol_qr_svd or nag_gen_svd.

c(m) / **c**(m,k) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: if c has rank 1, it must hold the vector $c = U^H b$, as returned by nag_lin_lsq_sol or nag_lin_lsq_sol_qr_svd in its argument b, or by nag_gen_svd in its optional argument c_vec; here b is the single right-hand side vector of the original problem. If c has rank 2, it must hold the matrix $C = U^H B$, as returned by nag_lin_lsq_sol or nag_lin_lsq_sol_qr_svd in its argument b, or by nag_gen_svd in its optional argument c_mat; here each column of B is a right-hand side vector of the original problem.

Constraints: c must be of the same type as vh.

x(n) / **x**(n,k) — real(kind=*wp*) / complex(kind=*wp*), intent(out)

Output: if x has rank 1, it holds the single solution vector x. If x has rank 2, then the ith column holds the solution vector corresponding to the right-hand side vector in the ith column of c.

Constraints: x must be of the same type and rank as c.

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.

solution — character(len=1), intent(in), optional

Input: specifies the type of solution required.

If solution = 'm' or 'M', the minimum norm solution;

if solution = 'b' or 'B', a basic solution.

Default: solution = 'm'.

Constraints: solution = 'm', 'M', 'b' or 'B'.

tol — real(kind=*wp*), intent(in), optional

Input: the relative tolerance used to determine the rank of A. tol should be chosen as approximately the largest relative error in the elements of A. A singular value is considered negligible if it is less than or equal to $\mathtt{tol}\times\sigma_1$ ($= \mathtt{tol}\times\|A\|_2$).

Default: tol = EPSILON(1.0_*wp*).

Constraints: $0.0 \leq \mathtt{tol} \leq 1.0$.

rank — integer, intent(out), optional

Output: the effective rank r of the matrix A; it is the number of singular values which are *not* considered negligible (see `tol`).

std_err / std_err(k) — real(kind=*wp*), intent(out), optional

Output: if `std_err` is a scalar, it returns the standard error of the single solution vector x, defined as $\|Ax-b\|_2/\sqrt{m-r}$ if $m>r$, and zero if $m=r$, where r is the effective rank of A. If `std_err` is an array, then `std_err(`i`)` returns the standard error of the solution vector in the ith column of `x`.

Constraints: if `c` has rank 1, `std_err` must be a scalar; if `c` has rank 2, `std_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.
303	Array arguments have inconsistent shapes.
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.

That example illustrates calls to this procedure following a call to `nag_lin_lsq_sol`: `nag_lin_lsq_sol` computes the minimum norm solution with one value of `tol`, and then alternative solutions to the same problem are computed using this procedure, reusing results returned by `nag_lin_lsq_sol` in the first $\min(m,n)$ rows of `a`, in `sigma` and in `b`. The calls are:

```
tol = 0.005_wp
CALL nag_lin_lsq_sol(a,b,x_ls,sigma=sigma,tol=tol,rank=rank, &
                     std_err=std_err_ls)

tol = 0.0005_wp
CALL nag_lin_lsq_sol_svd(a(:min(m,n),:),sigma,b,x_ls,tol=tol,rank=rank, &
                         std_err=std_err_ls)

tol = 0.005_wp
CALL nag_lin_lsq_sol_svd(a(:min(m,n),:),sigma,b,x_b,solution='Basic', &
                         tol=tol,rank=rank,std_err=std_err_b)
```

The effect of the call to `nag_lin_lsq_sol` could be achieved by a call to the procedure `nag_gen_svd` followed by a call to this procedure as follows:

```
CALL nag_gen_svd(a,sigma,overwrite='v',c_vec=b)

CALL nag_lin_lsq_sol_svd(a(:min(m,n),:),sigma,b,x_ls,tol=tol,rank=rank, &
                         std_err=std_err_ls)
```

To enable a second problem with a different right-hand side to be solved, without recomputing the SVD of A, `nag_lin_lsq_sol` must return the whole of the matrix U in an array `u`, of shape `(m,m)`; this must then be applied to the new right-hand side b to compute $c=U^Hb$. This procedure may be then be used as follows:

```
CALL nag_lin_lsq_sol(a,b,x_ls,sigma=sigma,u=u,tol=tol,rank=rank, &
                     std_err=std_err_ls)

READ (*,*) b            ! read new right-hand side
b = Matmul(Conjg(u),b)

CALL nag_lin_lsq_sol_svd(a(:min(m,n),:),sigma,b,x_ls,tol=tol,rank=rank, &
                         std_err=std_err_ls)
```

6 Further Comments

6.1 Algorithmic Detail

The procedure first determines the numerical rank r, using the value of `tol`: r is the number of singular values σ_i which are greater than `tol` $\times\ \sigma_1$.

Let:

$\hat{V}$ consist of the first r columns of V;

$\hat{\Sigma}$ consist of the leading $r \times r$ sub-matrix of Σ;

$\hat{c}_1$ consist of the first r elements of c.

If the minimum norm solution has been requested, it is computed as

$$x = \hat{V}\hat{\Sigma}^{-1}\hat{c}_1.$$

If a basic solution has been requested and $r < n$, the procedure performs a QR factorization with column pivoting of the $r \times n$ matrix $\hat{\Sigma}\hat{V}^H$:

$$\hat{\Sigma}\hat{V}^H = Q \begin{pmatrix} R_1 & R_2 \end{pmatrix} P^T$$

where Q is an orthogonal (or unitary) matrix of order r, R_1 is upper triangular, and P is a permutation matrix. A basic solution is then computed as:

$$x = P \begin{pmatrix} R_1^{-1}Q^H\hat{c}_1 \\ 0 \end{pmatrix}.$$

6.2 Accuracy

For a discussion of the sensitivity of the solution to uncertainties in the data, see Golub and Van Loan [2], Sections 5.3 (for full rank problems) and 5.5 (for rank-deficient problems).

