Module 9.4: nag_con_nlin_lsq
Constrained Nonlinear Least-squares

`nag_con_nlin_lsq` contains procedures for solving constrained nonlinear least-squares problems.

Contents

Introduction

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

- `nag_con_nlin_lsq_sol`
  Please note that this procedure is scheduled for withdrawal from the Library at a future release. Computes a constrained minimum of a smooth (nonlinear) sum of squares function subject to a set of constraints (which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints), using a sequential quadratic programming (SQP) method. It may also be used for unconstrained, bound-constrained and linearly constrained optimization. As many first derivatives as possible should be supplied by the user; any unspecified derivatives are approximated by finite differences. It treats all matrices as dense and hence is not intended for large sparse problems.

- `nag_con_nlin_lsq_sol_1`
  Supersedes `nag_con_nlin_lsq_sol`, which will be withdrawn from the Library at a future release. Computes a constrained minimum of a smooth (nonlinear) sum of squares function subject to a set of constraints, but avoids unnecessary function evaluations whilst verifying and/or approximating derivatives by finite differences.

- `nag_con_nlin_lsq_cntrl_init` assigns default values to the components of a structure of the derived type `nag_con_nlin_lsq_cntrl_`*wp*.

- `nag_con_nlin_lsq_cntrl_`*wp* may be used to supply optional parameters to the procedures `nag_con_nlin_lsq_sol` and `nag_con_nlin_lsq_sol_1`.

Procedure: nag_con_nlin_lsq_sol

Please note that this procedure is scheduled for withdrawal from the Library at a future release.

1 Description

nag_con_nlin_lsq_sol is designed to solve a nonlinear least-squares problem — minimizing a smooth (nonlinear) sum of squares function subject to constraints on the variables.

The problem is assumed to be stated in the following form:

$$\underset{x \in R^n}{\text{minimize}} \;\; F(x) = \tfrac{1}{2}\sum_{i=1}^{m}(f_i(x) - y_i)^2 \;\; \text{subject to} \;\; l \le \left\{ \begin{array}{c} x \\ A_{\mathrm{L}}x \\ c(x) \end{array} \right\} \le u, \tag{1}$$

where $F(x)$ is a nonlinear objective function, the $f_i(x)$ are subfunctions, the y_i are constant and the constraints are grouped as follows:

n *simple bounds* on the variables x;

n_{L} *linear constraints*, defined by the n_{L} by n constant matrix A_{L};

n_{N} *nonlinear constraints*, defined by the vector $c(x)$ of constraint functions.

(The functions $f_i(x) - y_i$ are often referred to as 'residuals'.) The subfunctions and the constraint functions are assumed to be smooth, i.e., at least twice-continuously differentiable. (The method used by this procedure will usually solve (1) if there are only isolated discontinuities away from the solution.)

The simple bounds on the variables, the linear constraints and the nonlinear constraints are distinguished from one another for reasons of computational efficiency (although the simple bounds could have been included in the definition of the linear constraints, and the linear constraints in the definition of the nonlinear constraints). There may be no linear constraints, in which case the matrix A_{L} is empty ($n_{\mathrm{L}} = 0$), or no nonlinear constraints, in which case the vector $c(x)$ is empty ($n_{\mathrm{N}} = 0$).

Upper bounds and/or lower bounds can be specified separately for the variables and constraints. An *equality* constraint can be specified by setting $l_i = u_i$. If certain bounds are not present, the associated elements of l and u can be set to special values that will be treated as $-\infty$ or $+\infty$.

You must supply an initial estimate of the solution to (1), together with a procedure obj_fun that defines the subfunctions $f(x)$ (see Section 3.1), and (if $n_{\mathrm{N}} > 0$) a procedure con_fun which defines the nonlinear constraint functions $c(x)$ (see Section 3.2). On every call, these procedures must return values of $f(x)$ and $c(x)$, and as many partial derivatives as possible. For maximum reliability, you should provide all partial derivatives (see Chapter 8 of Gill *et al.* [10] for a detailed discussion). Any derivatives which are not provided are approximated by finite differences.

Several options are available for controlling the operation of this procedure, covering facilities such as:

printed output, at the end of each iteration and at the final solution;

verifying or estimating partial derivatives;

algorithmic parameters, such as tolerances and iteration limits.

These options are grouped together in the optional argument control, which is a structure of the derived type nag_con_nlin_lsq_cntrl_*wp*.

The method used by this procedure is described in detail in the Mathematical Background section of this module document.

2 Usage

```
USE nag_con_nlin_lsq

CALL nag_con_nlin_lsq_sol(obj_fun, x, obj_f, f  [, 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.

$m \geq 1$ — the number of subfunctions or 'residuals'

$n \geq 1$ — the number of variables

$n_{\mathrm{L}} \geq 0$ — the number of linear constraints

$n_{\mathrm{N}} \geq 0$ — the number of nonlinear constraints

3.1 Mandatory Arguments

obj_fun — subroutine

The procedure `obj_fun`, supplied by the user, must calculate the vector $f(x)$ of subfunctions and (optionally) its Jacobian ($= \partial f/\partial x$) at a specified point x.

Its specification is:

```
subroutine obj_fun(first_call, x, finish, f, f_jac)
```

`logical, intent(in) :: first_call`

Input: `first_call` will be `.true.` when this procedure calls `obj_fun` for the first time, and `.false.` for all subsequent calls. It allows you to save computation time if certain data must be read or calculated only once. See also the description of `f_jac`.

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

Shape: `x` has shape (n).

Input: the point x at which the subfunctions and (optionally) elements of the objective Jacobian are to be evaluated.

`logical, intent(inout) :: finish`

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

Output: if you wish to terminate the call to this procedure, you should set `finish` to `.true.`, and then this procedure will terminate with `error%code` = 201.

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

Shape: `f` has shape (m).

Output: `f(`i`)` must contain the value of the ith subfunction f_i at the point x, for $i = 1, 2, \ldots, m$.

real(kind=*wp*), intent(inout), optional :: f_jac(:,:)

Shape: f_jac has shape (m, n).

Input: if f_jac is present, its elements must remain unchanged except as specified below.

Output: if f_jac is present, then:

if f_deriv = .true.(the default; see Section 3.2), the ith row of f_jac must contain *all* the elements of the vector ∇f_i given by

$$\nabla f_i = \left(\frac{\partial f_i}{\partial x_1}, \frac{\partial f_i}{\partial x_2}, \ldots, \frac{\partial f_i}{\partial x_n} \right)^T,$$

where $\partial f_i / \partial x_j$ is the partial derivative of the ith subfunction with respect to the jth variable evaluated at the point x, for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. Constant elements need be loaded into f_jac only during the first call to obj_fun (when first_call = .true.). This facility is useful when many Jacobian elements are identically zero, in which case f_jac may be initialized to zero during the first call to obj_fun. Note that although a constant non-zero element f_jac(i, j) only needs to be set on the first call to obj_fun, the corresponding i in the definition of f(i) *must* be re-evaluated each time that obj_fun is called.

If f_deriv = .false., any *available* partial derivatives of $f_i(x)$ must be assigned to the corresponding elements in the ith row of f_jac; the remaining elements *must remain unchanged*. Just before obj_fun is called, each element of f_jac is set to a special value. On return from this procedure, any element that retains the value is estimated by finite differences, at non-trivial expense. If you do not supply a value for control%diff_int (see the type definition for nag_con_nlin_lsq_cntrl_*wp*), an interval for each element of x is computed automatically at the start of the optimization. The automatic procedure can usually identify constant elements of f_jac, which are then computed once only by finite differences.

Note: obj_fun should be thoroughly tested before being supplied to this procedure. The components cheap_test, obj_verify and major_iter_lim of the optional argument control can be used to assist this process (see the type definition of nag_con_nlin_lsq_cntrl_*wp*).

x(n) — real(kind=*wp*), intent(inout)

Input: an initial estimate of the solution.

Output: the final estimate of the solution.

obj_f — real(kind=*wp*), intent(out)

Output: the value of the objective function at the final iterate.

f(m) — real(kind=*wp*), intent(out)

Output: f(i) contains the value of the ith subfunction f_i at the final iterate, 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.

f_deriv — logical, intent(in), optional

Input: specifies whether or not all elements of the objective Jacobian are provided by the user-supplied procedure obj_fun.

If f_deriv = .true. (the default), then all elements of the objective Jacobian must be provided by obj_fun via its argument f_jac.

If f_deriv = .false., then it is assumed that some elements of the objective Jacobian are not provided; this procedure will estimate them using finite differences. The computation of finite difference approximations usually increases the total run-time, since a call to obj_fun is needed for each variable for which partial derivatives are estimated. For example, if the Jacobian has the form

$$\begin{pmatrix} * & * & * & * & * \\ * & ? & ? & * & * \\ * & * & ? & * & * \\ * & * & * & * & * \end{pmatrix}$$

where '$*$' indicates an element provided by the user and '?' indicates an element to be estimated, this procedure will call obj_fun twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1, 4 and 5 are known, they require no calls to obj_fun.) Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* [10] for a discussion of limiting accuracy). At times, central differences are used rather than forward differences, in which case twice as many calls to obj_fun are needed. (The switch to central differences is determined by considerations of accuracy and is not under user control.)

f_deriv = .true. should be used whenever possible, since this procedure is more reliable (and will usually be more efficient) when all derivatives are exact.

Default: f_deriv = .true..

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

Output: the Jacobian matrix of the subfunctions at the final iterate (or its finite difference approximation), i.e., f_jac(i, j) contains the value of the partial derivative $\partial f_i/\partial x_j$ at the final point given in x, for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

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

Input: the coefficients of the constant vector y.

Default: y = 0.0.

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

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

Input: the lower and upper bounds on all the variables. To specify a non-existent lower bound (i.e., $l_j = -\infty$), set x_lower(j) $\leq$ −control%inf_bound; to specify a non-existent upper bound (i.e., $u_j = +\infty$), set x_upper(j) $\geq$ +control%inf_bound (see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

Constraints:

x_lower(j) $\leq$ x_upper(j) for $j = 1, 2, \ldots, n$;

$|\ \beta\ |$ < control%inf_bound when x_lower(j) = x_upper(j) = β.

Default: x_lower = −control%inf_bound; x_upper = +control%inf_bound.

a(n_L, n) — real(kind=*wp*), intent(in), optional

Input: the ith row of a must contain the coefficients of the ith linear constraint, for $i = 1, 2, \ldots, n_L$.

Default: the problem contains no linear constraints (i.e., $n_L = 0$).

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

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

Input: the lower and upper bounds on all the linear constraints. To specify a non-existent lower bound (i.e., $l_j = -\infty$), set lin_lower(j) $\leq$ −control%inf_bound; to specify a non-existent upper bound (i.e., $u_j = +\infty$), set lin_upper(j) $\geq$ +control%inf_bound (see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

Constraints:

`lin_lower` and `lin_upper` must not be present unless `a` is present;

`lin_lower`$(j) \leq$ `lin_upper`(j) for $j = 1, 2, \ldots, n_{\text{L}}$;

$|\beta| <$ `control%inf_bound` when `lin_lower`$(j) =$ `lin_upper`$(j) = \beta$.

Default: `lin_lower` = −`control%inf_bound`; `lin_upper` = +`control%inf_bound`.

num_nlin_con — integer, intent(in), optional

Input: the number of nonlinear constraints, n_{N}.

Constraints: `num_nlin_con` must be present if `con_fun` is present; `num_nlin_con` ≥ 0.

Default: `num_nlin_con` = 0.

con_deriv — logical, intent(in), optional

Input: specifies whether or not all elements of the constraint Jacobian are provided by the user-supplied procedure `con_fun`.

If `con_deriv` = `.true.` (the default), then all elements of the constraint Jacobian must be provided by `con_fun` via its argument `con_jac`.

If `con_deriv` = `.false.`, then it is assumed that some elements of the constraint Jacobian are not provided; this procedure will estimate them using finite differences. The computation of finite difference approximations usually increases the total run-time, since a call to `con_fun` is needed for each variable for which partial derivatives are estimated. For example, if the Jacobian has the form

$$\begin{pmatrix} * & * & * & * & * \\ * & ? & ? & * & * \\ * & * & ? & * & * \\ * & * & * & * & * \end{pmatrix}$$

where '$*$' indicates an element provided by the user and '?' indicates an element to be estimated, this procedure will call `con_fun` twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1, 4 and 5 are known, they require no calls to `con_fun`.) Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* [10] for a discussion of limiting accuracy). At times, central differences are used rather than forward differences, in which case twice as many calls to `con_fun` are needed. (The switch to central differences is determined by considerations of accuracy and is not under user control.)

`con_deriv` = `.true.` should be used whenever possible, since this procedure is more reliable (and will usually be more efficient) when all derivatives are exact.

Constraints: `con_deriv` must not be present unless `con_fun` and `num_nlin_con` are present.

Default: `con_deriv` = `.true.`.

con_fun — subroutine, optional

The procedure `con_fun`, supplied by the user, must calculate the vector $c(x)$ of nonlinear constraint functions and (optionally) its Jacobian ($= \partial c / \partial x$) at a specified point x.

Its specification is:

