Module 9.2: nag_nlin_lsq
Unconstrained Nonlinear Least-squares

nag_nlin_lsq contains a procedure for solving nonlinear least-squares problems and another for estimating the associated variance–covariance matrix for such problems.

Contents

Introduction

This module contains three procedures and a derived type as follows.

- `nag_nlin_lsq_sol` computes an unconstrained minimum of a sum of squares of m nonlinear functions in n variables (where $m \geq n$). You must provide a procedure that defines the nonlinear functions. For maximum reliability, you should also provide all their first partial derivatives. If you do not wish to provide any derivatives, an option is provided whereby this procedure will approximate them using finite differences.
- `nag_nlin_lsq_cov` estimates elements of the variance–covariance matrix C at the solution returned by `nag_nlin_lsq_sol`. This procedure can be used to find either the diagonal elements of C, or the elements of the jth column of C, or the whole of C.
- `nag_nlin_lsq_cntrl_init` assigns default values to all the structure components of the derived type `nag_nlin_lsq_cntrl_`*wp*.
- `nag_nlin_lsq_cntrl_`*wp* may be used to supply optional parameters to `nag_nlin_lsq_sol`.

Procedure: nag_nlin_lsq_sol

1 Description

nag_nlin_lsq_sol computes an unconstrained minimum of a sum of squares of m nonlinear functions in n variables.

The procedure is applicable to problems of the form:

$$\text{Minimize } F(x) = \sum_{i=1}^{m} (f_i(x))^2$$

where $x = (x_1, x_2, \ldots, x_n)^T$ and $m \geq n$. (The functions $f_i(x)$ are often referred to as 'residuals'.) You must supply a procedure to calculate the values of the $f_i(x)$ and, optionally, their first derivatives $\partial f_i / \partial x_j$ at any point x.

This procedure is intended for objective functions which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

A description of the algorithm is given in the Mathematical Background section of this module document.

By default, the user-supplied procedure lsq_fun is expected to return the appropriate values of the first partial derivatives of the $f_i(x)$; if these are not available then the optional argument deriv (see Section 3.2) must be supplied as .false. (this procedure will then approximate them using finite differences).

If the function F represents the goodness of fit of a nonlinear model to observed data and you wish to compute elements of the variance–covariance matrix of the estimated regression coefficients (i.e., by calling nag_nlin_lsq_cov after calling this procedure), then the optional arguments s and v (see Section 3.2) must be supplied in the call to this procedure.

2 Usage

```
USE nag_nlin_lsq
CALL nag_nlin_lsq_sol(lsq_fun, x, f_sum_sq, f_vec  [, 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 '**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.

$n \geq 1$ — the number of variables

$m \geq n$ — the number of nonlinear functions or residuals

3.1 Mandatory Arguments

lsq_fun — subroutine

The procedure lsq_fun, supplied by the user, must calculate the vector of values $f_i(x)$ and, optionally, their first derivatives $\partial f_i / \partial x_j$ at any point x. (However, if you do not wish to calculate the residuals at a particular x, there is the option of setting a parameter to cause this procedure to terminate immediately.)

Its specification is:

```
subroutine lsq_fun(x, finish, f_vec, f_jac)

real(kind=wp), intent(in) ::  x(:)
```

Shape: `x` has shape (n).

Input: the point x at which the values of f_i and (optionally) $\partial f_i/\partial x_j$ are required, for $i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$.

```
logical, intent(inout) ::  finish
```

Input: `finish` will always be `.false.` on entry.

Output: if you wish to terminate the call to this procedure then `finish` should be set to `.true.`. If `finish` is `.true.` on exit from `lsq_fun`, then this procedure will terminate with `error%code` = 202.

```
real(kind=wp), intent(out) ::  f_vec(:)
```

Shape: `f_vec` has shape (m).

Output: unless `finish` is reset to `.true.`, `f_vec(`i`)` must contain the value of f_i at the point x, for $i = 1, 2, \ldots, m$.