6.3 Timing

Computing a minimum norm solution requires $O(nr)$ floating-point operations; computing a basic solution is more expensive, and requires $O(nr^2)$ operations.

Procedure: nag_qr_fac

1 Description

nag_qr_fac is a generic procedure which computes the QR factorization of a real or complex $m \times n$ general matrix A. The factorization takes the following forms.

If $m \geq n$:

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = Q_1 R,$$

where Q is an $m \times m$ real orthogonal or complex unitary matrix, Q_1 consists of the first n columns of Q, and R is an $n \times n$ upper triangular matrix.

If $m < n$:

$$A = QR = Q \, (R_1 \; R_2),$$

where R is upper trapezoidal, R_1 is $n \times n$ upper triangular, and R_2 is rectangular.

By default, the orthogonal or unitary matrix Q is represented as the product of $\min(m, n)$ elementary reflectors; this representation can be passed to the procedure nag_qr_orth to perform further operations with Q.

Note that for any $k < \min(m, n)$, the information returned in the first k columns of the array a represents a QR factorization of the first k columns of A.

If the optional argument q is present, Q (or its leading columns if $m > n$) is formed explicitly and returned in q. If $m \geq n$, the first n columns of Q form an orthonormal basis for the space spanned by the columns of A.

If the optional argument pivot is present, then the procedure computes the QR factorization of A *with column pivoting* — that is, the QR factorization of A with its columns interchanged, or in other words the QR factorization of AP, where P is a *permutation* matrix. The column interchanges are chosen so that

$$|r_{11}| \geq |r_{22}| \geq |r_{33}| \ldots,$$

and, moreover, for each k

$$|r_{kk}| \geq \|R_{k:j,j}\|_2 \text{ for } j = k+1, \ldots, \min(m, n).$$

The procedure also allows specified columns of A to be moved to the leading columns of AP at the start of the factorization and fixed there. The remaining columns are free to be interchanged so that at the ith stage the pivot column is chosen to be the column which maximizes the 2-norm of elements i to m over columns i to n.

The procedure can optionally return an estimate of the reciprocal of the condition number of R in the 1-norm, $\kappa_1(R)$ (see the optional argument rcond). If $m \geq n$, the condition number of R in the 2-norm is equal to that of A: $\kappa_2(R) = \kappa_2(A)$; $\kappa_1(R)$ differs from $\kappa_2(R)$ by a factor of at most n, and hence can be used to test whether A is near to being numerically rank-deficient.

2 Usage

```
USE nag_lin_lsq

CALL nag_qr_fac(a, tau  [, optional arguments])
```

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 '$\mathtt{x}(n)$' 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.

m — the number of rows of A

n — the number of columns of A

n_Q — the number of leading columns to be computed of the orthogonal (unitary) matrix Q

n_Q must satisfy the constraint: $\min(m, n) \le n_Q \le m$; hence, if $m < n$, $n_Q = m$.

3.1 Mandatory Arguments

a(m, n) — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: the m by n matrix A.

Output: details of the factorization.

If $m \ge n$, the elements below the diagonal are overwritten by details of the matrix Q and the upper triangle is overwritten by the corresponding elements of the n by n upper triangular matrix R;

if $m < n$, the strictly lower triangular part is overwritten by details of the matrix Q and the remaining elements are overwritten by the corresponding elements of the m by n upper trapezoidal matrix R.

tau($\min(m, n)$) — real(kind=*wp*) / complex(kind=*wp*), intent(out)

Output: further details of the transformation matrix Q; a and tau together may be required for passing to nag_qr_orth, nag_lin_lsq_sol_qr or nag_lin_lsq_sol_qr_svd.

Constraints: tau 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.

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

Input: if $\mathtt{pivot}(i) \neq 0$, then the ith column of A is moved to the beginning of AP before the factorization is computed and is fixed in place during the computation. Otherwise, the ith column of A is a free column (i.e., one which may be interchanged during the computation with any other free column).

Output: details of the permutation matrix P. More precisely, if $\mathtt{pivot}(i) = k$, then the kth column of A is moved to become the ith column of AP; in other words, the columns of AP are the columns of A in the order $\mathtt{pivot}(1), \mathtt{pivot}(2), \ldots, \mathtt{pivot}(n)$.

Default: if pivot is absent, no columns are interchanged, i.e., the QR factorization is computed without column pivoting.

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

Output: an estimate of the *reciprocal* of the condition number of R if $m \ge n$, or of R_1 if $m < n$ (this is less likely to be useful). If $m \ge n$ and rcond is less than the relative accuracy of the data, then R is approximately singular to that working accuracy, and therefore A is numerically rank-deficient. The reciprocal is returned rather than the condition number itself to avoid the risk of overflow.

q(m, n_Q) — real(kind=wp) / complex(kind=wp), intent(out), optional

Output: the leading n_Q columns of the orthogonal or unitary matrix Q, where $\min(m,n) \le n_Q \le m$.

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

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
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 2 and 3 of this module document.