```
subroutine con_fun(first_call, x, finish, needc, con_f, con_jac)

logical, intent(in) :: first_call
```

Input: `first_call` will be `.true.` when this procedure calls `con_fun` for the first time, and `.false.` for all subsequent calls. It allows you to save computation time if certain data must be read or calculated only once. See also the description of `con_jac`.

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

Shape: `x` has shape (n).

Input: the point x at which the constraint functions and (optionally) elements of the constraint Jacobian are to be evaluated.

`logical, intent(inout) :: finish`

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

Output: if you wish to terminate the call to this procedure, you should set `finish` to `.true.`, and then this procedure will terminate with `error%code` = 201.

`integer, intent(in) :: needc(:)`

Shape: `needc` has shape (n_N).

Input: specifies the indices of the elements of `con_f` and (optionally) `con_jac` that must be evaluated. If `needc(`i`)` > 0, then the ith element of `con_f`, and (optionally) elements of the ith row of `con_jac`, must be evaluated at x, for $i = 1, 2, \ldots, n_N$.

`real(kind=`*wp*`), intent(inout) :: con_f(:)`

Shape: `con_f` has shape (n_N).

Input: the zero vector.

Output: if `needc(`i`)` > 0, `con_f(`i`)` must contain the value of the ith nonlinear constraint at the point x, for $i = 1, 2, \ldots, n_N$. Otherwise, `con_f(`i`)` need not be set.

`real(kind=`*wp*`), intent(inout), optional :: con_jac(:,:)`

Shape: `con_jac` has shape (n_N, n).

Input: if `con_jac` is present, its elements must remain unchanged except as specified below.

Output: if `con_jac` is present, then for each i such that `needc(`i`)` > 0:

if `con_deriv` = `.true.`(the default), the ith row of `con_jac` must contain *all* the elements of the vector ∇c_i given by

$$\nabla c_i = \left(\frac{\partial c_i}{\partial x_1}, \frac{\partial c_i}{\partial x_2}, \ldots, \frac{\partial c_i}{\partial x_n} \right)^T,$$

where $\partial c_i / \partial x_j$ is the partial derivative of the ith constraint with respect to the jth variable evaluated at the point x, for $i = 1, 2, \ldots, n_N$ and $j = 1, 2, \ldots, n$. Constant elements need be loaded into `con_jac` only during the first call to `con_fun` (when `first_call` = `.true.`). This facility is useful when many Jacobian elements are identically zero, in which case `con_jac` may be initialized to zero during the first call to `con_fun`. Note that although a constant non-zero element `con_jac(`i,j`)` only needs to be set on the first call to `con_fun`, the corresponding i in the definition of `con_f(`i`)` *must* be re-evaluated each time that `con_fun` is called.

If `con_deriv` = `.false.`, any *available* partial derivatives of $c_i(x)$ must be assigned to the corresponding elements in the ith row of `con_jac`; the remaining elements *must remain unchanged*. Just before `con_fun` is called, each element of `con_jac` is set to a special value. On return from this procedure, any element that retains the value is estimated by finite differences, at non-trivial expense. If you do not supply a value for `control%diff_int` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*), an interval for each element of x is computed automatically at the start of the optimization. The automatic procedure can usually identify constant elements of `con_jac`, which are then computed once only by finite differences.

If `needc(`i`)`≤ 0, the ith row of `con_jac` need not be set.

Note: if there are any nonlinear constraints, then the first call to `con_fun` will precede the first call to `obj_fun` (see Section 3.1). `con_fun` should be thoroughly tested before being supplied to this procedure. The components `cheap_test`, `con_verify` and `major_iter_lim` of the optional argument `control` can be used to assist this process (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

Constraints: `con_fun` must be present if `num_nlin_con` is present and greater than zero.

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

Output: `con_f`(i) contains the value of the ith nonlinear constraint function c_i at the final iterate, for $i = 1, 2, \ldots, n_N$.

Constraints: `con_f` must not be present unless `con_fun` and `num_nlin_con` are present.

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

Output: the Jacobian matrix of the nonlinear constraint functions at the final iterate (or its finite difference approximation), i.e., `con_jac`(i, j) contains the value of the partial derivative $\partial c_i / \partial x_j$ at the final point given in `x`, for $i = 1, 2, \ldots, n_N$ and $j = 1, 2, \ldots, n$.

Constraints: `con_jac` must not be present unless `con_fun` and `num_nlin_con` are present.

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

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

Input: the lower and upper bounds on all the nonlinear constraints. To specify a non-existent lower bound (i.e., $l_j = -\infty$), set `nlin_lower`(j) $\leq -$`control%inf_bound`; to specify a non-existent upper bound (i.e., $u_j = +\infty$), set `nlin_upper`(j) $\geq +$`control%inf_bound` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

Constraints:

`nlin_lower` and `nlin_upper` must not be present unless `con_fun` and `num_nlin_con` are present;

`nlin_lower`(j) $\leq$ `nlin_upper`(j) for $j = 1, 2, \ldots, n_N$;

$|\beta| <$ `control%inf_bound` when `nlin_lower`(j) = `nlin_upper`(j) = β.

Default: `nlin_lower` = $-$`control%inf_bound`; `nlin_upper` = $+$`control%inf_bound`.

cold_start — logical, intent(in), optional

Input: controls the specification of the initial working set in both the procedure for finding a feasible point for the linear constraints and bounds, and in the first QP subproblem thereafter.

With a *cold start* (i.e., `cold_start = .true.`), this procedure chooses the first working set based on the values of the variables and constraints at the initial point. Broadly speaking, the initial working set will include equality constraints and bounds or inequality constraints that violate or 'nearly' satisfy their bounds (to within the crash tolerance `control%crash_tol`; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

With a *warm start* (i.e., `cold_start = .false.`), the arrays `x_state`, `lin_state` (if $n_L > 0$), `nlin_state` and `nlin_lambda` (if $n_N > 0$) together with the array `r`, must be supplied and initialized. The arrays `x_state` and `lin_state` determine the initial working set of the procedure to find a feasible point with respect to the bounds and linear constraints, whereas the array `nlin_state` determines the initial working set of the first QP subproblem after such a feasible point has been found. This procedure will override the contents of these arrays if necessary, so that a poor choice of the working set will not cause a fatal error. A warm start will be advantageous if a good estimate of the initial working set is available, for example when this procedure is called repeatedly to solve related problems.

Default: `cold_start = .true.`.

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

Input: if cold_start = .true. (the default), x_state need not be initialized.

If cold_start = .false., x_state specifies the status of the upper and lower bounds on the variables which together with the array lin_state define the initial working set for the procedure that finds a feasible point for the linear constraints and bounds. Possible values for x_state(j) are as follows:

x_state(j)	Meaning
0	The corresponding constraint should *not* be in the initial QP working set.
1	This constraint should be in the working set at its lower bound.
2	This constraint should be in the working set at its upper bound.
3	This constraint should be in the initial working set. This value must not be specified unless the corresponding lower and upper bounds are equal.

Any other values will be modified by this procedure. Note that x_state already contains valid values if it was present in a previous call with the same value of n. (See also the description of cold_start.) This procedure also adjusts (if necessary) the values supplied in x to be consistent with x_state.

Output: the status of the constraints in the QP working set at the point returned in x. The significance of each possible value of x_state(j) is as follows:

x_state(j)	Meaning
−2	This constraint violates its lower bound by more than the linear feasibility tolerance control%lin_feas_tol (see the type definition for nag_con_nlin_lsq_cntrl_*wp*). This value can only occur when no feasible point can be found for a QP subproblem.
−1	This constraint violates its upper bound by more than the linear feasibility tolerance. This value can only occur when no feasible point can be found for a QP subproblem.
0	This constraint is satisfied to within the linear feasibility tolerance, but is not in the QP working set.
1	This constraint is included in the QP working set at its lower bound.
2	This constraint is included in the QP working set at its upper bound.
3	This constraint is included in the QP working set as an equality. This can only occur when the corresponding upper and lower bounds are equal.

Constraints: if cold_start = .false., x_state must be present.

lin_state(n_L) — integer, intent(inout), optional

Input: if cold_start = .true. (the default), lin_state need not be initialized.

If cold_start = .false., lin_state specifies the status of the upper and lower bounds on the linear constraints which together with the array x_state define the initial working set for the procedure that finds a feasible point for the linear constraints and bounds. Possible values for lin_state(j) are as follows:

lin_state(j)	Meaning
0	The corresponding constraint should *not* be in the initial QP working set.
1	This constraint should be in the working set at its lower bound.
2	This constraint should be in the working set at its upper bound.
3	This constraint should be in the initial working set. This value must not be specified unless the corresponding lower and upper bounds are equal.

Any other values will be modified by this procedure. Note that lin_state already contains valid values if it was present in a previous call with the same value of n_L. (See also the description of cold_start.)

Output: the status of the constraints in the QP working set at the point returned in x. The significance of each possible value of lin_state(j) is as follows:

lin_state(j)	Meaning
−2	This constraint violates its lower bound by more than the linear feasibility tolerance control%lin_feas_tol (see the type definition for nag_con_nlin_lsq_cntrl_*wp*). This value can only occur when no feasible point can be found for a QP subproblem.
−1	This constraint violates its upper bound by more than the linear feasibility tolerance. This value can only occur when no feasible point can be found for a QP subproblem.
0	This constraint is satisfied to within the linear feasibility tolerance, but is not in the QP working set.
1	This constraint is included in the QP working set at its lower bound.
2	This constraint is included in the QP working set at its upper bound.
3	This constraint is included in the QP working set as an equality. This can only occur when the corresponding upper and lower bounds are equal.

Constraints: lin_state must not be present unless a is present. If cold_start = .false., lin_state must be present if $n_L > 0$.

nlin_state(n_N) — integer, intent(inout), optional

Input: if cold_start = .true. (the default), nlin_state need not be initialized.

If cold_start = .false., nlin_state specifies the status of the upper and lower bounds on the nonlinear constraints, which together with the active set at the conclusion of the procedure to find a feasible point for the linear constraints and bounds, define the initial working set for the first QP subproblem. Possible values for nlin_state(j) are as follows:

nlin_state(j)	Meaning
0	The corresponding constraint should *not* be in the initial QP working set.