```
real(kind=wp), intent(out), optional ::  f_jac(:,:)
```

Shape: `f_jac` has shape (m, n).

Output: if present, `f_jac(`i, j`)` must contain the value of the first derivative $\partial f_i/\partial x_j$ at the point x, for $i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$.

Note: if no derivatives are supplied (i.e., the optional argument `deriv` (see Section 3.2) is `.false.`), then `f_jac` will not be supplied in any call to `lsq_fun`; however, the definition of `lsq_fun` must still contain the argument `f_jac` in its specification. (By default, `lsq_fun` will always be called with `f_jac` present.)

Note: the subroutine `lsq_fun` should be thoroughly tested before being supplied to this procedure. The components `verify` and `max_iter` of the optional argument `control` (i.e., `control%verify` and `control%max_iter`) can be used to assist this process (see the type definition for `nag_nlin_lsq_cntrl_`*wp*).

x(n**)** — real(kind=*wp*), intent(inout)

Input: an initial estimate of the solution.

Output: the point at which this procedure terminated. If `error%level` = 0, `x` contains x^* (an estimate of the solution).

f_sum_sq — real(kind=*wp*), intent(out)

Output: the value of $F(x)$ at the final point given in `x`.

f_vec(m**)** — real(kind=*wp*), intent(out)

Output: `f_vec(`i`)` contains the value of the residual $f_i(x)$ at the final point given in `x`, for $i = 1, 2, \ldots, m$.

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.

deriv — logical, intent(in), optional

Input: specifies whether or not first derivatives are provided in the user-supplied procedure `lsq_fun`.

If `deriv = .true.`, then first derivatives are assumed to be provided in `lsq_fun` via its argument `f_jac`;

if `deriv = .false.`, then it is assumed that no first derivatives are provided.

Default: `deriv = .true.`.

iter — integer, intent(out), optional

Output: the number of iterations which have been performed in this procedure.

f_jac(m, n) — real(kind=*wp*), intent(out), optional

Output: `f_jac(i,j)` contains the value of the first derivative $\partial f_i/\partial x_j$ at the final point given in `x`, for $i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$. If `deriv = .false.`, then `f_jac` contains a finite difference approximation to the first derivatives.

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

Output: the singular values of the Jacobian matrix at the final point given in `x`. Thus `s` may be useful as information about the structure of the problem.

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

Output: the matrix V associated with the singular value decomposition

$$J = USV^T$$

of the Jacobian matrix at the final point given in `x`, stored by rows. This matrix may be useful for statistical purposes, since it is the matrix of orthonormalised eigenvectors of $J^T J$.

control — type(nag_nlin_lsq_cntrl_*wp*), intent(in), optional

Input: a structure containing scalar components; these are used to alter the default values of those parameters which control the behaviour of the algorithm and level of printed output. The initialization of this structure and its use is described in the procedure document for `nag_nlin_lsq_cntrl_init`.

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	The user-supplied Jacobian matrix appears to be incorrect. As a first step, you should check that the code which defines the elements of the Jacobian matrix is correct, for example by computing them at a point where the correct values are known. However, care should be taken that the chosen point fully tests the evaluation of the whole matrix. It is remarkable how often the values $x = 0$ or $x = 1$ are used to test evaluation procedures, and how often the special properties of these numbers make the test meaningless.
202	User requested termination. This exit occurs if you have set `finish` to `.true.` in `lsq_fun`.
203	(This failure is not likely to occur.) The method for computing the singular value decomposition of the Jacobian matrix has failed to converge in a reasonable number of sub-iterations. It may be worth applying this procedure again starting with an initial approximation which is not too close to the point at which the failure occurred.

Warnings (error%level = 1):

error%code	Description
101	The limiting number of iterations (determined by the component `max_iter` of the optional argument `control` (i.e., `control%max_iter`)) has been reached. If steady reductions in the sum of squares, $F(x)$, were monitored up to the point where this exit occurred then `control%max_iter` was set too small, so the calculations should be restarted from the final point held in `x`. This exit may also indicate that $F(x)$ has no minimum.
102	The conditions for a minimum have not all been satisfied, but a lower point could not be found. This could be because the component `optim_tol` of the optional argument `control` (i.e., `control%optim_tol`) has been set so small that rounding errors in the evaluation of the residuals make attainment of the convergence conditions impossible.

The values 203, 101 and 102 may also be caused by mistakes in `lsq_fun`, by the formulation of the problem or by an awkward function. If there are no such mistakes, it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document. This example could be modified to use some (or all) of the optional arguments described in Section 3.2.

6 Further Comments

6.1 Termination Criteria

A successful exit (`error%level` = 0) is made from this procedure when (B1, B2 and B3) or B4 or B5 hold, where

$$\text{B1} \quad \equiv \alpha^{(k)} \times \|p^{(k)}\| < (\texttt{optim\_tol} + \texttt{EPSILON(1.0\_}wp\texttt{)}) \times \left(1.0 + \|x^{(k)}\|\right)$$

$$\text{B2} \equiv |F^{(k)} - F^{(k-1)}| < (\texttt{optim\_tol} + \texttt{EPSILON}(\texttt{1.0\_}wp))^2 \times \left(1.0 + F^{(k)}\right)$$

$$\text{B3} \equiv \|g^{(k)}\| < (\texttt{EPSILON}(\texttt{1.0\_}wp))^{1/3} \times \left(1.0 + F^{(k)}\right)$$

$$\text{B4} \equiv F^{(k)} < (\texttt{EPSILON}(\texttt{1.0\_}wp))^2$$

$$\text{B5} \equiv \|g^{(k)}\| < \left(\texttt{EPSILON}(\texttt{1.0\_}wp) \times \sqrt{F^{(k)}}\right)^{1/2}$$

and where $F^{(k)}$ and $g^{(k)}$ are the values of $F(x)$ and its vector of first derivatives at $x^{(k)}$, while the norm $\|.\|$ and the component `optim_tol` of the optional argument `control` (i.e., `control%optim_tol`) are as defined in the type definition for `nag_nlin_lsq_cntrl_wp`.

If `error%level` $= 0$ then the vector in `x` on exit, $x_{\rm sol}$, is almost certainly an estimate of $x_{\rm true}$, the position of the minimum to the accuracy specified by `control%optim_tol`.

If `error%code` $= 102$, then $x_{\rm sol}$ may still be a good estimate of $x_{\rm true}$, but to verify this you should make the following checks. If

(a) the sequence $\left\{F\left(x^{(k)}\right)\right\}$ converges to $F(x_{\rm sol})$ at a superlinear or a fast linear rate, and

(b) $g(x_{\rm sol})^T g(x_{\rm sol}) < 10 \times \texttt{EPSILON}(\texttt{1.0\_}wp)$,

then it is almost certain that $x_{\rm sol}$ is a close approximation to the minimum. When (b) is true, then usually $F(x_{\rm sol})$ is a close approximation to $F(x_{\rm true})$.

Further suggestions about confirmation of a computed solution are given in the Chapter Introduction.

6.2 Scaling

Ideally, the problem should be scaled so that, at the solution, $F(x)$ and the corresponding values of the x_j are each in the range $(-1, +1)$, and so that at points one unit away from the solution, $F(x)$ differs from its value at the solution by approximately one unit. This will usually imply that the Hessian matrix of $F(x)$ at the solution is well conditioned. It is unlikely that you will be able to follow these recommendations very closely, but it is worth trying as sensible scaling will reduce the difficulty of the minimization problem.

6.3 Accuracy

If the problem is reasonably well scaled and a successful exit is made, then, for a computer with a mantissa of t decimals, one would expect to get about $t/2 - 1$ decimals accuracy in the components of x and between $t - 1$ (if $F(x)$ is of order 1 at the minimum) and $2t - 2$ (if $F(x)$ is close to zero at the minimum) decimals accuracy in $F(x)$.

7 Description of Printed Output

This section describes the intermediate and final printout produced by this procedure. The level of printed output can be controlled via the components `list` and `print_level` of the optional argument `control` (i.e., `control%list` and `control%print_level`). For example, a listing of the parameter settings to be used by this procedure is output unless `control%list` is set to `.false.`. Note also that the intermediate printout and the final printout (in full) are produced only if `control%print_level` $\geq$ 10 (the default).

The intermediate printout produced by this procedure is as follows.

When `control%print_level` ≥ 5, the following line of output is produced at every iteration.

`Itn`	the current iteration number (k say).
`Step`	the step $\alpha^{(k)}$ taken along the computed search direction $p^{(k)}$.
`Nfun`	the cumulative number of calls to `lsq_fun`.

`Objective`	the current value of the objective function, $F(x^{(k)})$.
`Norm g`	the Euclidean norm of the gradient of $F(x^{(k)})$.
`Grade`	the grade of the Jacobian matrix, i.e., the dimension of the sub-space for which the Jacobian matrix can be used as a valid approximation to the curvature (see Gill and Murray [2]).

When `control%print_level` ≥ 20, the following output is produced at every iteration.

`x`	the current point $x^{(k)}$.
`g`	the current gradient of $F(x^{(k)})$.
`Singular values`	the singular values of the current approximation to the Jacobian matrix.

The final printout produced by this procedure is as follows.

When `control%print_level` $= 1$ or `control%print_level` ≥ 10, the following output is produced at the final iteration.

`x`	the final point x^*.
`g`	the gradient of $F(x^*)$.
`Singular values`	the singular values of the Jacobian matrix at the final point x^*.

When `control%print_level` > 0, details of the total number of iterations performed along with the final values of the objective function, the Euclidean norm of the gradient and the Euclidean norm of the residuals are also output.

Numerical values are output with a fixed number of digits: they are not guaranteed to be accurate to this precision.

Procedure: nag_nlin_lsq_cov

1 Description

nag_nlin_lsq_cov returns estimates of elements of the (symmetric) variance–covariance matrix C of the estimated regression coefficients for a nonlinear least-squares problem. The estimates are derived from the Jacobian of the functions $f_i(x)$ at the solution returned by nag_nlin_lsq_sol. (The functions $f_i(x)$ are often referred to as 'residuals'.)

This procedure is intended for use when the nonlinear least-squares function, $F(x) = \sum_{i=1}^{m} (f_i(x))^2$, represents the goodness of fit of a nonlinear model to observed data. It assumes that the Hessian of $F(x)$ at the solution can be adequately approximated by $2J^TJ$, where J is the Jacobian matrix whose (i,j)th element is $\partial f_i/\partial x_j$.

This procedure can be used to find either the diagonal elements of C, or the elements of the jth column of C, or the whole of C.

A description of the algorithm is given in the Mathematical Background section of this module document.

2 Usage