The first of these examples illustrates a call to this procedure, followed by a call to `nag_qr_orth` to form the leading columns of Q. The calls are:

```
CALL nag_qr_fac(a,tau,pivot=pivot,rcond=rcond)

CALL nag_qr_orth(a,tau,q=q)
```

The same effect could have been achieved by a single call to this procedure:

```
CALL nag_qr_fac(a,tau,pivot=pivot,q=q,rcond=rcond)
```

6 Further Comments

6.1 Algorithmic Detail

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

6.2 Accuracy

The computed factorization is the exact factorization of a nearby matrix $A+E$, where $\|E\|_2 = O(\epsilon)\,\|A\|_2$.

If the matrix Q is computed, this differs from an exactly orthogonal (unitary) matrix by a matrix F such that $\|F\|_2 = O(\epsilon)$.

The estimate of the reciprocal of the condition number returned in `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). Strictly speaking, the algorithm estimates the condition number in the 1-norm, but this differs from the condition number in the 2-norm by a factor of at most n.

6.3 Timing

For real data, the total number of floating-point operations performed is roughly as follows:

	$m \geq n$	$m < n$
QR factorization:	$(2/3)n^2(3m-n)$	$(2/3)m^2(3n-m)$
Form leading n columns of Q:	$(2/3)n^2(3m-n)$	
Form leading m columns of Q:	$4mn(m-n)+(4/3)n^3$	$(4/3)m^3$
Estimate `rcond`:	$2cn^2$	$2cm^2$

where $4 \leq c \leq 11$.

For complex data, 4 times as many (real) floating-point operations are performed.

Procedure: nag_qr_orth

1 Description

nag_qr_orth is intended for use following a call to nag_qr_fac which performs the QR factorization of a general real or complex $m \times n$ matrix A. nag_qr_fac represents the orthogonal or unitary matrix Q as the product of $\min(m, n)$ elementary reflectors:

$$Q = H_1 H_2 \ldots H_{\min(m,n)}$$

where

$$H_i = I - \tau_i v_i v_i^H;$$

the vector v_i is stored in a$(i + 1 : m, i)$ and the scalar τ_i is stored in tau(i).

This procedure accepts this representation and may be used to carry out either or both of the following tasks:

- form Q explicitly as a square matrix or form only its leading columns; Q can be returned either in the optional argument q or overwritten on a.
- apply Q to a given real (complex) matrix C from the left or right, overwriting C with QC, CQ, $Q^H C$ or CQ^H ($Q^H = Q^T$ for real data) or to a real (complex) vector c from the left only, overwriting c with Qc or $Q^H c$.

2 Usage

```
USE nag_lin_lsq
CALL nag_qr_orth(a, tau  [, optional arguments])
```

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.

m — the number of rows of the factorized matrix A
n — the number of columns of the factorized matrix A
n_Q — the number of leading columns to be computed of the orthogonal (unitary) matrix Q
m_C — the number of rows of C or of elements of c: $m_C = m$ if Q is applied from the left
n_C — the number of columns of C: $n_C = m$ if Q is applied from the right

n_Q must satisfy the constraint: $\min(m, n) \leq n_Q \leq m$; hence, if $m < n$, $n_Q = m$.

3.1 Mandatory Arguments

a(m, n) — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: details of the representation of Q as returned by nag_qr_fac.

Output: if q_on_a is present and set to .true., the leading $\min(m, n)$ columns of a are overwritten by the leading $\min(m, n)$ columns of Q; otherwise (by default), a is unchanged.

tau(min(m, n)) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: further details of the representation of Q as returned by `nag_qr_fac`.

Constraints: `tau` 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.

q_on_a — logical, intent(in), optional

Input: specifies whether the leading columns of the matrix Q are to be overwritten on `a`.

If `q_on_a` = `.true.`, the leading min(m, n) columns of Q are overwritten on `a`;

if `q_on_a` = `.false.`, the leading n_Q columns of Q are returned in `q` if present, or else are not formed explicitly.

Default: `q_on_a` = `.false.`.

q(m, n_Q) — real(kind=*wp*) / complex(kind=*wp*), intent(out), optional

Output: the leading n_Q columns of the orthogonal or unitary matrix Q, where $\min(m, n) \leq n_Q \leq m$.

Note: if `q_on_a` is present and set to `.true.`, and `q` is also present, then `q` is not used and a warning is raised.

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

side — character(len=1), intent(in), optional

Input: specifies whether Q (or Q^T or Q^H) is to be applied to C from the left or from the right (*always from the left* for c).

If `side` = `'l'` or `'L'`, from the left;

if `side` = `'r'` or `'R'`, from the right.

Default: `side` = `'l'`.

Constraints: if `c_mat` is present, then `side` = `'l'`, `'L'`, `'r'` or `'R'`; if `c_vec` is present, then `side` = `'l'` or `'L'`. `side` must not be present unless `c_mat` or `c_vec` is present.

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

Input: specifies whether Q, Q^T or Q^H is to be applied to C or c.

If `trans` = `'n'` or `'N'`, Q is applied;

if `trans` = `'t'` or `'T'` (real matrices only), Q^T is applied;

if `trans` = `'c'` or `'C'` (complex matrices only), Q^H is applied.

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

Constraints:

for the real case `trans` = `'n'`, `'N'`, `'t'` or `'T'`;

for the complex case `trans` = `'n'`, `'N'`, `'c'` or `'C'`;

`trans` must not be present unless `c_mat` or `c_vec` is present.

c_mat(m_C, n_C) — real(kind=*wp*) / complex(kind=*wp*), intent(inout), optional

Input: the matrix C.

Output: overwritten by QC, Q^TC, Q^HC, CQ, CQ^T or CQ^H, according to the values of `side` and `trans`.