1	This constraint should be in the working set at its lower bound.
2	This constraint should be in the working set at its upper bound.
3	This constraint should be in the initial working set. This value must not be specified unless the corresponding lower and upper bounds are equal.

Any other values will be modified by this procedure. Note that nlin_state already contains valid values if it was present in a previous call with the same value of n_N. (See also the description of cold_start.)

Output: the status of the constraints in the QP working set at the point returned in x. The significance of each possible value of nlin_state(j) is as follows:

nlin_state(j)	Meaning
−2	This constraint violates its lower bound by more than the nonlinear feasibility tolerance control%nlin_feas_tol (see the type definition for nag_con_nlin_lsq_cntrl_*wp*). This value can only occur when no feasible point can be found for a QP subproblem.
−1	This constraint violates its upper bound by more than the nonlinear feasibility tolerance. This value can only occur when no feasible point can be found for a QP subproblem.
0	This constraint is satisfied to within the nonlinear feasibility tolerance, but is not in the QP working set.
1	This constraint is included in the QP working set at its lower bound.
2	This constraint is included in the QP working set at its upper bound.
3	This constraint is included in the QP working set as an equality. This can only occur when the corresponding upper and lower bounds are equal.

Constraints: nlin_state must not be present unless con_fun and num_nlin_con are present. If cold_start = .false., nlin_state must be present if $n_N > 0$.

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

Output: the values of the QP multipliers for the bound constraints from the last QP subproblem. x_lambda(j) should be non-negative if x_state(j) = 1 and non-positive if x_state(j) = 2.

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

Output: the values of the QP multipliers for the linear constraints from the last QP subproblem. lin_lambda(j) should be non-negative if lin_state(j) = 1 and non-positive if lin_state(j) = 2.

Constraints: lin_lambda must not be present unless a is present.

nlin_lambda(n_N) — real(kind=*wp*), intent(inout), optional

Input: if cold_start = .true. (the default), nlin_lambda need not be initialized. If cold_start = .false., nlin_lambda must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by the array nlin_state.

Note that:

if the jth constraint is defined as 'inactive' (nlin_state(j) = 0), nlin_lambda(j) should be zero;

if the jth constraint is an inequality active at its lower bound (nlin_state(j) = 1), nlin_lambda(j) should be non-negative;

if the jth constraint is an inequality active at its upper bound (nlin_state(j) = 2), nlin_lambda(j) should be non-positive.

If necessary, this procedure will modify nlin_lambda to match these rules.

Output: the values of the QP multipliers for the nonlinear constraints from the last QP subproblem. nlin_lambda(j) should be non-negative if nlin_state(j) = 1 and non-positive if nlin_state(j) = 2.

Constraints: nlin_lambda must not be present unless con_fun and num_nlin_con are present. If cold_start = .false., nlin_lambda must be present if $n_N > 0$.

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

Input: if cold_start = .true. (the default), r need not be initialized.

If cold_start = .false., r must contain the upper triangular Cholesky factor R of the initial approximation of the Hessian of the Lagrangian function, with the variables in the natural order. Elements in the strictly lower triangular part of r are assumed to be zero and need not be assigned.

Note that r already contains satisfactory information if it was present in a previous call to this procedure with control%hessian = .true. (the default; see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

Output: if control%hessian = .true., r contains the upper triangular Cholesky factor R of H, the approximate (untransformed) Hessian of the Lagrangian, with the variables in the natural order.

If control%hessian = .false., r contains the upper triangular Cholesky factor R of $Q^T\tilde{H}Q$, an estimate of the transformed and re-ordered Hessian of the Lagrangian at x (see (10) in Section 1 of the Mathematical Background section of this module document).

Constraints: if cold_start = .false., r must be present.

major_iter — integer, intent(out), optional

Output: the number of major iterations performed.

minor_iter — integer, intent(out), optional

Output: the number of minor iterations performed.

control — type(nag_con_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_con_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.
305	Invalid absence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code **Description**

201 User requested termination.

This exit occurs if you have set `finish` to `.true.` in `obj_fun` or `con_fun`.

202 No feasible point was found for the linear constraints and bounds, which means that either no feasible point exists for the given value of `control%lin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*), or no feasible point could be found in the number of iterations specified by `control%minor_iter_lim` (default value = $\max(50,3(n+n_{\rm L}+n_{\rm N}))$).

You should check that there are no constraint redundancies. If the data for the constraints are accurate only to an absolute precision σ, you should ensure that the value of `control%lin_feas_tol` is *greater* than σ. For example, if all the elements of $A_{\rm L}$ are of order unity and are accurate only to three decimal places, then `control%lin_feas_tol` should be at least 10^{-3}.

203 No feasible point could be found for the nonlinear constraints. The problem may have no feasible solution. This means that there has been a sequence of QP subproblems for which no feasible point could be found (indicated by `I` at the end of each line of intermediate printout produced by the major iterations; see Section 7.1).

This behaviour will occur if there is no feasible point for the nonlinear constraints. (However, there is no general test that can determine whether a feasible point exists for a set of nonlinear constraints.) If the infeasible subproblems occur from the very first major iteration, it is highly likely that no feasible point exists. If infeasibilities occur when earlier subproblems have been feasible, small constraint inconsistencies may be present. You should check the validity of constraints with negative values of `nlin_state` (see Section 3.2). If you are convinced that a feasible point *does* exist, this procedure should be restarted at a different starting point.

204 x does not satisfy the first-order Kuhn–Tucker conditions (see Section 1 of the Mathematical Background section of this module document), and no improved point for the merit function (see Section 7.1) could be found during the final linesearch.

This sometimes occurs because an overly stringent accuracy has been requested, i.e., the value of `control%optim_tol` is too small (default value = $(\texttt{EPSILON(1.0\_}wp\texttt{)})^{0.72}$; see the type definition of `nag_con_nlin_lsq_cntrl_`*wp*). In this case you should apply the following tests to determine whether or not the final solution is acceptable (see Gill *et al.* [10], for a discussion of the attainable accuracy):

(a) the final value of `Norm Gz` (see Section 7.1) is significantly less than that at the starting point;

(b) during the final major iterations, the values of `Step` and `Mnr` (see Section 7.1) are both one;

(c) the last few values of both `Norm Gz` and `Violtn` (see Section 7.1) become small at a fast linear rate; and

(d) `Cond Hz` (see Section 7.1) is small.

If all these conditions hold, x is almost certainly a local minimum of (1).

If many iterations have occurred in which essentially no progress has been made and this procedure has failed completely to move from the initial point, then procedures `obj_fun` and/or `con_fun` may be incorrect. You should refer to the description of `error%code` = 205 and check the Jacobians using `control%cheap_test = .false.` (default value = `.true.`; see the type definition for **nag_con_nlin_lsq_cntrl_*wp***). Unfortunately, there may be small errors in the objective and constraint Jacobians that cannot be detected by the verification. Finite difference approximations to first derivatives can be catastrophically affected even by small inaccuracies. An indication of this situation is a dramatic alteration in the iterates if the finite difference interval is altered. One might also suspect this type of error if a switch is made to central differences even when `Norm Gz` and `Violtn` (see Section 7.1) are large.

Another possibility is that the search direction has become inaccurate because of ill conditioning in the Hessian approximation or the matrix of constraints in the working set; either form of ill conditioning tends to be reflected in large values of `Mnr` (the number of iterations required to solve each QP subproblem; see Section 7.1).

If the condition estimate of the projected Hessian (`Cond Hz`; see Section 7.1) is extremely large, it may be worthwhile rerunning this procedure from the final point using `cold_start = .false.` (see Section 3.2). In this situation `x_state`, `lin_state` (if $n_L > 0$), `nlin_state` and `nlin_lambda` (if $n_N > 0$; see Section 3.2) should be left unaltered, and R should be reset to the identity matrix.

If the condition estimate of the matrix of constraints in the working set (`Cond T`; see Section 7.1) is extremely large, it may be worthwhile rerunning this procedure with relaxed values of `control%lin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`) and/or `control%nlin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))` or `(EPSILON(1.0_`*wp*`))`$^{0.33}$; see the type definition for **nag_con_nlin_lsq_cntrl_*wp***). (Constraint dependencies are often indicated by wide variations in size in the diagonal elements of the matrix T, whose diagonals will be printed if the printing parameter `control%major_print_level` ≥ 30 (default value = 10; see Section 7.1).)

205 The user-provided derivatives of the subfunctions and/or constraints appear to be incorrect.

Large errors were found in the derivatives of the subfunctions and/or constraints. This exit occurs if the verification process indicated that at least one Jacobian element had no correct figures. You should refer to the printed output to determine which elements are suspected to be in error.

As a first step, you should check that the code for computing the nonlinear functions and constraints 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 functions and constraints. 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.

Jacobian checking will be ineffective if the subfunctions (see $f_i(x)$ in (1)) use information computed by the constraints, since they are not necessarily computed prior to each evaluation.

Errors in programming the subfunctions or constraints may be quite subtle in that the values are 'almost' correct. For example, the nonlinear function value may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which it depends.

Warnings (error%level = 1):

error%code	Description
101	The final iterate x satisfies the first-order Kuhn–Tucker conditions (see Section 1 of the Mathematical Background section of this module document) to the accuracy requested, but the sequence of iterates has not yet converged. This procedure was terminated because no further improvement could be made in the merit function (see Section 7.1). This exit may occur in several circumstances. The most common situation is that you have asked for a solution with accuracy that is not attainable with the given precision of the problem (as specified by `control%fun_prec` (default value = $(\texttt{EPSILON(1.0\_}wp\texttt{)})^{0.9}$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*)). This condition will also occur if, by chance, an iterate is an 'exact' Kuhn–Tucker point, but the change in the variables was significant at the previous iteration. (This situation often happens when minimizing very simple functions, such as quadratics.) If conditions (a)–(d) described under `error%code` = 204 are satisfied, x is likely to be a solution of (1) even if `error%code` = 101.