```
USE nag_nlin_lsq
CALL nag_nlin_lsq_cov(f_vec, s, v  [, 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 '**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.

$n \geq 1$ — the number of singular values
$m \geq n$ — the number of nonlinear functions or residuals

3.1 Mandatory Arguments

f_vec(m) — real(kind=*wp*), intent(in)
Input: the values of the residuals $f_1(x), \ldots, f_m(x)$, as returned by nag_nlin_lsq_sol.

s(n) — real(kind=*wp*), intent(in)
Input: the singular values of the Jacobian matrix J, as returned by nag_nlin_lsq_sol.

v(n, n) — real(kind=*wp*), intent(inout)
Input: the right singular vectors of J, as returned by nag_nlin_lsq_sol.
Output: overwritten by C if the optional arguments j and cj (see Section 3.2) are not present or if j is present and j = −1. Otherwise, v is unchanged.

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.

j — integer, intent(in), optional

Input: if j $= -1$, the whole of C is required. If j $= 0$, the diagonal elements of C are required. If j > 0, the elements of the jth column of C are required.

Constraints: $-1 \leq$ j $\leq$ n and cj must be present if j is present and j ≥ 0.

Default: if cj is not present then j $= -1$, and 0 otherwise.

cj(n) — real(kind=wp), intent(out), optional

Output: if j is present and j $= -1$, cj is not used. If j is present and j $= 0$ or j is not present, cj(i) contains the value of c_{ii}. If j is present and j > 0, cj(i) contains the value of c_{ij}.

rank — integer, intent(out), optional

Output: r, the assumed rank of the Jacobian matrix J. The value of r is computed by regarding singular values s(i) that are not larger than $10 \times$ EPSILON(1.0_wp)$\times$ s(1) as zero.

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

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

4 Error Codes

Fatal errors (error%level = 3):

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

Failures (error%level = 2):

error%code	Description
201	The singular values are all zero, so that at the solution the Jacobian matrix J has rank 0.