Constraints:

`c_mat` must be of the same type as `a`;

`c_mat` and `c_vec` must not both be present.

c_vec(m) — real(kind=wp) / complex(kind=wp), intent(inout), optional

Input: the vector c.

Output: overwritten by Qc, Q^Tc or Q^Hc according to the value of `trans`.

Constraints:

`c_vec` must be of the same type as `a`;

`c_mat` and `c_vec` must not both be present.

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.
303	Array arguments have inconsistent shapes.
304	Invalid presence of an optional argument.
320	The procedure was unable to allocate enough memory.

Warnings (error%level = 1):

error%code	Description
101	Optional argument present but not used. `q` is present when `q_on_a` is `.true.`; the matrix Q is returned in `a`, and `q` is not used.
102	No computation performed. `q_on_a` is not present or is set to `.false.`, and none of `q`, `c_mat` or `c_vec` is present; no computation has been requested.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

In this example `nag_qr_orth` is called to form the leading columns of Q in an array `q`. The call is:

```
CALL nag_qr_orth(a,tau,q=q)
```

To overwrite the leading columns of Q on the array `a`, the call would be:

```
CALL nag_qr_orth(a,tau,q_on_a=.true.)
```

6 Further Comments

6.1 Algorithmic Detail

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

6.2 Accuracy

If the matrix Q is formed, it differs from an exactly orthogonal (unitary) matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$,

If the matrix C is to be transformed, the computed result differ from the exact result by a matrix F such that $\|F\|_2 = O(\epsilon)\|C\|_2$.

6.3 Timing

For real data, the total number of floating-point operations performed is roughly as follows:

	$m \geq n$	$m < n$
Form leading n columns of Q:	$(2/3)n^2(3m-n)$	
Form leading m columns of Q:	$4mn(m-n)+(4/3)n^3$	$(4/3)m^3$
Compute QC or Q^TC:	$2n_Cn(2m-n)$	$2n_Cm^2$
Compute Qc or Q^Tc:	$2n(2m-n)$	$2m^2$
Compute CQ or CQ^T:	$2m_Cn(2m-n)$	$2m_Cm^2$

For complex data, 4 times as many (real) floating-point operations are performed.

Procedure: nag_lin_lsq_sol_qr

1 Description

nag_lin_lsq_sol_qr is a generic procedure which solves a real or complex linear least-squares problem, assuming that a QR factorization of the coefficient matrix has already been computed by nag_qr_fac.

Notation: the problem is to find x so as to

$$\text{minimze} \quad \|b - Ax\|_2$$

where A is an $m \times n$ matrix, b is an m-element right-hand side vector, and x is an n-element solution vector. The procedure can handle either a single right-hand side or several right-hand sides (stored as the columns of the array b).

First, assume $m \geq n$ (the most usual case).

If A has rank n (that is, A has *full rank*), there is a unique solution. If you are sure that A has full rank and is nowhere near to being rank-deficient (you can use the optional argument rcond in nag_qr_fac to test for this), then this procedure can reliably find the solution, using a QR factorization of A without pivoting.

However, if you are not confident that A has full rank, you should use the column pivoting option when calling nag_qr_fac (that is, supply the optional argument pivot). See the Module Introduction for advice on using the QR factorization with column pivoting for the numerical determination of rank. If A is deemed to be of full rank n, then this procedure can find the unique solution (the optional argument pivot *must* be supplied). If A is deemed to have well determined rank $r < n$, then a solution can be found by setting the optional argument rank to r; this solution is a basic solution (with r non-zero components), not a minimum norm solution. To find a minimum norm solution or to make the most reliable numerical determination of rank, use the procedure nag_lin_lsq_sol_qr_svd.

If $m < n$, the problem is equivalent to finding a solution to an underdetermined system of linear equations $Ax = b$. There are an infinite number of solutions. This procedure can compute a basic solution, with m non-zero components if A is known to have full rank m, or with $r < m$ non-zero components if A has well determined rank r; the column-pivoting option *must* be used in the QR factorization of A, and the rank r supplied through the optional argument rank, as when $m \geq n$.