102	The limiting number of iterations was reached before normal termination occurred. If the algorithm appears to be making satisfactory progress, then the value of `control%major_iter_lim` (default value = $\max(50, 3 \times (n + n_{\rm L}) + 10 \times n_{\rm N})$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*) may be too small. If so, either increase its value and rerun this procedure or, alternatively, rerun this procedure using `cold_start` = `.false.` (see Section 3.2). If the algorithm seems to be making little or no progress however, then you should check for incorrect Jacobians or ill conditioning (as described under `error%code` = 204). Note that ill conditioning in the working set is sometimes resolved automatically by the algorithm, in which case performing additional iterations may be helpful. However, ill conditioning in the Hessian approximation tends to persist once it has begun, so that allowing additional iterations without altering R is usually inadvisable. If the quasi-Newton update of the Hessian approximation was reset during the latter major iterations (i.e., an `R` occurs at the end of each line of intermediate printout; see Section 7.1), it may be worthwhile rerunning this procedure using `cold_start` = `.false.` (see Section 3.2).

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 Accuracy

If error%code $= 0$ on exit, then the vector returned in the array x is an estimate of the solution to an accuracy of approximately control%optim_tol (default value = (EPSILON(1.0_*wp*))$^{0.72}$; see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

6.2 Termination Criteria

This procedure returns with error%code $= 0$ if the iterates have converged to a point x^* that satisfies the first-order Kuhn–Tucker conditions (see Section 1 of the Mathematical Background section of this module document) to the accuracy requested by control%optim_tol (default value = (EPSILON(1.0_*wp*))$^{0.72}$; see the type definition for nag_con_nlin_lsq_cntrl_*wp*), i.e., the projected gradient and active constraint residuals are negligible at x^*.

6.3 Overflow

If the printed output before the overflow error contains a warning about serious ill conditioning in the working set when adding the jth constraint, it may be possible to avoid the difficulty by increasing the magnitude of control%nlin_feas_tol (default value = SQRT(EPSILON(1.0_*wp*)) or (EPSILON(1.0_*wp*))$^{0.33}$; see the type definition for nag_con_nlin_lsq_cntrl_*wp*) and/or control%lin_feas_tol (default value = SQRT(EPSILON(1.0_*wp*))) and rerunning the program. If the message recurs even after this change, the offending linearly dependent constraint (with index 'j') *must* be removed from the problem.

If overflow occurs in one of the user-supplied procedures (e.g., if the subfunctions involve exponentials or singularities), it may help to specify tighter bounds for some of the variables (i.e., reduce the gap between the appropriate l_j and u_j).

7 Description of Printed Output

7.1 Major Iteration Printout

This section describes the intermediate and final printout produced by the major iterations of this procedure (see Section 1 of the Mathematical Background section of this module document). The level of printed output can be controlled via the components list and major_print_level of the optional argument control. 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 are produced only if control%major_print_level ≥ 10 (the default).

When control%major_print_level ≥ 5 and control%lt80_char = .true. (the default), the following line of output (< 80 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

Maj is the major iteration count.

Mnr is the number of minor iterations required by the feasibility and optimality phases of the QP subproblem. Generally, Mnr will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see the Mathematical Background section of this module document).
Note that Mnr may be greater than control%minor_iter_lim (default value = $\max(50, 3\times(n+n_{\mathrm{L}}+n_{\mathrm{N}}))$; see the type definition for nag_con_nlin_lsq_cntrl_*wp*) if some iterations are required for the feasibility phase.

Step is the step taken along the computed search direction. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.

`Merit Function` is the value of the augmented Lagrangian merit function (see Section 3 of the Mathematical Background section of this module document) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters As the solution is approached, `Merit Function` will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by `I` at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty parameters. During a sequence of major iterations with infeasible subproblems, the sequence of `Merit Function` values will decrease monotonically until either a feasible subproblem is obtained or this procedure terminates with `error%code` = 203 (no feasible point could be found for the nonlinear constraints).

If no nonlinear constraints are present (i.e., $n_{\rm N} = 0$), this entry contains `Objective`, the value of the objective function $F(x)$. The objective function will decrease monotonically to its optimal value when there are no nonlinear constraints.

`Norm Gz` is $\|Z^T g_{\rm FR}\|$, the Euclidean norm of the projected gradient (see Section 2 of the Mathematical Background section of this module document). `Norm Gz` will be approximately zero in the neighbourhood of a solution.

`Violtn` is the Euclidean norm of the residuals of nonlinear constraints that are violated or in the predicted active set (not printed if $n_{\rm N} = 0$). `Violtn` will be approximately zero in the neighbourhood of a solution.

`Cond Hz` is a lower bound on the condition number of the projected Hessian approximation H_Z ($H_Z = Z^T H_{\rm FR} Z = R_Z^T R_Z$; see (10) in Section 1 and (15) in Section 2 of the Mathematical Background section of this module document). The larger this number, the more difficult the problem.

`M` is printed if the quasi-Newton update has been modified to ensure that the Hessian approximation is positive definite (see Section 4 of the Mathematical Background section of this module document).

`I` is printed if the QP subproblem has no feasible point.

`C` is printed if central differences have been used to compute the unspecified objective and constraint Jacobians. If the value of `Step` is zero, the switch to central differences was made because no lower point could be found in the linesearch. (In this case, the QP subproblem is re-solved with the central difference gradient and Jacobian.)

If the value of `Step` is non-zero, central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 1 of the Mathematical Background section of this module document).

`L` is printed if the linesearch has produced a relative change in x greater than the value defined by `control%step_limit` (default value = 2.0; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

If this output occurs frequently during later iterations of the run, `control%step_limit` should be set to a larger value.

`R` is printed if the approximate Hessian has been refactorized. If the diagonal condition estimator of R indicates that the approximate Hessian is badly conditioned, the approximate Hessian is refactorized using column interchanges. If necessary, R is modified so that its diagonal condition estimator is bounded.

When `control%major_print_level` ≥ 5 and `control%lt80_char` = `.false.`, the following line of output (up to 132 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Maj` (as above)

`Mnr` (as above)

`Step` (as above)

`Nfun` is the cumulative number of evaluations of the objective function needed for the linesearch. Evaluations needed for the estimation of the Jacobians by finite differences are not included. `Nfun` is printed as a guide to the amount of work required for the linesearch.

`Merit Function`	(as above)
`Norm Gz`	(as above)
`Violtn`	(as above)
`Nz`	is the number of columns of Z (see Section 2 of the Mathematical Background section of this module document). The value of `Nz` is the number of variables minus the number of constraints in the predicted active set; i.e., `Nz` $= n - ($ `Bnd` + `Lin` + `Nln`$)$.
`Bnd`	is the number of simple bound constraints in the predicted active set.
`Lin`	is the number of general linear constraints in the predicted active set.
`Nln`	is the number of nonlinear constraints in the predicted active set (not printed if $n_N = 0$).
`Penalty`	is the Euclidean norm of the vector of penalty parameters used in the augmented Lagrangian merit function (not printed if $n_N = 0$).
`Cond H`	is a lower bound on the condition number of the Hessian approximation H.
`Cond Hz`	(as above)
`Cond T`	is a lower bound on the condition number of the matrix of predicted active constraints.
`Conv`	is a three-letter indication of the status of the three convergence tests defined in the description of `control%optim_tol` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). Each letter is `T` if the test is satisfied, and `F` otherwise. The three tests indicate whether: (a) the sequence of iterates has converged; (b) the projected gradient (`Norm Gz`) is sufficiently small; and (c) the norm of the residuals of constraints in the predicted active set (`Violtn`) is small enough. If any of these indicators is `F` on termination with `error%level` = 0, you should check the solution carefully.
`M`	(as above)
`I`	(as above)
`C`	(as above)
`L`	(as above)
`R`	(as above)

The final printout includes a listing of the status of every variable and constraint.

The following describes the printout for each variable. A full stop (.) is printed for any numerical value that is zero.

`Varbl` gives the name (`V`) and index j, for $j = 1, 2, \ldots, n$ of the variable.

`State` gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than `control%lin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*), `State` will be `++` or `--` respectively. (The latter situation can occur only when there is no feasible point for the bounds and linear constraints.)
A key is sometimes printed before `State` to give additional information about the state of a variable.

`A` *Alternative optimum possible.* The variable is active at one of its bounds, but its Lagrange multiplier is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change to the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them would encounter a bound immediately. In either case the values of the Lagrange multipliers might also change.

D	*Degenerate.* The variable is free, but it is equal to (or very close to) one of its bounds.
I	*Infeasible.* The variable is currently violating one of its bounds by more than control%lin_feas_tol.

Value — is the value of the variable at the final iterate.

Lower Bound — is the lower bound specified for the variable. None indicates that x_lower(j) $\leq$ $-$control%inf_bound (default value $= 10^{20}$; see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

Upper Bound — is the upper bound specified for the variable. None indicates that x_upper(j) $\geq$ control%inf_bound.

Lagr Mult — is the Lagrange multiplier for the associated bound. This will be zero if State is FR unless x_lower(j) $\leq -$control%inf_bound and x_upper(j) $\geq$ control%inf_bound, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if State is LL, and non-positive if State is UL.

Slack — is the difference between the variable Value and the nearer of its (finite) bounds x_lower(j) and x_upper(j). A blank entry indicates that the associated variable is not bounded (i.e., x_lower(j) $\leq -$control%inf_bound and x_upper(j) $\geq$ control%inf_bound).