Warnings (error%level = 1):

error%code	Description
101	At the solution the Jacobian matrix J contains linear (or near linear) dependencies amongst its columns. In this case the required elements of C have still been computed based upon J having an assumed rank equal to rank (see Section 3.2).

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document. This example could be modified to use some (or all) of the optional arguments described in Section 3.2.

6 Further Comments

The computed elements of C will be the exact covariances corresponding to a closely neighbouring Jacobian matrix J.

6.1 Algorithmic Detail

If overflow occurs then either an element of C is very large, or the singular values and/or right singular vectors have been supplied incorrectly.

6.2 Timing

The time taken by the procedure is approximately proportional to n^3 when the whole of C is required, and approximately proportional to n^2 otherwise.

Procedure: nag_nlin_lsq_cntrl_init

1 Description

nag_nlin_lsq_cntrl_init assigns default values to all the structure components of the derived type nag_nlin_lsq_cntrl_*wp*.

2 Usage

```
USE nag_nlin_lsq
CALL nag_nlin_lsq_cntrl_init(control)
```

3 Arguments

3.1 Mandatory Argument

control — type(nag_nlin_lsq_cntrl_*wp*), intent(out)

Output: a structure containing the default values of those parameters which control the behaviour of the algorithm and level of printed output. A description of its components is given in the document for the derived type nag_nlin_lsq_cntrl_*wp*.

4 Error Codes

None.

5 Examples of Usage

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

Derived Type: nag_nlin_lsq_cntrl_*wp*

Note. The names of derived types containing real/complex components are precision dependent. For double precision the name of this type is nag_nlin_lsq_cntrl_dp. For single precision the name is nag_nlin_lsq_cntrl_sp. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

A structure of type nag_nlin_lsq_cntrl_*wp* is used to supply a number of optional parameters: these govern the level of printed output and a number of tolerances and limits, which allow you to influence the behaviour of the algorithm. If this structure is supplied then it *must* be initialized prior to use by calling the procedure nag_nlin_lsq_cntrl_init, which assigns default values to all the structure components. You may then assign required values to selected components of the structure (as appropriate).

2 Type Definition

The public components are listed below; components are grouped according to their function. A full description of the purpose of each component is given in Section 3.

```
type nag_nlin_lsq_cntrl_wp
  !  Printing parameters
  logical ::  list
  integer ::  unit
  integer ::  print_level
  !
  !  Algorithm choice and tolerances
  logical ::  lin_deriv
  real(kind=wp) ::  linesearch_tol
  real(kind=wp) ::  step_max
  real(kind=wp) ::  optim_tol
  integer ::  max_iter
  logical ::  verify
end type nag_nlin_lsq_cntrl_wp
```

3 Components

3.1 Printing Parameters

list — logical

Controls the printing of the parameter settings in the call to nag_nlin_lsq_sol as follows.

If list = .true., then the parameter settings are printed;

if list = .false., then the parameter settings are not printed.

Default: list = .true..

unit — integer

Specifies the Fortran unit number to which all output produced by nag_nlin_lsq_sol is sent.

Default: unit = the default Fortran unit number for your implementation.

Constraints: a valid output unit.

print_level — integer

Controls the amount of output produced by nag_nlin_lsq_sol, as indicated below. A detailed description of the printed output is given in Section 7 of the procedure document for nag_nlin_lsq_sol.

The following output is sent to the Fortran unit number defined by `unit`:

$\leq$	0	No output.
	1	The final solution only.
$\geq$	5	One line of output for each iteration (no printout of the final solution).
$\geq$	10	The final solution and one line of output for each iteration.
$\geq$	20	The final solution and one line of output for each iteration in addition to the variables, the gradient and the singular values of the Jacobian matrix at each iteration.

Default: `print_level` = 10.

3.2 Algorithm Choice and Tolerances

lin_deriv — logical

Note: `lin_deriv` is ignored if the optional argument `deriv` (see Section 3.2 of the procedure document for `nag_nlin_lsq_sol`) has been supplied and set to `.false.`.

`lin_deriv` specifies whether the linear minimizations (i.e., minimizations of $F(x^{(k)} + \alpha^{(k)}p^{(k)})$ with respect to $\alpha^{(k)}$) are to be performed by a procedure which only requires the evaluation of the $f_i(x)$ (`lin_deriv` = `.false.`), or by a procedure which also requires the first derivatives of the $f_i(x)$ (`lin_deriv` = `.true.`).

It will often be possible to evaluate the first derivatives of the residuals in about the same amount of computer time that is required for the evaluation of the residuals themselves; if this is so then `nag_nlin_lsq_sol` should be used with `lin_deriv` = `.true.` (the default). However, if the evaluation of the derivatives takes more than about 4 times as long as the evaluation of the residuals, then a setting of `lin_deriv` = `.false.` will usually be preferable (although the default setting is slightly more robust).

Default: `lin_deriv` = `.true.`.

linesearch_tol — real(kind=*wp*)

`linesearch_tol` specifies how accurately the linear minimizations are to be performed.

Every iteration of `nag_nlin_lsq_sol` involves the minimization of $F(x^{(k)} + \alpha^{(k)}p^{(k)})$ with respect to $\alpha^{(k)}$. The minimum with respect to $\alpha^{(k)}$ will be located more accurately for small values of `linesearch_tol` (say 0.01) than for large values (say 0.9).

Although accurate linear minimizations will generally reduce the number of iterations performed by `nag_nlin_lsq_sol`, they will increase the number of calls made to `lsq_fun` on each iteration. On balance it is usually more efficient to perform a low-accuracy minimization.