2 Usage

```
USE nag_lin_lsq
CALL nag_lin_lsq_sol_qr(a, tau, b, x  [, 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, tau, b and x are of type real(kind=wp).

Complex data: a, tau, b and x are of type complex(kind=wp).

One / many right-hand sides

One r.h.s.: b and x are rank-1 arrays, and the optional argument std_err is a scalar.

Many r.h.s.: b and x are rank-2 arrays, and the optional argument std_err 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.

m — the number of equations

n — the number of unknowns

k — the number of right-hand sides

3.1 Mandatory Arguments

a(m, n) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: details of the QR factorization as returned by nag_qr_fac.

tau$(\min(m, n))$ — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: further details of the orthogonal matrix Q as returned by nag_qr_fac.

Constraints: tau must be of the same type as a.

b(m) / **b**(m, k) — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: if b has rank 1, it holds the single right-hand side vector b. If b has rank 2, each of its columns holds a right-hand side vector.

Output: each right-hand side vector b is overwritten by $Q^H b$.

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

x(n) / **x**(n, k) — real(kind=*wp*) / complex(kind=*wp*), intent(out)

Output: if x has rank 1, it holds the single solution vector x. If x has rank 2, then the ith column holds the solution vector corresponding to the right-hand side vector in the ith column of b.

Constraints: x must be of the same type and rank as b.

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.

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

Input: details of the permutation matrix P as returned by nag_qr_fac if column pivoting was used. pivot *must* be present in the call to this procedure, if it was present in the call to nag_qr_fac.

Default: it is assumed that column pivoting was not used.

Constraints: $1 \leq \texttt{pivot}(i) \leq n$, for $i = 1, 2, \ldots, n$; pivot must be present if $m < n$.

rank — integer, intent(in), optional

Input: the rank r of the matrix.

Default: rank $= \min(m, n)$.

Constraints: $0 \leq \texttt{rank} \leq \min(m, n)$; rank must not be present unless pivot is present.

std_err / **std_err**(k) — real(kind=*wp*), intent(out), optional

Output: if std_err is a scalar, it returns the standard error of the single solution vector x, defined as $\|Ax - b\|_2/\sqrt{m - r}$ if $m > r$, and zero if $m = r$, where r is the rank of A if supplied in rank, or $\min(m, n)$ otherwise. If std_err is an array, then std_err(i) returns the standard error of the solution vector in the ith column of x.

Constraints: if b has rank 1, std_err must be a scalar; if b has rank 2, std_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.
303	Array arguments have inconsistent shapes.
305	Invalid absence of an optional argument.
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 2 of this module document.

6 Further Comments

6.1 Algorithmic Detail

A description of the method employed can be found in the Module Introduction.

6.2 Accuracy

For a discussion of the sensitivity of the solution to uncertainties in the data, see Golub and Van Loan [2], Sections 5.3 (for full rank problems) and 5.5 (for rank-deficient problems).

6.3 Timing

Computing a solution requires roughly $2\min(m,n)\,(2m-\min(m,n))+r^2$ floating-point operations for real problems, and 4 times as many floating-point operations for complex problems.

Procedure: nag_lin_lsq_sol_qr_svd

1 Description

`nag_lin_lsq_sol_qr_svd` is a generic procedure which solves a real or complex linear least-squares problem, assuming that a QR factorization of the coefficient matrix has already been computed by `nag_qr_fac`.

The solution is obtained by computing the SVD (Singular Value Decomposition) of the $n \times n$ upper triangular matrix R, and combining this with the QR factorization to give the SVD of A.

`nag_qr_fac` and this procedure together provide the same facilities as the single procedure `nag_lin_lsq_sol`.

This procedure has options to return the relevant parts of the SVD so that they can subsequently be passed to `nag_lin_lsq_sol_svd` to solve additional problems with the same A but different b without recomputing the SVD of A. `nag_lin_lsq_sol_qr_svd` may also be followed by calls to `nag_lin_lsq_sol_svd` to try the effect of varying the value of `tol`, or to compare a minimum norm solution with a basic solution.

2 Usage

```
USE nag_lin_lsq
CALL nag_lin_lsq_sol_qr_svd(a, tau, b, x  [, 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`, `tau`, `b` and `x` and the optional argument `u` are of type real(kind=*wp*).

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

One / many right-hand sides

One r.h.s.: `b` and `x` are rank-1 arrays, and the optional argument `std_err` is a scalar.

Many r.h.s.: `b` and `x` are rank-2 arrays, and the optional argument `std_err` 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.

m — the number of equations

n — the number of unknowns

k — the number of right-hand sides

3.1 Mandatory Arguments

$\mathbf{a}(m, n)$ — real(kind=*wp*) / complex(kind=*wp*), intent(inout)

Input: details of the QR factorization of A as returned by `nag_qr_fac`.

Output: the leading $\min(m, n)$ rows of the matrix of the right singular vectors V^H.

tau(min(m,n)) — real(kind=wp) / complex(kind=wp), intent(in)

Input: further details of the orthogonal matrix Q as returned by nag_qr_fac.

Constraints: tau must be of the same type as a.

b(m) / **b**(m,k) — real(kind=wp) / complex(kind=wp), intent(inout)

Input: if b has rank 1, it holds the single right-hand side vector b. If b has rank 2, each of its columns holds a right-hand side vector.

Output: each right-hand side vector b is overwritten by $U^H b$.

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

x(n) / **x**(n,k) — real(kind=wp) / complex(kind=wp), intent(out)

Output: if x has rank 1, it holds the single solution vector x. If x has rank 2, then the ith column holds the solution vector corresponding to the right-hand side vector in the ith column of b.

Constraints: x must be of the same type and rank as b.

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.

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

Input: details of the permutation matrix P as returned by nag_qr_fac if column pivoting was used.

Default: it is assumed that column pivoting was not used.

Constraints: $1 \leq$ pivot(i) $\leq n$, for $i = 1, 2, \ldots, n$.

solution — character(len=1), intent(in), optional

Input: specifies the type of solution required.

If solution = 'm' or 'M', the minimum norm solution;

if solution = 'b' or 'B', a basic solution.

Default: solution = 'm'.

Constraints: solution = 'm', 'M', 'b' or 'B'.

tol — real(kind=wp), intent(in), optional

Input: the relative tolerance used to determine the rank of A. tol should be chosen as approximately the largest relative error in the elements of A. A singular value is considered negligible if it is less than or equal to tol$\times\sigma_1$ (= tol$\times$ $\|A\|_2$).

Default: tol = EPSILON(1.0_wp).

Constraints: $0.0 \leq$ tol ≤ 1.0.

rank — integer, intent(out), optional

Output: the effective rank r of the matrix A; it is the number of singular values which are *not* considered negligible (see tol).

std_err / **std_err**(k) — real(kind=wp), intent(out), optional

Output: if std_err is a scalar, it returns the standard error of the single solution vector x, defined as $\|Ax - b\|_2/\sqrt{m-r}$ if $m > r$, and zero if $m = r$, where r is the effective rank of A. If std_err is an array, then std_err(i) returns the standard error of the solution vector in the ith column of x.