The meaning of the printout for linear and nonlinear constraints is the same as that given above for variables, with 'variable' replaced by 'constraint', x_lower and x_upper are replaced by either lin_lower and lin_upper, or by nlin_lower and nlin_upper, respectively, control%lin_feas_tol is replaced by control%nlin_feas_tol for the nonlinear constraints and with the following changes in the heading:

L Con — gives the name (L) and index j, for $j = 1, 2, \ldots, n_{\mathrm{L}}$ of the linear constraint.

N Con — gives the name (N) and index j, for $j = 1, 2, \ldots, n_{\mathrm{N}}$ of the nonlinear constraint.

Note that movement off a constraint (as opposed to a variable moving away from its bound) can be interpreted as allowing the entry in the Slack column to become positive.

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

7.2 Minor Iteration Printout

This section describes the intermediate and final printout produced by the minor iterations of this procedure, which involves solving a QP subproblem of the form

$$\underset{p}{\text{minimize}}\ g^T p + \tfrac{1}{2} p^T H p \ \text{ subject to } \ \bar{l} \leq \left\{ \begin{array}{c} p \\ A_{\mathrm{L}} p \\ A_{\mathrm{N}} p \end{array} \right\} \leq \bar{u} \qquad (2)$$

at every major iteration. (For more details see Section 1 of the Mathematical Background section of this module document.) The level of printed output can be controlled via the component minor_print_level of the optional argument control. Note that the intermediate printout and the final printout are produced only if control%minor_print_level ≥ 10 (default value $= 0$, which produces no output).

When control%minor_print_level ≥ 5 and control%lt80_char = .true., the following line of output (< 80 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration of the QP subproblem.

Itn — is the iteration count.

Step — is the step taken along the computed search direction. If a constraint is added during the current iteration, Step will be the step to the nearest constraint. During the optimality phase, the step can be greater than one only if the factor R_Z is singular (see Section 2 of the Mathematical Background section of this module document).

Ninf — is the number of violated constraints (infeasibilities). This will be zero during the optimality phase.

`Sinf/Objective` is the value of the current objective function. If x is not feasible, `Sinf` gives a weighted sum of the magnitudes of the constraint violations. If x is feasible, `Objective` is the value of the QP objective function in (2). The output line for the final iteration of the feasibility phase (i.e., the first iteration for which `Ninf` is zero) will give the value of the true objective at the first feasible point.
During the optimality phase, the value of the objective function will be non-increasing. During the feasibility phase, the number of constraint infeasibilities will not increase until either a feasible point is found, or the optimality of the multipliers implies that no feasible point exists. Once optimal multipliers are obtained, the number of infeasibilities can increase, but the sum of infeasibilities will either remain constant or be reduced until the minimum sum of infeasibilities is found.

`Norm Gz` is $||Z^T q_{\text{FR}}||$, the Euclidean norm of the reduced gradient of the QP objective function in (2) with respect to Z (see Section 2 of the Mathematical Background section of this module document). During the optimality phase, this norm will be approximately zero after a unit step.

When `control%minor_print_level` ≥ 5 and `control%lt80_char` $=$ `.false.`, the following line of output (up to 120 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration of the QP subproblem. The following convention is used for numbering the constraints: indices 1 through n refer to the bounds on the variables, and indices $n+1$ through $n+n_{\text{L}}$ or $n+n_{\text{L}}+n_{\text{N}}$ refer to the general constraints (if any). When the status of a constraint changes, the index of the constraint is printed, along with the designation `L` (lower bound), `U` (upper bound), `E` (equality), `F` (temporarily fixed variable) or `A` (artificial constraint).

`Itn` (as above)

`Jdel` is the index of the constraint deleted from the QP working set. If `Jdel` is zero, no constraint was deleted.

`Jadd` is the index of the constraint added to the QP working set. If `Jadd` is zero, no constraint was added.

`Step` (as above)

`Ninf` (as above)

`Sinf/Objective` (as above)

`Bnd` is the number of simple bound constraints in the current QP working set.

`Lin` is the number of general linear constraints in the current QP working set.

`Art` is the number of artificial constraints in the QP working set.

`Zr` is the dimension of the sub-space in which the QP objective function in (2) (Section 7.2) is currently being minimized. The value of `Zr` is the number of variables minus the number of constraints in the working set; i.e., `Zr` $= n -$ (`Bnd`+`Lin`+`Art`).
The value of n_Z, the number of columns of Z (see Section 1 of the Mathematical Background section of this module document) can be calculated as $n_Z = n -$ (`Bnd`+`Lin`). A zero value of n_Z implies that x lies at a vertex of the feasible region.

`Norm Gz` (as above)

`Norm Gf` is $||q_{\text{FR}}||$, the Euclidean norm of the gradient of the QP objective function in (2) with respect to the free variables, i.e., variables not currently held at a bound (see Section 2 of the Mathematical Background section of this module document).

`Cond T` is a lower bound on the condition number of the QP working set.

`Cond Rz` is a lower bound on the condition number of the triangular factor R_1 (the first `Zr` rows and columns of the factor R_Z; see Section 2 of the Mathematical Background section of this module document). If the estimated rank of the matrix H in (2) is zero, `Cond Rz` is not printed.

The final printout includes a listing of the status of every variable and constraint.

The following describes the printout for each variable. A full stop (.) is printed for any numerical value that is zero.

`Varbl`	gives the name (`V`) and index j, for $j = 1, 2, \ldots, n$ of the variable.
`State`	gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than `control%lin_feas_tol`, `State` will be `++` or `--` respectively. A key is sometimes printed before `State` to give additional information about the state of a variable.

- `A` *Alternative optimum possible.* The variable is active at one of its bounds, but its Lagrange multiplier is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change to the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them would encounter a bound immediately. In either case the values of the Lagrange multipliers might also change.
- `D` *Degenerate.* The variable is free, but it is equal to (or very close to) one of its bounds.
- `I` *Infeasible.* The variable is currently violating one of its bounds by more than `control%lin_feas_tol`.

`Value`	is the value of the variable at the final iterate.
`Lower Bound`	is the lower bound specified for the variable. `None` indicates that $\bar{l}_j \leq -$`control%inf_bound` (default value $= 10^{20}$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).
`Upper Bound`	is the upper bound specified for the variable. `None` indicates that $\bar{u}_j \geq$ `control%inf_bound`.
`Lagr Mult`	is the Lagrange multiplier for the associated bound. This will be zero if `State` is `FR` unless $\bar{l}_j \leq -$`control%inf_bound` and $\bar{u}_j \geq$`control%inf_bound`, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is `LL`, and non-positive if `State` is `UL`.
`Slack`	is the difference between the variable `Value` and the nearer of its (finite) bounds $\bar{l}_j$ and $\bar{u}_j$. A blank entry indicates that the associated variable is not bounded (i.e., $\bar{l}_j \leq -$`control%inf_bound` and $\bar{u}_j \geq$ `control%inf_bound`).

The meaning of the printout for general constraints is the same as that given above for variables, with 'variable' replaced by 'constraint', $\bar{l}_j$ and $\bar{u}_j$ are replaced by $\bar{l}_{j+n}$ and $\bar{u}_{j+n}$ respectively, `control%lin_feas_tol` is replaced by `control%nlin_feas_tol` for the nonlinear constraints and with the following changes in the heading:

`L Con`	gives the name (`L`) and index j, for $j = 1, 2, \ldots, n_{\rm L} + n_{\rm N}$ of the constraint unless an initial feasible point (for the linear constraints and bounds) is being sought, in which case $j = 1, 2, \ldots, n_{\rm L}$.

Note that movement off a constraint (as opposed to a variable moving away from its bound) can be interpreted as allowing the entry in the `Slack` column to become positive.

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

Procedure: nag_con_nlin_lsq_sol_1

1 Description

nag_con_nlin_lsq_sol_1 is designed to solve a nonlinear least-squares problem — minimizing a smooth (nonlinear) sum of squares function subject to constraints on the variables.

The problem is assumed to be stated in the following form:

$$\underset{x \in R^n}{\text{minimize}} \;\; F(x) = \tfrac{1}{2}\sum_{i=1}^{m}(f_i(x) - y_i)^2 \;\; \text{subject to} \;\; l \leq \left\{ \begin{array}{c} x \\ A_{\rm L}x \\ c(x) \end{array} \right\} \leq u, \tag{3}$$

where $F(x)$ is a nonlinear objective function, the $f_i(x)$ are subfunctions, the y_i are constant and the constraints are grouped as follows:

n *simple bounds* on the variables x;

$n_{\rm L}$ *linear constraints*, defined by the $n_{\rm L}$ by n constant matrix $A_{\rm L}$;

$n_{\rm N}$ *nonlinear constraints*, defined by the vector $c(x)$ of constraint functions.

(The functions $f_i(x) - y_i$ are often referred to as 'residuals'.) The subfunctions and the constraint functions are assumed to be smooth, i.e., at least twice-continuously differentiable. (The method used by this procedure will usually solve (3) if there are only isolated discontinuities away from the solution.)

The simple bounds on the variables, the linear constraints and the nonlinear constraints are distinguished from one another for reasons of computational efficiency (although the simple bounds could have been included in the definition of the linear constraints, and the linear constraints in the definition of the nonlinear constraints). There may be no linear constraints, in which case the matrix $A_{\rm L}$ is empty ($n_{\rm L} = 0$), or no nonlinear constraints, in which case the vector $c(x)$ is empty ($n_{\rm N} = 0$).

Upper bounds and/or lower bounds can be specified separately for the variables and constraints. An *equality* constraint can be specified by setting $l_i = u_i$. If certain bounds are not present, the associated elements of l and u can be set to special values that will be treated as $-\infty$ or $+\infty$.

You must supply an initial estimate of the solution to (3), together with a procedure obj_fun that defines the subfunctions $f(x)$ (see Section 3.1), and (if $n_{\rm N} > 0$) a procedure con_fun which defines the nonlinear constraint functions $c(x)$ (see Section 3.2). On every call, these procedures must return values of $f(x)$ and $c(x)$, and as many partial derivatives as possible. For maximum reliability, you should provide all partial derivatives (see Chapter 8 of Gill *et al.* [10] for a detailed discussion). Any derivatives which are not provided are approximated by finite differences.

Several options are available for controlling the operation of this procedure, covering facilities such as:

printed output, at the end of each iteration and at the final solution;

verifying or estimating partial derivatives;

algorithmic parameters, such as tolerances and iteration limits.

These options are grouped together in the optional argument control, which is a structure of the derived type nag_con_nlin_lsq_cntrl_*wp*.

The method used by this procedure is described in detail in the Mathematical Background section of this module document.

2 Usage