Default: if the optional argument `deriv` (see Section 3.2 of the procedure document for `nag_nlin_lsq_sol`) has been supplied and set to `.false.` or `lin_deriv` is set to `.false.`, then the default value of `linesearch_tol` is 0.5 for a multivariate problem and 0.0 for a univariate problem. If `deriv` and `lin_deriv` are both `.true.` (the default), then the default value of `linesearch_tol` is 0.9 for a multivariate problem and 0.0 univariate problem.

Constraints: $0.0 \leq$ `linesearch_tol` < 1.0.

step_max — real(kind=*wp*)

`step_max` specifies an estimate of the Euclidean distance between the solution and the starting point. (For maximum efficiency, a slight overestimate is preferable.)

`nag_nlin_lsq_sol` will ensure that, for each iteration,

$$\sum_{j=1}^{n}(x_j^{(k)} - x_j^{(k-1)})^2 \leq (\texttt{step\_max})^2$$

where k is the iteration number. Thus, if the problem has more than one solution, `nag_nlin_lsq_sol` is most likely to find the one nearest to the starting point. On difficult problems, a realistic choice

can prevent the sequence of $x^{(k)}$ entering a region where the problem is ill behaved and can help avoid overflow in the evaluation of $F(x)$. However, an underestimate of step_max can lead to inefficiency.

Default: step_max = 100000.0.

Constraints: step_max $\geq$ optim_tol.

optim_tol — real(kind=*wp*)

optim_tol specifies the accuracy in x to which the solution is required. If x_{true} is the true value of x at the minimum, then x_{sol}, the estimated position prior to a normal exit, is such that

$$\|x_{\text{sol}} - x_{\text{true}}\| < \texttt{optim\_tol} \times (1.0 + \|x_{\text{true}}\|),$$

where $\|y\| = \sqrt{\sum_{j=1}^{n} y_j^2}$.

For example, if the elements of x_{sol} are not much larger than 1.0 in modulus and if optim_tol = 0.00001, then x_{sol} is usually accurate to about 5 decimal places. (For further details see the Mathematical Background section of this module document.) If $F(x)$ and the variables are scaled roughly as described in Section 6 of the procedure document for nag_nlin_lsq_sol, then the default setting will usually be appropriate.

Default: optim_tol = SQRT(EPSILON(1.0_*wp*)).

Constraints: 10×EPSILON(1.0_*wp*) $\leq$ optim_tol $<$ 1.0.

max_iter — integer

max_iter specifies the maximum number of iterations allowed before termination.

If you wish to check that a call to nag_nlin_lsq_sol is correct before attempting to solve the problem in full then max_iter may be set to 0. No iterations will be performed but the initialization stages prior to the first iteration will be processed and a listing of parameter settings output if list = .true. (the default). Any derivative checking (as specified by verify) will also be performed.

Default: max_iter = max(50, 5 × number of variables).

Constraints: max_iter $\geq$ 0.

verify — logical

Note: verify is ignored if the optional argument deriv (see Section 3.2 of the procedure document for nag_nlin_lsq_sol) has been supplied and set to .false..

Input: if verify = .true., then a check of the derivatives defined by lsq_fun will be made at the starting point x.

A starting point of $x = 0$ or $x = 1$ should be avoided if this test is to be meaningful, but if either of these starting points is necessary then nag_nlin_lsq_sol should initially be called at an alternative point but with max_iter set to zero. If this test is successfully passed then the optimization process can be restarted from the original starting point with max_iter reset to a value > 0 and verify set to .false..

Default: verify = .true..

Example 1: Minimization with derivatives

To find least-squares estimates of x_1, x_2 and x_3 in the model

$$y = x_1 + \frac{t_1}{x_2 t_2 + x_3 t_3}$$

using the 15 sets of data given in the following table.

y	0.14	0.18	0.22	0.25	0.29	0.32	0.35	0.39	0.37	0.58	0.73	0.96	1.34	2.10	4.39
t_1	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0
t_2	15.0	14.0	13.0	12.0	11.0	10.0	9.0	8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0
t_3	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0

The initial point is $x^{(0)} = (0.5,\ 1.0,\ 1.5)^T$.

The optimal solution (to five figures) is $x^* = (0.0082410,\ 1.13304,\ 2.3437)^T$.

The data sets for y and t are stored in the arrays `y` and `t` which are declared in the module `nlin_lsq_ex01_mod` to allow communication between the main program and the procedure `lsq_fun`. The data is read into the arrays in the main program but accessed from `lsq_fun`, the procedure being defined in the module. Note that a USE statement for the user-defined module *must* be included in the main program.