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

sigma(min(m, n)) — real(kind=wp), intent(out), optional

Output: the singular values of A, in descending order.

u(m, n_U) — real(kind=wp) / complex(kind=wp), intent(out), optional

Output: the first n_U columns of the matrix U. The most likely values of n_U are: min(m, n), giving the first min(m, n) columns of U (the left singular vectors); or m, giving the whole of U. U is needed if you wish to solve additional problems with the same matrix A but different right-hand sides, without recomputing the SVD of A.

Constraints: `u` must be of the same type as `a` and $\min(m, n) \leq n_U \leq m$.

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.
303	Array arguments have inconsistent shapes.
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 3 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The procedure uses `nag_gen_svd` to compute the SVD of R, and `nag_qr_orth` to help compute the SVD of A, overwriting `a` with the first min(m, n) columns of V^H and overwriting `b` with $U^H b$.

This procedure then calls `nag_lin_lsq_sol_svd` to solve the linear least-squares problem. `nag_lin_lsq_sol_svd` first determines the rank r of A, using the value of `tol` as described in the Module Introduction. It then computes either the minimum norm solution or a basic solution, as described in the document for `nag_lin_lsq_sol_svd`.

6.2 Accuracy

For a discussion of the sensitivity of the solution to uncertainties in the data, see Golub and Van Loan [2] , Sections 5.3 (for full rank problems) and 5.5 (for rank-deficient problems).

6.3 Timing

The number of floating point operations required to compute the SVD of R is proportional to n^3.

Given the relevant parts of the SVD of A, computing a minimum norm solution requires $O(nr)$ floating-point operations; computing a basic solution is more expensive, and requires $O(nr^2)$ operations.

Example 1: Solution of a Real Linear Least-squares Problem Using the SVD

This program calls the procedure `nag_lin_lsq_sol` to compute a solution to a linear least-squares problem using the singular value decomposition.

Assuming that the data is only accurate to within ±0.5%, `tol` is set to 0.005; to this tolerance, the matrix A is numerically rank-deficient, and the minimum norm solution is returned by default.

The singular values of A are printed out. They show that with `tol` reduced by a factor of 10, A would be regarded as being of full rank. The program makes a subsequent call to `nag_lin_lsq_sol_svd` to compute the solution returned with `tol` = 0.0005.

Finally, the program calls `nag_lin_lsq_sol_svd` again to compute a basic solution with the original value `tol` = 0.005.

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_lin_lsq_ex01

  ! Example Program Text for nag_lin_lsq
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_lin_lsq, ONLY : nag_lin_lsq_sol, nag_lin_lsq_sol_svd
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND, MIN
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, m, n, ns, rank
  REAL (wp) :: std_err_b, std_err_ls, tol
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: a(:,:), b(:), sigma(:), x_b(:), x_ls(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_lin_lsq_ex01'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) m, n
  ns = MIN(m,n)

  ALLOCATE (a(m,n),b(m),sigma(ns),x_b(n),x_ls(n)) ! Allocate storage

  READ (nag_std_in,*) (a(i,:),i=1,m)
  READ (nag_std_in,*) b

  ! Compute the minimum norm solution with tol = 0.005

  tol = 0.005_wp

  CALL nag_lin_lsq_sol(a,b,x_ls,sigma=sigma,tol=tol,rank=rank, &
   std_err=std_err_ls)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,'(1X,A,F7.4)') 'Minimum norm solution with tol =', &
   tol
```

```
      WRITE (nag_std_out,'(2X,F7.4)') x_ls
      WRITE (nag_std_out,'(1X,A,I7,A,F7.4)') 'rank =', rank, &
       '  standard error =', std_err_ls
      WRITE (nag_std_out,*)
      WRITE (nag_std_out,*) 'Singular values of A'
      WRITE (nag_std_out,'(2X,F7.4)') sigma

      ! Compute the minimum norm solution with tol = 0.0005

      tol = 0.0005_wp

      CALL nag_lin_lsq_sol_svd(a(:ns,:),sigma,b,x_ls,tol=tol,rank=rank, &
       std_err=std_err_ls)

      WRITE (nag_std_out,*)
      WRITE (nag_std_out,'(1X,A,F7.4)') 'Minimum norm solution with tol =', &
       tol
      WRITE (nag_std_out,'(2X,F7.4)') x_ls
      WRITE (nag_std_out,'(1X,A,I7,A,F7.4)') 'rank =', rank, &
       '  standard error =', std_err_ls

      ! Compute a basic solution with tol = 0.005

      tol = 0.005_wp

      CALL nag_lin_lsq_sol_svd(a(:ns,:),sigma,b,x_b,solution='Basic',tol=tol, &
       rank=rank,std_err=std_err_b)

      WRITE (nag_std_out,*)
      WRITE (nag_std_out,'(1X,A,F7.4)') 'Basic solution with tol =', tol
      WRITE (nag_std_out,'(2X,F7.4)') x_b
      WRITE (nag_std_out,'(1X,A,I7,A,F7.4)') 'rank =', rank, &
       '  standard error =', std_err_b