```
USE nag_con_nlin_lsq
CALL nag_con_nlin_lsq_sol_1(obj_fun, x, obj_f, f  [, 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.

$m \geq 1$ — the number of subfunctions or 'residuals'
$n \geq 1$ — the number of variables
$n_{\mathrm{L}} \geq 0$ — the number of linear constraints
$n_{\mathrm{N}} \geq 0$ — the number of nonlinear constraints

3.1 Mandatory Arguments

obj_fun — subroutine

The procedure obj_fun, supplied by the user, must calculate the vector $f(x)$ of subfunctions and (optionally) its Jacobian ($= \partial f / \partial x$) at a specified point x.

Its specification is:

```
subroutine obj_fun(first_call, x, finish, f, f_jac, needf)
```

`logical, intent(in) :: first_call`

Input: first_call will be .true. when this procedure calls obj_fun for the first time, and .false. for all subsequent calls. It allows you to save computation time if certain data must be read or calculated only once. See also the description of f_jac.

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

Shape: x has shape (n).

Input: the point x at which the subfunctions and (optionally) elements of the objective Jacobian are to be evaluated.

`logical, intent(inout) :: finish`

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

Output: if you wish to terminate the call to this procedure, you should set finish to .true., and then this procedure will terminate with error%code = 201.

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

Shape: f has shape (m).

Output: if needf is not present, f(i) must contain the value of the ith subfunction f_i at the point x, for $i = 1, 2, \ldots, m$. Otherwise, the only element of f that needs to be set is the one for which the corresponding element of needf is > 0.

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

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

Input: if `f_jac` is present, its elements must remain unchanged except as specified below.

Output: if `f_jac` is present, then:

if `f_deriv` = `.true.`(the default; see Section 3.2), the ith row of `f_jac` must contain *all* the elements of the vector ∇f_i given by

$$\nabla f_i = \left(\frac{\partial f_i}{\partial x_1}, \frac{\partial f_i}{\partial x_2}, \ldots, \frac{\partial f_i}{\partial x_n} \right)^T,$$

where $\partial f_i / \partial x_j$ is the partial derivative of the ith subfunction with respect to the jth variable evaluated at the point x, for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. Constant elements need be loaded into `f_jac` only during the first call to `obj_fun` (when `first_call` = `.true.`). This facility is useful when many Jacobian elements are identically zero, in which case `f_jac` may be initialized to zero during the first call to `obj_fun`. Note that although a constant non-zero element `f_jac`(i, j) only needs to be set on the first call to `obj_fun`, the corresponding i in the definition of `f`(i) *must* be re-evaluated each time that `obj_fun` is called.

If `f_deriv` = `.false.`, any *available* partial derivatives of $f_i(x)$ must be assigned to the corresponding elements in the ith row of `f_jac`; the remaining elements *must remain unchanged*. Just before `obj_fun` is called, each element of `f_jac` is set to a special value. On return from this procedure, any element that retains the value is estimated by finite differences, at non-trivial expense. If you do not supply a value for `control%diff_int` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*), an interval for each element of x is computed automatically at the start of the optimization. The automatic procedure can usually identify constant elements of `f_jac`, which are then computed once only by finite differences.

`integer, intent(in), optional :: needf(:)`

Shape: `needf` has shape (m).

Input: if `needf` is present, it specifies which element of `f` should be evaluated at `x`, i.e., `f`(i) if `needf`$(i) > 0$. The remaining elements need not be set.

Note: `obj_fun` should be thoroughly tested before being supplied to this procedure. The components `cheap_test`, `obj_verify` and `major_iter_lim` of the optional argument `control` can be used to assist this process (see the type definition of `nag_con_nlin_lsq_cntrl_`*wp*).

x(n) — real(kind=*wp*), intent(inout)

Input: an initial estimate of the solution.

Output: the final estimate of the solution.

obj_f — real(kind=*wp*), intent(out)

Output: the value of the objective function at the final iterate.

f(m) — real(kind=*wp*), intent(out)

Output: `f`(i) contains the value of the ith subfunction f_i at the final iterate, 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.

f_deriv — logical, intent(in), optional

Input: specifies whether or not all elements of the objective Jacobian are provided by the user-supplied procedure `obj_fun`.

If `f_deriv` = `.true.` (the default), then all elements of the objective Jacobian must be provided by `obj_fun` via its argument `f_jac`.

If `f_deriv` = `.false.`, then it is assumed that some elements of the objective Jacobian are not provided; this procedure will estimate them using finite differences. The computation of finite difference approximations usually increases the total run-time, since a call to `obj_fun` is needed for each variable for which partial derivatives are estimated. For example, if the Jacobian has the form

$$\begin{pmatrix} * & * & * & * & * \\ * & ? & ? & * & * \\ * & * & ? & * & * \\ * & * & * & * & * \end{pmatrix}$$

where '$*$' indicates an element provided by the user and '?' indicates an element to be estimated, this procedure will call `obj_fun` twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1, 4 and 5 are known, they require no calls to `obj_fun`.) Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* [10] for a discussion of limiting accuracy). At times, central differences are used rather than forward differences, in which case twice as many calls to `obj_fun` are needed. (The switch to central differences is determined by considerations of accuracy and is not under user control.)

`f_deriv` = `.true.` should be used whenever possible, since this procedure is more reliable (and will usually be more efficient) when all derivatives are exact.

Default: `f_deriv` = `.true.`.

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

Output: the Jacobian matrix of the subfunctions at the final iterate (or its finite difference approximation), i.e., `f_jac`(i, j) contains the value of the partial derivative $\partial f_i/\partial x_j$ at the final point given in `x`, for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

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

Input: the coefficients of the constant vector y.

Default: `y` = 0.0.

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

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

Input: the lower and upper bounds on all the variables. To specify a non-existent lower bound (i.e., $l_j = -\infty$), set `x_lower`$(j) \leq -$`control%inf_bound`; to specify a non-existent upper bound (i.e., $u_j = +\infty$), set `x_upper`$(j) \geq +$`control%inf_bound` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

Constraints:

`x_lower`$(j) \leq$ `x_upper`(j) for $j = 1, 2, \ldots, n$;

$|\,\beta\,| <$ `control%inf_bound` when `x_lower`(j) = `x_upper`$(j) = \beta$.

Default: `x_lower` = $-$`control%inf_bound`; `x_upper` = $+$`control%inf_bound`.

a(n_{L}, n) — real(kind=*wp*), intent(in), optional

Input: the ith row of `a` must contain the coefficients of the ith linear constraint, for $i = 1, 2, \ldots, n_{\text{L}}$.

Default: the problem contains no linear constraints (i.e., $n_{\text{L}} = 0$).

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

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

Input: the lower and upper bounds on all the linear constraints. To specify a non-existent lower bound (i.e., $l_j = -\infty$), set `lin_lower(`j`)` $\leq$ −`control%inf_bound`; to specify a non-existent upper bound (i.e., $u_j = +\infty$), set `lin_upper(`j`)` $\geq$ +`control%inf_bound` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

Constraints:

`lin_lower` and `lin_upper` must not be present unless `a` is present;

`lin_lower(`j`)` $\leq$ `lin_upper(`j`)` for $j = 1, 2, \ldots, n_L$;

$|\beta| <$ `control%inf_bound` when `lin_lower(`j`)` = `lin_upper(`j`)` = β.

Default: `lin_lower` = −`control%inf_bound`; `lin_upper` = +`control%inf_bound`.

num_nlin_con — integer, intent(in), optional

Input: the number of nonlinear constraints, n_N.

Constraints: `num_nlin_con` must be present if `con_fun` is present; `num_nlin_con` ≥ 0.

Default: `num_nlin_con` = 0.

con_deriv — logical, intent(in), optional

Input: specifies whether or not all elements of the constraint Jacobian are provided by the user-supplied procedure `con_fun`.

If `con_deriv = .true.` (the default), then all elements of the constraint Jacobian must be provided by `con_fun` via its argument `con_jac`.

If `con_deriv = .false.`, then it is assumed that some elements of the constraint Jacobian are not provided; this procedure will estimate them using finite differences. The computation of finite difference approximations usually increases the total run-time, since a call to `con_fun` is needed for each variable for which partial derivatives are estimated. For example, if the Jacobian has the form

$$\begin{pmatrix} * & * & * & * & * \\ * & ? & ? & * & * \\ * & * & ? & * & * \\ * & * & * & * & * \end{pmatrix}$$

where '$*$' indicates an element provided by the user and '?' indicates an element to be estimated, this procedure will call `con_fun` twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1, 4 and 5 are known, they require no calls to `con_fun`.) Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* [10] for a discussion of limiting accuracy). At times, central differences are used rather than forward differences, in which case twice as many calls to `con_fun` are needed. (The switch to central differences is determined by considerations of accuracy and is not under user control.)

`con_deriv = .true.` should be used whenever possible, since this procedure is more reliable (and will usually be more efficient) when all derivatives are exact.

Constraints: `con_deriv` must not be present unless `con_fun` and `num_nlin_con` are present.

Default: `con_deriv = .true.`.

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

Output: `con_f(`i`)` contains the value of the ith nonlinear constraint function c_i at the final iterate, for $i = 1, 2, \ldots, n_N$.

Constraints: `con_f` must not be present unless `con_fun` and `num_nlin_con` are present.

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

Output: the Jacobian matrix of the nonlinear constraint functions at the final iterate (or its finite difference approximation), i.e., `con_jac`(i, j) contains the value of the partial derivative $\partial c_i / \partial x_j$ at the final point given in `x`, for $i = 1, 2, \ldots, n_N$ and $j = 1, 2, \ldots, n$.

Constraints: `con_jac` must not be present unless `con_fun` and `num_nlin_con` are present.