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.

```
MODULE nlin_lsq_ex01_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: t(:,:), y(:)

CONTAINS

  SUBROUTINE lsq_fun(x,finish,f_vec,f_jac)

    ! Procedure to evaluate the residuals and optionally their 1st
    ! derivatives. The procedure may also be used when the
    ! linesearch to be used does not require derivatives (see
    ! the optional argument control), since it can deal with the
    ! presence or absence of f_jac.

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC PRESENT, SIZE
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    ! .. Array Arguments ..
    REAL (wp), OPTIONAL, INTENT (OUT) :: f_jac(:,:)
    REAL (wp), INTENT (OUT) :: f_vec(:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Local Scalars ..
    INTEGER :: i
    REAL (wp) :: denom, dummy
```

```
    ! .. Executable Statements ..

    IF (PRESENT(f_jac)) THEN
      DO i = 1, SIZE(f_vec)
        denom = x(2)*t(i,2) + x(3)*t(i,3)
        f_vec(i) = x(1) + t(i,1)/denom - y(i)
        f_jac(i,1) = 1.0_wp
        dummy = -1.0_wp/(denom*denom)
        f_jac(i,2) = t(i,1)*t(i,2)*dummy
        f_jac(i,3) = t(i,1)*t(i,3)*dummy
      END DO
    ELSE
      DO i = 1, SIZE(f_vec)
        denom = x(2)*t(i,2) + x(3)*t(i,3)
        f_vec(i) = x(1) + t(i,1)/denom - y(i)
      END DO
    END IF

  END SUBROUTINE lsq_fun

END MODULE nlin_lsq_ex01_mod

PROGRAM nag_nlin_lsq_ex01

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_nlin_lsq, ONLY : nag_nlin_lsq_sol
  USE nlin_lsq_ex01_mod, ONLY : lsq_fun, t, y, wp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Local Scalars ..
  INTEGER :: i, m, n, r
  REAL (wp) :: f_sum_sq
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: f_vec(:), x(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_nlin_lsq_ex01'

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

  ! Read number of observations (m), variables (n) and controls (r)
  READ (nag_std_in,*) m, n, r

  ALLOCATE (f_vec(m),x(n),t(m,r),y(m)) ! Allocate storage

  ! Read in data for y and corresponding controls t(1,2,...,r)
  READ (nag_std_in,*) (y(i),t(i,:),i=1,m)

  ! Read the starting point x
  READ (nag_std_in,*) x

  ! Solve the problem
  CALL nag_nlin_lsq_sol(lsq_fun,x,f_sum_sq,f_vec)

  DEALLOCATE (f_vec,x,t,y)     ! Deallocate storage

END PROGRAM nag_nlin_lsq_ex01
```

2 Program Data

```
Example Program Data for nag_nlin_lsq_ex01
 15  3  3                  :Values of m, n, r
 0.14    1.00 15.00  1.00
 0.18    2.00 14.00  2.00
 0.22    3.00 13.00  3.00
 0.25    4.00 12.00  4.00
 0.29    5.00 11.00  5.00
 0.32    6.00 10.00  6.00
 0.35    7.00  9.00  7.00
 0.39    8.00  8.00  8.00
 0.37    9.00  7.00  7.00
 0.58   10.00  6.00  6.00
 0.73   11.00  5.00  5.00
 0.96   12.00  4.00  4.00
 1.34   13.00  3.00  3.00
 2.10   14.00  2.00  2.00
 4.39   15.00  1.00  1.00  :End of (y(i), (t(i,j), j=1,r), i=1,m)
 0.50  1.00  1.50          :End of x
```

3 Program Results

```
 Example Program Results for nag_nlin_lsq_ex01

 Parameters
 ----------

 number of residuals (m)        15       number of variables (n)         3

 list...................     .true.       print_level............        10
 lin_deriv..............     .true.       linesearch_tol.........  9.00E-01
 step_max...............  1.00E+05        optim_tol..............  1.49E-08

 deriv..................     .true.       verify.................     .true.
 max_iter...............        50        unit...................         6

 Verification of the Jacobian matrix
 -----------------------------------

 The Jacobian matrix seems to be ok.


 Intermediate Results
 --------------------

  Itn      Step  Nfun      Objective    Norm g    Grade
    0              1   1.021037E+01   3.2E+01        3
    1   1.0E+00    2   1.987296E-01   2.8E+00        3
    2   1.0E+00    3   9.232381E-03   1.9E-01        3
    3   1.0E+00    4   8.214916E-03   1.2E-03        3
    4   1.0E+00    5   8.214877E-03   5.0E-08        2
    5   1.0E+00    6   8.214877E-03   4.7E-09        0
    6   1.0E+00    7   8.214877E-03   1.2E-09        0

 Final Result
 ------------

       x                 g         Singular values
  8.24106E-02          1.2E-09          4.1E+00
  1.13304E+00         -1.9E-11          1.6E+00
  2.34370E+00          1.8E-11          6.1E-02
```

```
exit from nag_nlin_lsq after     6 iterations.

final objective value =    0.8214877E-02

final residual norm =   9.1E-02

final gradient norm =   1.2E-09
```

Example 2: Estimation of the variance–covariance matrix C

To estimate the variance–covariance matrix C for the least-squares estimates of x_1, x_2 and x_3 in the model

$$y = x_1 + \frac{t_1}{x_2 t_2 + x_3 t_3}$$

using the data given in Example 1. The least-squares solution is computed using the procedure `nag_nlin_lsq_sol`.