      DEALLOCATE (a,b,sigma,x_b,x_ls) ! Deallocate storage

    END PROGRAM nag_lin_lsq_ex01
```

2 Program Data

```
Example Program Data for nag_lin_lsq_ex01
  6  5                             : Values of m, n
 -0.09   0.14  -0.46   0.68   1.29
 -1.56   0.20   0.29   1.09   0.51
 -1.48  -0.43   0.89  -0.71  -0.96
 -1.09   0.84   0.77   2.11  -1.27
  0.08   0.55  -1.13   0.14   1.74
 -1.59  -0.72   1.06   1.24   0.34 : End of Matrix A
 -0.01
  0.04
  0.05
 -0.03
  0.02
 -0.06                             : End of right-hand side vector b
```

3 Program Results

```
Example Program Results for nag_lin_lsq_ex01

Minimum norm solution with tol = 0.0050
 -0.0440
  0.0440
 -0.0293
 -0.0439
 -0.0062
rank =      4  standard error = 0.0225

Singular values of A
  3.9997
  2.9962
  2.0001
  0.9988
  0.0025

Minimum norm solution with tol = 0.0005
 -0.1841
 -0.3719
 -0.6189
  0.1097
 -0.2632
rank =      5  standard error = 0.0318

Basic solution with tol = 0.0050
 -0.0370
  0.0647
  0.0000
 -0.0515
  0.0066
rank =      4  standard error = 0.0225
```

Example 2: Solution of a Real Linear Least-squares Problem Using the QR Factorization

This example uses the same data as Example 1.

The program calls `nag_qr_fac` to perform a QR factorization of A with column pivoting, and prints the matrix R. It calls `nag_qr_orth` to compute the leading columns of Q for illustration, and prints them, although they are not needed for solving the linear least-squares problem.

The program then calls `nag_lin_lsq_sol_qr` to compute a solution, assuming that the problem is of full rank.

However, the estimate of the condition number shows that if the data are only known to an accuracy of $\pm 0.5\%$ (1 part in 200), the problem should be regarded as rank-deficient. Inspection of the matrix R shows a clear separation between the leading 4×4 sub-matrix and the (5,5) element. The program calls `nag_lin_lsq_sol_qr` a second time with `rank` = 4; note that a copy of the original right-hand side b must be kept for this second call.

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_lin_lsq_ex02

  ! Example Program Text for nag_lin_lsq
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_lin_lsq, ONLY : nag_qr_fac, nag_qr_orth, nag_lin_lsq_sol_qr
  USE nag_write_mat, ONLY : nag_write_gen_mat, nag_write_tri_mat
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND, MIN
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, m, n, ns
  REAL (wp) :: rcond, std_err
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: pivot(:)
  REAL (wp), ALLOCATABLE :: a(:,:), b(:), bb(:), q(:,:), tau(:), x(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_lin_lsq_ex02'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) m, n
  ns = MIN(m,n)

  ALLOCATE (pivot(n),a(m,n),b(m),bb(m),q(m,ns),tau(ns), &
   x(n))                            ! Allocate storage

  READ (nag_std_in,*) (a(i,:),i=1,m)
  READ (nag_std_in,*) b
  bb = b

  ! Compute the QR factorization

  pivot = 0
```

```
    CALL nag_qr_fac(a,tau,pivot=pivot,rcond=rcond)

    WRITE (nag_std_out,*)

    CALL nag_write_tri_mat('upper',a(:n,:),format='(1X,F7.4)', &
     title='Matrix R')

    WRITE (nag_std_out,'(1X,A,ES11.2)') 'Estimated condition number =', &
     1/rcond

    ! Compute the leading min(m,n) columns of Q

    CALL nag_qr_orth(a,tau,q=q)

    WRITE (nag_std_out,*)

    CALL nag_write_gen_mat(q,format='(1X,F7.4)',title='Leading columns of Q' &
     )

    ! Compute the solution, assuming that A has full rank

    CALL nag_lin_lsq_sol_qr(a,tau,b,x,pivot=pivot,std_err=std_err)

    WRITE (nag_std_out,*)
    WRITE (nag_std_out,*) 'Solution assuming that A has full rank'
    WRITE (nag_std_out,'(3X,F7.4)') x
    WRITE (nag_std_out,'(1X,A,F7.4)') 'standard error =', std_err

    ! Compute the solution, assuming that A has rank 4

    b = bb

    CALL nag_lin_lsq_sol_qr(a,tau,b,x,pivot=pivot,rank=4,std_err=std_err)

    WRITE (nag_std_out,*)
    WRITE (nag_std_out,*) 'Solution assuming that A has rank 4'
    WRITE (nag_std_out,'(3X,F7.4)') x
    WRITE (nag_std_out,'(1X,A,F7.4)') 'standard error =', std_err

    DEALLOCATE (pivot,a,b,bb,q,tau,x) ! Deallocate storage

  END PROGRAM nag_lin_lsq_ex02
```

2 Program Data

```
Example Program Data for nag_lin_lsq_ex02
  6  5                            : Values of m, n
 -0.09   0.14  -0.46   0.68   1.29
 -1.56   0.20   0.29   1.09   0.51
 -1.48  -0.43   0.89  -0.71  -0.96
 -1.09   0.84   0.77   2.11  -1.27
  0.08   0.55  -1.13   0.14   1.74
 -1.59  -0.72   1.06   1.24   0.34 : End of Matrix A
 -0.01
  0.04
  0.05
 -0.03
  0.02
 -0.06                            : End of right-hand side vector b
```