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

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

Input: the lower and upper bounds on all the nonlinear constraints. To specify a non-existent lower bound (i.e., $l_j = -\infty$), set `nlin_lower`(j) $\leq$ −`control%inf_bound`; to specify a non-existent upper bound (i.e., $u_j = +\infty$), set `nlin_upper`(j) $\geq$ +`control%inf_bound` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

Constraints:

`nlin_lower` and `nlin_upper` must not be present unless `con_fun` and `num_nlin_con` are present;

`nlin_lower`(j) $\leq$ `nlin_upper`(j) for $j = 1, 2, \ldots, n_N$;

$|\beta| <$ `control%inf_bound` when `nlin_lower`(j) = `nlin_upper`(j) = β.

Default: `nlin_lower` = −`control%inf_bound`; `nlin_upper` = +`control%inf_bound`.

con_fun — subroutine, optional

The procedure `con_fun`, supplied by the user, must calculate the vector $c(x)$ of nonlinear constraint functions and (optionally) its Jacobian ($= \partial c / \partial x$) at a specified point x.

Its specification is:

```
subroutine con_fun(first_call, x, finish, needc, con_f, con_jac)
```

```
logical, intent(in) :: first_call
```

Input: `first_call` will be `.true.` when this procedure calls `con_fun` for the first time, and `.false.` for all subsequent calls. It allows you to save computation time if certain data must be read or calculated only once. See also the description of `con_jac`.

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

Shape: `x` has shape (n).

Input: the point x at which the constraint functions and (optionally) elements of the constraint Jacobian are to be evaluated.

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

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

Output: if you wish to terminate the call to this procedure, you should set `finish` to `.true.`, and then this procedure will terminate with `error%code` = 201.

```
integer, intent(in) :: needc(:)
```

Shape: `needc` has shape (n_N).

Input: specifies the indices of the elements of `con_f` and (optionally) `con_jac` that must be evaluated. If `needc`(i) > 0, then the ith element of `con_f`, and (optionally) elements of the ith row of `con_jac`, must be evaluated at x, for $i = 1, 2, \ldots, n_N$.

```
real(kind=wp), intent(inout) :: con_f(:)
```

Shape: `con_f` has shape (n_N).

Input: the zero vector.

Output: if `needc`(i) > 0, `con_f`(i) must contain the value of the ith nonlinear constraint at the point x, for $i = 1, 2, \ldots, n_N$. Otherwise, `con_f`(i) need not be set.

real(kind=*wp*), intent(inout), optional :: con_jac(:,:)

Shape: con_jac has shape (n_N, n).

Input: if con_jac is present, its elements must remain unchanged except as specified below.

Output: if con_jac is present, then for each i such that needc(i) > 0:

if con_deriv = .true.(the default), the ith row of con_jac must contain *all* the elements of the vector ∇c_i given by

$$\nabla c_i = \left(\frac{\partial c_i}{\partial x_1}, \frac{\partial c_i}{\partial x_2}, \ldots, \frac{\partial c_i}{\partial x_n} \right)^T,$$

where $\partial c_i / \partial x_j$ is the partial derivative of the ith constraint with respect to the jth variable evaluated at the point x, for $i = 1, 2, \ldots, n_N$ and $j = 1, 2, \ldots, n$. Constant elements need be loaded into con_jac only during the first call to con_fun (when first_call = .true.). This facility is useful when many Jacobian elements are identically zero, in which case con_jac may be initialized to zero during the first call to con_fun. Note that although a constant non-zero element con_jac(i, j) only needs to be set on the first call to con_fun, the corresponding i in the definition of con_f(i) *must* be re-evaluated each time that con_fun is called.

If con_deriv = .false., any *available* partial derivatives of $c_i(x)$ must be assigned to the corresponding elements in the ith row of con_jac; the remaining elements *must remain unchanged*. Just before con_fun is called, each element of con_jac is set to a special value. On return from this procedure, any element that retains the value is estimated by finite differences, at non-trivial expense. If you do not supply a value for control%diff_int (see the type definition for nag_con_nlin_lsq_cntrl_*wp*), an interval for each element of x is computed automatically at the start of the optimization. The automatic procedure can usually identify constant elements of con_jac, which are then computed once only by finite differences.

If needc(i)≤ 0, the ith row of con_jac need not be set.

Note: if there are any nonlinear constraints, then the first call to con_fun will precede the first call to obj_fun (see Section 3.1). con_fun should be thoroughly tested before being supplied to this procedure. The components cheap_test, con_verify and major_iter_lim of the optional argument control can be used to assist this process (see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

Constraints: con_fun must be present if num_nlin_con is present and greater than zero.

cold_start — logical, intent(in), optional

Input: controls the specification of the initial working set in both the procedure for finding a feasible point for the linear constraints and bounds, and in the first QP subproblem thereafter.

With a *cold start* (i.e., cold_start = .true.), this procedure chooses the first working set based on the values of the variables and constraints at the initial point. Broadly speaking, the initial working set will include equality constraints and bounds or inequality constraints that violate or 'nearly' satisfy their bounds (to within the crash tolerance control%crash_tol; see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

With a *warm start* (i.e., cold_start = .false.), the arrays x_state, lin_state (if $n_L > 0$), nlin_state and nlin_lambda (if $n_N > 0$) together with the array r, must be supplied and initialized. The arrays x_state and lin_state determine the initial working set of the procedure to find a feasible point with respect to the bounds and linear constraints, whereas the array nlin_state determines the initial working set of the first QP subproblem after such a feasible point has been found. This procedure will override the contents of these arrays if necessary, so that a poor choice of the working set will not cause a fatal error. A warm start will be advantageous if a good estimate of the initial working set is available, for example when this procedure is called repeatedly to solve related problems.

Default: cold_start = .true..

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

Input: if `cold_start` = `.true.` (the default), `x_state` need not be initialized.

If `cold_start` = `.false.`, `x_state` specifies the status of the upper and lower bounds on the variables which together with the array `lin_state` define the initial working set for the procedure that finds a feasible point for the linear constraints and bounds. Possible values for `x_state`(j) are as follows:

`x_state`(j)	Meaning
0	The corresponding constraint should *not* be in the initial QP working set.
1	This constraint should be in the working set at its lower bound.
2	This constraint should be in the working set at its upper bound.
3	This constraint should be in the initial working set. This value must not be specified unless the corresponding lower and upper bounds are equal.

Any other values will be modified by this procedure. Note that `x_state` already contains valid values if it was present in a previous call with the same value of n. (See also the description of `cold_start`.) This procedure also adjusts (if necessary) the values supplied in `x` to be consistent with `x_state`.

Output: the status of the constraints in the QP working set at the point returned in `x`. The significance of each possible value of `x_state`(j) is as follows:

`x_state`(j)	Meaning
-2	This constraint violates its lower bound by more than the linear feasibility tolerance `control%lin_feas_tol` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). This value can only occur when no feasible point can be found for a QP subproblem.
-1	This constraint violates its upper bound by more than the linear feasibility tolerance. This value can only occur when no feasible point can be found for a QP subproblem.
0	This constraint is satisfied to within the linear feasibility tolerance, but is not in the QP working set.
1	This constraint is included in the QP working set at its lower bound.
2	This constraint is included in the QP working set at its upper bound.
3	This constraint is included in the QP working set as an equality. This can only occur when the corresponding upper and lower bounds are equal.

Constraints: if `cold_start` = `.false.`, `x_state` must be present.

lin_state(n_{L}) — integer, intent(inout), optional

Input: if `cold_start` = `.true.` (the default), `lin_state` need not be initialized.

If `cold_start` = `.false.`, `lin_state` specifies the status of the upper and lower bounds on the linear constraints which together with the array `x_state` define the initial working set for the procedure that finds a feasible point for the linear constraints and bounds. Possible values for `lin_state`(j) are as follows:

`lin_state`(j)	Meaning
0	The corresponding constraint should *not* be in the initial QP working set.
1	This constraint should be in the working set at its lower bound.
2	This constraint should be in the working set at its upper bound.
3	This constraint should be in the initial working set. This value must not be specified unless the corresponding lower and upper bounds are equal.

Any other values will be modified by this procedure. Note that `lin_state` already contains valid values if it was present in a previous call with the same value of n_{L}. (See also the description of `cold_start`.)

Output: the status of the constraints in the QP working set at the point returned in `x`. The significance of each possible value of `lin_state`(j) is as follows:

`lin_state`(j)	Meaning
-2	This constraint violates its lower bound by more than the linear feasibility tolerance `control%lin_feas_tol` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). This value can only occur when no feasible point can be found for a QP subproblem.
-1	This constraint violates its upper bound by more than the linear feasibility tolerance. This value can only occur when no feasible point can be found for a QP subproblem.
0	This constraint is satisfied to within the linear feasibility tolerance, but is not in the QP working set.
1	This constraint is included in the QP working set at its lower bound.
2	This constraint is included in the QP working set at its upper bound.
3	This constraint is included in the QP working set as an equality. This can only occur when the corresponding upper and lower bounds are equal.

Constraints: `lin_state` must not be present unless `a` is present. If `cold_start` = `.false.`, `lin_state` must be present if $n_{\rm L} > 0$.

nlin_state($n_{\rm N}$) — integer, intent(inout), optional

Input: if `cold_start` = `.true.` (the default), `nlin_state` need not be initialized.

If `cold_start` = `.false.`, `nlin_state` specifies the status of the upper and lower bounds on the nonlinear constraints, which together with the active set at the conclusion of the procedure to find a feasible point for the linear constraints and bounds, define the initial working set for the first QP subproblem. Possible values for `nlin_state`(j) are as follows:

`nlin_state`(j)	Meaning
0	The corresponding constraint should *not* be in the initial QP working set.
1	This constraint should be in the working set at its lower bound.
2	This constraint should be in the working set at its upper bound.
3	This constraint should be in the initial working set. This value must not be specified unless the corresponding lower and upper bounds are equal.

Any other values will be modified by this procedure. Note that `nlin_state` already contains valid values if it was present in a previous call with the same value of $n_{\rm N}$. (See also the description of `cold_start`.)

Output: the status of the constraints in the QP working set at the point returned in `x`. The significance of each possible value of `nlin_state`(j) is as follows:

`nlin_state`(j)	Meaning
-2	This constraint violates its lower bound by more than the nonlinear feasibility tolerance `control%nlin_feas_tol` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). This value can only occur when no feasible point can be found for a QP subproblem.
-1	This constraint violates its upper bound by more than the nonlinear feasibility tolerance. This value can only occur when no feasible point can be found for a QP subproblem.
0	This constraint is satisfied to within the nonlinear feasibility tolerance, but is not in the QP working set.
1	This constraint is included in the QP working set at its lower bound.
2	This constraint is included in the QP working set at its upper bound.
3	This constraint is included in the QP working set as an equality. This can only occur when the corresponding upper and lower bounds are equal.