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.

```
MODULE nlin_lsq_ex02_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: t(:,:), y(:)

CONTAINS

  SUBROUTINE lsq_fun(x,finish,f_vec,f_jac)

    ! Procedure to evaluate the residuals and optionally their 1st
    ! derivatives. The procedure may also be used when the
    ! linesearch to be used does not require derivatives
    ! (see the optional argument control), since it can deal
    ! with the presence or absence of f_jac.

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC PRESENT, SIZE
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    ! .. Array Arguments ..
    REAL (wp), OPTIONAL, INTENT (OUT) :: f_jac(:,:)
    REAL (wp), INTENT (OUT) :: f_vec(:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Local Scalars ..
    INTEGER :: i
    REAL (wp) :: denom, dummy
    ! .. Executable Statements ..

    IF (PRESENT(f_jac)) THEN
      DO i = 1, SIZE(f_vec)
        denom = x(2)*t(i,2) + x(3)*t(i,3)
        f_vec(i) = x(1) + t(i,1)/denom - y(i)
        f_jac(i,1) = 1.0_wp
        dummy = -1.0_wp/(denom*denom)
        f_jac(i,2) = t(i,1)*t(i,2)*dummy
        f_jac(i,3) = t(i,1)*t(i,3)*dummy
      END DO
    ELSE
      DO i = 1, SIZE(f_vec)
```

```
        denom = x(2)*t(i,2) + x(3)*t(i,3)
        f_vec(i) = x(1) + t(i,1)/denom - y(i)
      END DO
    END IF

  END SUBROUTINE lsq_fun

END MODULE nlin_lsq_ex02_mod

PROGRAM nag_nlin_lsq_ex02

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_nlin_lsq, ONLY : nag_nlin_lsq_sol, nag_nlin_lsq_cov, &
   nag_nlin_lsq_cntrl_init, nag_nlin_lsq_cntrl_wp => nag_nlin_lsq_cntrl_dp
  USE nag_write_mat, ONLY : nag_write_gen_mat
  USE nlin_lsq_ex02_mod, ONLY : lsq_fun, t, y, wp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Local Scalars ..
  INTEGER :: i, m, n, r
  REAL (wp) :: f_sum_sq
  TYPE (nag_nlin_lsq_cntrl_wp) :: control
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: f_vec(:), s(:), v(:,:), x(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_nlin_lsq_ex02'

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

  ! Read number of observations (m), variables (n) and controls (r)
  READ (nag_std_in,*) m, n, r

  ALLOCATE (f_vec(m),x(n),t(m,r),y(m),s(n),v(n,n)) ! Allocate storage

  ! Read in data for y and corresponding controls t(1,2,...,r)
  READ (nag_std_in,*) (y(i),t(i,:),i=1,m)

  ! Read the starting point x
  READ (nag_std_in,*) x

  ! initialize control structure and set required control parameters
  CALL nag_nlin_lsq_cntrl_init(control)

  control%print_level = 1

  ! Solve the problem
  CALL nag_nlin_lsq_sol(lsq_fun,x,f_sum_sq,f_vec,s=s,v=v,control=control)

  ! Compute the estimated variance-covariance matrix C
  CALL nag_nlin_lsq_cov(f_vec,s,v)

  WRITE (nag_std_out,*)

  CALL nag_write_gen_mat(v,title='variance-covariance matrix C')

  DEALLOCATE (f_vec,s,t,v,x,y) ! Deallocate storage
```

```
    END PROGRAM nag_nlin_lsq_ex02
```

2 Program Data

```
Example Program Data for nag_nlin_lsq_ex02
 15  3  3                  :Values of m, n, r
 0.14    1.00 15.00  1.00
 0.18    2.00 14.00  2.00
 0.22    3.00 13.00  3.00
 0.25    4.00 12.00  4.00
 0.29    5.00 11.00  5.00
 0.32    6.00 10.00  6.00
 0.35    7.00  9.00  7.00
 0.39    8.00  8.00  8.00
 0.37    9.00  7.00  7.00
 0.58   10.00  6.00  6.00
 0.73   11.00  5.00  5.00
 0.96   12.00  4.00  4.00
 1.34   13.00  3.00  3.00
 2.10   14.00  2.00  2.00
 4.39   15.00  1.00  1.00  :End of (y(i), (t(i,j), j=1,r), i=1,m)
 0.50  1.00  1.50          :End of x
```

3 Program Results

```
 Example Program Results for nag_nlin_lsq_ex02

 Parameters
 ----------

 number of residuals (m)        15       number of variables (n)         3

 list...................    .true.       print_level............         1
 lin_deriv..............    .true.       linesearch_tol.........  9.00E-01
 step_max...............  1.00E+05       optim_tol..............  1.49E-08

 deriv..................    .true.       verify.................    .true.
 max_iter...............        50       unit...................         6

 Verification of the Jacobian matrix
 -----------------------------------

 The Jacobian matrix seems to be ok.


 Final Result
 ------------

       x                  g         Singular values
  8.24106E-02          1.2E-09          4.1E+00
  1.13304E+00         -1.9E-11          1.6E+00
  2.34370E+00          1.8E-11          6.1E-02

 exit from nag_nlin_lsq after     6 iterations.

 final objective value =    0.8214877E-02

 final residual norm =   9.1E-02

 final gradient norm =   1.2E-09
```

```
 variance-covariance matrix C
     1.5312E-04   2.8698E-03  -2.6565E-03
     2.8698E-03   9.4802E-02  -9.0983E-02
    -2.6565E-03  -9.0983E-02   8.7781E-02
```

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_nlin_lsq_ex03`

Minimization without derivatives.

`nag_nlin_lsq_ex04`

Estimation of the diagonal elements of the variance–covariance matrix C.

Mathematical Background

1 Description

`nag_nlin_lsq_sol` minimizes a sum of squares of m nonlinear functions each with n variables, that is problems of the form:

$$\text{Minimize } F(x) = \sum_{i=1}^{m} (f_i(x))^2 \tag{1}$$

where $x = (x_1, x_2, \ldots, x_n)^T$ and $m \geq n$.

From a user-supplied starting point $x^{(0)}$, `nag_nlin_lsq_sol` generates a sequence of points $x^{(1)}$, $x^{(2)}, \ldots$, which is intended to converge to a local minimum of $F(x)$. The sequence of points is given by

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$$

where the vector $p^{(k)}$ is a direction of search, and $\alpha^{(k)}$ is chosen such that $F(x^{(k)} + \alpha^{(k)} p^{(k)})$ is approximately a minimum with respect to $\alpha^{(k)}$.