3 Program Results

```
Example Program Results for nag_lin_lsq_ex02

Matrix R
   2.8904  0.5162 -1.7198  0.2024 -1.5026
          -2.7084 -0.3648 -0.0873  1.1475
                   2.2523  0.8397 -0.0060
                          -1.0086  0.7116
                                  -0.0034
Estimated condition number =   2.03E+03

Leading columns of Q
  -0.0311 -0.4822  0.2000  0.0632 -0.8302
  -0.5397 -0.2912  0.0247 -0.2609  0.3231
  -0.5120  0.2569 -0.6646 -0.2520 -0.3540
  -0.3771  0.3970  0.7132 -0.3491 -0.1226
   0.0277 -0.6372 -0.0199 -0.5012  0.2059
  -0.5501 -0.2304  0.0932  0.7010  0.1540

Solution assuming that A has full rank
  -0.1841
  -0.3719
  -0.6189
   0.1097
  -0.2632
standard error = 0.0318

Solution assuming that A has rank 4
  -0.0370
   0.0647
   0.0000
  -0.0515
   0.0066
standard error = 0.0225
```

Example 3: Solution of a Real Linear Least-squares Problem Using the QR Factorization Followed by the SVD

This example again uses the same data as Example 1.

The program calls `nag_qr_fac` to perform a QR factorization of A with column pivoting, and prints the matrix R, and the estimate of its condition number.

If the problem is considered to be of full rank (for the given value of `tol`), the program calls `nag_lin_lsq_sol_qr` to solve the problem, otherwise it calls `nag_lin_lsq_sol_qr_svd` to use the SVD for a reliable determination of the numerical rank of the problem and to compute a minimum norm solution.

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_lin_lsq_ex03

  ! Example Program Text for nag_lin_lsq
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_lin_lsq, ONLY : nag_qr_fac, nag_lin_lsq_sol_qr, &
   nag_lin_lsq_sol_qr_svd
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND, MIN
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, m, n, ns, rank
  REAL (wp) :: rcond, std_err, tol
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: pivot(:)
  REAL (wp), ALLOCATABLE :: a(:,:), b(:), tau(:), x(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_lin_lsq_ex03'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) m, n
  ns = MIN(m,n)

  ALLOCATE (pivot(n),a(m,n),b(m),tau(ns),x(n)) ! Allocate storage

  READ (nag_std_in,*) (a(i,:),i=1,m)
  READ (nag_std_in,*) b

  ! Compute the QR factorization

  pivot = 0

  CALL nag_qr_fac(a,tau,pivot=pivot,rcond=rcond)

  tol = 0.005_wp
  WRITE (nag_std_out,*)
  WRITE (nag_std_out,'(1X,2(A,ES11.2))') 'tol =', tol, '  rcond =', rcond
  WRITE (nag_std_out,*)

  IF (rcond>tol) THEN
```

```
    ! Compute the solution, assuming that A has full rank,
    ! using the QR factorization

    CALL nag_lin_lsq_sol_qr(a,tau,b,x,pivot=pivot,std_err=std_err)

    WRITE (nag_std_out,*) &
     'Solution from nag_lin_lsq_sol_qr, assuming A has full rank'

  ELSE

    ! Compute the minimum norm solution using the SVD

    CALL nag_lin_lsq_sol_qr_svd(a,tau,b,x,pivot=pivot,tol=tol,rank=rank, &
     std_err=std_err)

    WRITE (nag_std_out,'(1X,A,I7)') &
     'Solution from nag_lin_lsq_sol_qr_svd, assuming A has rank', rank

  END IF

  WRITE (nag_std_out,'(3X,F7.4)') x
  WRITE (nag_std_out,'(1X,A,F7.4)') 'standard error =', std_err

  DEALLOCATE (pivot,a,b,tau,x) ! Deallocate storage

END PROGRAM nag_lin_lsq_ex03
```

2 Program Data

```
Example Program Data for nag_lin_lsq_ex03
  6  5                            : Values of m, n
 -0.09   0.14  -0.46   0.68   1.29
 -1.56   0.20   0.29   1.09   0.51
 -1.48  -0.43   0.89  -0.71  -0.96
 -1.09   0.84   0.77   2.11  -1.27
  0.08   0.55  -1.13   0.14   1.74
 -1.59  -0.72   1.06   1.24   0.34 : End of Matrix A
 -0.01
  0.04
  0.05
 -0.03
  0.02
 -0.06                            : End of right-hand side vector b
```

3 Program Results

```
 Example Program Results for nag_lin_lsq_ex03

 tol =   5.00E-03  rcond =   4.92E-04

 Solution from nag_lin_lsq_sol_qr_svd, assuming A has rank      4
   -0.0440
    0.0440
   -0.0293
   -0.0439
   -0.0062
 standard error = 0.0225
```

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_lin_lsq_ex04`

Solution of a complex linear least-squares problem with one right-hand side using the SVD.

`nag_lin_lsq_ex05`

Solution of a real linear least-squares problem with many right-hand sides using the SVD.

`nag_lin_lsq_ex06`

Solution of a complex linear least-squares problem with many right-hand sides using the SVD.

`nag_lin_lsq_ex07`

Solution of a complex linear least-squares problem with one right-hand side using the QR factorization.

`nag_lin_lsq_ex08`

Solution of a real linear least-squares problem with many right-hand sides using the QR factorization.

`nag_lin_lsq_ex09`

Solution of a complex linear least-squares problem with many right-hand sides using the QR factorization.

`nag_lin_lsq_ex10`

Solution of a complex linear least-squares problem with one right-hand side using the QR factorization followed by the SVD (rank-deficient).

`nag_lin_lsq_ex11`

Basic solution of a real linear least-squares problem with many right-hand sides using the QR factorization followed by the SVD.

`nag_lin_lsq_ex12`

Basic solution of a complex linear least-squares problem with many right-hand sides using the QR factorization followed by the SVD.

References

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

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