The vector $p^{(k)}$ used depends upon the reduction in the sum of squares obtained during the last iteration. If the sum of squares was sufficiently reduced, then $p^{(k)}$ is the Gauss–Newton direction; otherwise the second derivatives of the $f_i(x)$ are taken into account using a quasi-Newton updating scheme.

The method is designed to ensure that steady progress is made whatever the starting point, and to have the rapid ultimate convergence of Newton's method.

The number of iterations required depends on the number of variables, the number of residuals, the behaviour of $F(x)$, the accuracy demanded and the distance of the starting point from the solution. The number of multiplications performed per iteration of `nag_nlin_lsq_sol` varies, but for $m \gg n$ is approximately $nm^2 + O(n^3)$. In addition, each iteration makes at least one call of `lsq_fun`. So, unless the residuals can be evaluated very quickly, the run time will be dominated by the time spent in `lsq_fun`.

When the sum of squares represents the goodness of fit of a nonlinear model to observed data, elements of the variance–covariance matrix of the estimated regression coefficients can subsequently be computed by calling `nag_nlin_lsq_cov` (using information returned by the procedure `nag_nlin_lsq_sol` in the mandatory argument `f_vec` and the optional arguments `s` and `v`).

From (1), the Hessian matrix $G(x) = \nabla^2 F(x)$ is of the form

$$G(x) = 2\left(J(x)^T J(x) + \sum_{i=1}^{m} f_i(x) G_i(x)\right),$$

where $J(x)$ is the Jacobian matrix of $f(x)$, and $G_i(x)$ is the Hessian matrix of $f_i(x)$. In the neighbourhood of a solution, $\|f(x)\|$ is often small compared to $\|J(x)^T J(x)\|$. For example, when $f(x)$ represents the goodness of fit of a nonlinear model to observed data. `nag_nlin_lsq_cov` is intended for use in such situations. It assumes that $2J(x)^T J(x)$ is an adequate approximation to $G(x)$, thereby avoiding the need to compute or approximate second derivatives of $\{f_i(x)\}$. For further information see Section 4.7 of Gill *et al.* [3].

The estimated variance–covariance matrix C is then given by

$$C = \sigma^2 (J^T J)^{-1} \quad \text{when} \quad J^T J \quad \text{is non-singular,}$$

where σ^2 is the estimated variance of the residual at the computed solution $\hat{x}$, given by

$$\sigma^2 = \frac{F(\hat{x})}{m-n} \quad \text{if} \quad m > n, \quad \text{and 0 if } m = n.$$

The diagonal (off-diagonal) elements of C are estimates of the variances (covariances) of the estimated regression coefficients. See Bard [1] and Wolberg [4] for further information on the use of C.

When J^TJ is singular then C is taken to be

$$C = \sigma^2(J^TJ)^\dagger,$$

where $(J^TJ)^\dagger$ is the pseudo-inverse of J^TJ, and

$$\sigma^2 = \frac{F(\hat{x})}{m-r},\ r = \text{rank}(J)$$

but in this case `error%level` $= 1$ on exit to warn you that J has linear dependencies in its columns. The assumed rank of J can be obtained directly from the optional argument `rank` (see Section 3.2 of the procedure document for `nag_nlin_lsq_cov`).

Suppose that $\hat{G} = 2J^TJ$ is an adequate approximation to G at $\hat{x}$ and let $H = \hat{G}^{-1}$. If x^* is the true solution, then the $100(1-\beta)\%$ confidence interval on $\hat{x}$ is

$$\hat{x_i} - \sqrt{c_{ii}}.t_{(1-\beta/2,m-n)} < x_i^* < \hat{x_i} + \sqrt{c_{ii}}.t_{(1-\beta/2,m-n)}, \quad i = 1,2,\ldots,n$$

where $t_{(1-\beta/2,m-n)}$ is the $100(1-\beta)/2$ percentage point of the t-distribution with $m-n$ degrees of freedom.

In the majority of problems, the residuals f_i, for $i = 1,2,\ldots,m$, contain the difference between the values of a model function $\phi(z,x)$ calculated for m different values of the independent variable z, and the corresponding observed values at these points. The minimization process determines the parameters, or constants x, of the fitted function $\phi(z,x)$. For any value, z_0, of the independent variable z, an unbiased estimate of the variance of ϕ is

$$\text{var } \phi = \frac{2F(\hat{x})}{m-n}\sum_{i=1}^{n}\sum_{k=1}^{n}\left[\frac{\partial\phi}{\partial x_i}\right]_{z_0}\left[\frac{\partial\phi}{\partial x_k}\right]_{z_0} h_{ik}.$$

The $100(1-\beta)\%$ confidence interval on F at the point z_0 is

$$\phi(z_0,\hat{x}) - \sqrt{\text{var } \phi}.t_{(\beta/2,m-n)} < \phi(z_0,x^*) < \phi(z_0,\hat{x}) + \sqrt{\text{var } \phi}.t_{(\beta/2,m-n)}.$$

For further details on the analysis of least-squares solutions see Bard [1] and Wolberg [4].

References

[1] Bard Y (1974) *Nonlinear Parameter Estimation* Academic Press

[2] Gill P E and Murray W (1978) Algorithms for the solution of the nonlinear least-squares problem *SIAM J. Numer. Anal.* **15** 977–992

[3] Gill P E, Murray W and Wright M H (1981) *Practical Optimization* Academic Press

[4] Wolberg J R (1967) *Prediction Analysis* Van Nostrand