Constraints: `nlin_state` must not be present unless `con_fun` and `num_nlin_con` are present. If `cold_start` = `.false.`, `nlin_state` must be present if $n_{\rm N} > 0$.

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

Output: the values of the QP multipliers for the bound constraints from the last QP subproblem. `x_lambda`(j) should be non-negative if `x_state`(j) = 1 and non-positive if `x_state`(j) = 2.

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

Output: the values of the QP multipliers for the linear constraints from the last QP subproblem. `lin_lambda`(j) should be non-negative if `lin_state`(j) = 1 and non-positive if `lin_state`(j) = 2.

Constraints: `lin_lambda` must not be present unless `a` is present.

nlin_lambda(n_N) — real(kind=*wp*), intent(inout), optional

Input: if `cold_start` = `.true.` (the default), `nlin_lambda` need not be initialized. If `cold_start` = `.false.`, `nlin_lambda` must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by the array `nlin_state`.

Note that:

if the jth constraint is defined as 'inactive' (`nlin_state`(j) = 0), `nlin_lambda`(j) should be zero;

if the jth constraint is an inequality active at its lower bound (`nlin_state`(j) = 1), `nlin_lambda`(j) should be non-negative;

if the jth constraint is an inequality active at its upper bound (`nlin_state`(j) = 2), `nlin_lambda`(j) should be non-positive.

If necessary, this procedure will modify `nlin_lambda` to match these rules.

Output: the values of the QP multipliers for the nonlinear constraints from the last QP subproblem. `nlin_lambda`(j) should be non-negative if `nlin_state`(j) = 1 and non-positive if `nlin_state`(j) = 2.

Constraints: `nlin_lambda` must not be present unless `con_fun` and `num_nlin_con` are present. If `cold_start` = `.false.`, `nlin_lambda` must be present if $n_N > 0$.

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

Input: if `cold_start` = `.true.` (the default), `r` need not be initialized.

If `cold_start` = `.false.`, `r` must contain the upper triangular Cholesky factor R of the initial approximation of the Hessian of the Lagrangian function, with the variables in the natural order. Elements in the strictly lower triangular part of `r` are assumed to be zero and need not be assigned.

Note that `r` already contains satisfactory information if it was present in a previous call to this procedure with `control%hessian` = `.true.` (the default; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

Output: if `control%hessian` = `.true.`, `r` contains the upper triangular Cholesky factor R of H, the approximate (untransformed) Hessian of the Lagrangian, with the variables in the natural order.

If `control%hessian` = `.false.`, `r` contains the upper triangular Cholesky factor R of $Q^T\tilde{H}Q$, an estimate of the transformed and re-ordered Hessian of the Lagrangian at x (see (10) in Section 1 of the Mathematical Background section of this module document).

Constraints: if `cold_start` = `.false.`, `r` must be present.

major_iter — integer, intent(out), optional

Output: the number of major iterations performed.

minor_iter — integer, intent(out), optional

Output: the number of minor iterations performed.

control — type(nag_con_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_con_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.
305	Invalid absence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code **Description**

201 User requested termination.

This exit occurs if you have set `finish` to `.true.` in `obj_fun` or `con_fun`.

202 No feasible point was found for the linear constraints and bounds, which means that either no feasible point exists for the given value of `control%lin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*), or no feasible point could be found in the number of iterations specified by `control%minor_iter_lim` (default value = $\max(50,3(n+n_{\rm L}+n_{\rm N}))$).

You should check that there are no constraint redundancies. If the data for the constraints are accurate only to an absolute precision σ, you should ensure that the value of `control%lin_feas_tol` is *greater* than σ. For example, if all the elements of $A_{\rm L}$ are of order unity and are accurate only to three decimal places, then `control%lin_feas_tol` should be at least 10^{-3}.

203 No feasible point could be found for the nonlinear constraints. The problem may have no feasible solution. This means that there has been a sequence of QP subproblems for which no feasible point could be found (indicated by `I` at the end of each line of intermediate printout produced by the major iterations; see Section 7.1).

This behaviour will occur if there is no feasible point for the nonlinear constraints. (However, there is no general test that can determine whether a feasible point exists for a set of nonlinear constraints.) If the infeasible subproblems occur from the very first major iteration, it is highly likely that no feasible point exists. If infeasibilities occur when earlier subproblems have been feasible, small constraint inconsistencies may be present. You should check the validity of constraints with negative values of `nlin_state` (see Section 3.2). If you are convinced that a feasible point *does* exist, this procedure should be restarted at a different starting point.

204 x does not satisfy the first-order Kuhn–Tucker conditions (see Section 1 of the Mathematical Background section of this module document), and no improved point for the merit function (see Section 7.1) could be found during the final linesearch.

This sometimes occurs because an overly stringent accuracy has been requested, i.e., the value of `control%optim_tol` is too small (default value = $(\texttt{EPSILON(1.0\_}wp\texttt{)})^{0.72}$; see the type definition of `nag_con_nlin_lsq_cntrl_`*wp*). In this case you should apply the following tests to determine whether or not the final solution is acceptable (see Gill *et al.* [10], for a discussion of the attainable accuracy):

(a) the final value of `Norm Gz` (see Section 7.1) is significantly less than that at the starting point;

(b) during the final major iterations, the values of `Step` and `Mnr` (see Section 7.1) are both one;

(c) the last few values of both `Norm Gz` and `Violtn` (see Section 7.1) become small at a fast linear rate; and

(d) `Cond Hz` (see Section 7.1) is small.

If all these conditions hold, x is almost certainly a local minimum of (3).

If many iterations have occurred in which essentially no progress has been made and this procedure has failed completely to move from the initial point, then procedures `obj_fun` and/or `con_fun` may be incorrect. You should refer to the description of `error%code` = 205 and check the Jacobians using `control%cheap_test = .false.` (default value = `.true.`; see the type definition for **nag_con_nlin_lsq_cntrl_*wp***). Unfortunately, there may be small errors in the objective and constraint Jacobians that cannot be detected by the verification. Finite difference approximations to first derivatives can be catastrophically affected even by small inaccuracies. An indication of this situation is a dramatic alteration in the iterates if the finite difference interval is altered. One might also suspect this type of error if a switch is made to central differences even when `Norm Gz` and `Violtn` (see Section 7.1) are large.

Another possibility is that the search direction has become inaccurate because of ill conditioning in the Hessian approximation or the matrix of constraints in the working set; either form of ill conditioning tends to be reflected in large values of `Mnr` (the number of iterations required to solve each QP subproblem; see Section 7.1).

If the condition estimate of the projected Hessian (`Cond Hz`; see Section 7.1) is extremely large, it may be worthwhile rerunning this procedure from the final point using `cold_start = .false.` (see Section 3.2). In this situation `x_state`, `lin_state` (if $n_L > 0$), `nlin_state` and `nlin_lambda` (if $n_N > 0$; see Section 3.2) should be left unaltered, and R should be reset to the identity matrix.

If the condition estimate of the matrix of constraints in the working set (`Cond T`; see Section 7.1) is extremely large, it may be worthwhile rerunning this procedure with relaxed values of `control%lin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`) and/or `control%nlin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))` or `(EPSILON(1.0_`*wp*`))`$^{0.33}$; see the type definition for **nag_con_nlin_lsq_cntrl_*wp***). (Constraint dependencies are often indicated by wide variations in size in the diagonal elements of the matrix T, whose diagonals will be printed if the printing parameter `control%major_print_level` $\geq$ 30 (default value = 10; see Section 7.1).)

205 The user-provided derivatives of the subfunctions and/or constraints appear to be incorrect.

Large errors were found in the derivatives of the subfunctions and/or constraints. This exit occurs if the verification process indicated that at least one Jacobian element had no correct figures. You should refer to the printed output to determine which elements are suspected to be in error.

As a first step, you should check that the code for computing the nonlinear functions and constraints 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 functions and constraints. 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.

Jacobian checking will be ineffective if the subfunctions (see $f_i(x)$ in (3)) use information computed by the constraints, since they are not necessarily computed prior to each evaluation.

Errors in programming the subfunctions or constraints may be quite subtle in that the values are 'almost' correct. For example, the nonlinear function value may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which it depends.

Warnings (error%level = 1):

error%code **Description**

101 The final iterate x satisfies the first-order Kuhn–Tucker conditions (see Section 1 of the Mathematical Background section of this module document) to the accuracy requested, but the sequence of iterates has not yet converged. This procedure was terminated because no further improvement could be made in the merit function (see Section 7.1).

This exit may occur in several circumstances. The most common situation is that you have asked for a solution with accuracy that is not attainable with the given precision of the problem (as specified by `control%fun_prec` (default value = (`EPSILON(1.0_`*wp*`)`)$^{0.9}$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*)). This condition will also occur if, by chance, an iterate is an 'exact' Kuhn–Tucker point, but the change in the variables was significant at the previous iteration. (This situation often happens when minimizing very simple functions, such as quadratics.)

If conditions (a)–(d) described under `error%code` = 204 are satisfied, x is likely to be a solution of (3) even if `error%code` = 101.

102 The limiting number of iterations was reached before normal termination occurred.

If the algorithm appears to be making satisfactory progress, then the value of `control%major_iter_lim` (default value = $\max(50, 3 \times (n + n_{\rm L}) + 10 \times n_{\rm N})$); see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*) may be too small. If so, either increase its value and rerun this procedure or, alternatively, rerun this procedure using `cold_start` = `.false.` (see Section 3.2). If the algorithm seems to be making little or no progress however, then you should check for incorrect Jacobians or ill conditioning (as described under `error%code` = 204).

Note that ill conditioning in the working set is sometimes resolved automatically by the algorithm, in which case performing additional iterations may be helpful. However, ill conditioning in the Hessian approximation tends to persist once it has begun, so that allowing additional iterations without altering R is usually inadvisable. If the quasi-Newton update of the Hessian approximation was reset during the latter major iterations (i.e., an `R` occurs at the end of each line of intermediate printout; see Section 7.1), it may be worthwhile rerunning this procedure using `cold_start` = `.false.` (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

6.1 Accuracy

If `error%code` = 0 on exit, then the vector returned in the array `x` is an estimate of the solution to an accuracy of approximately `control%optim_tol` (default value = (`EPSILON(1.0_`*wp*`)`)$^{0.72}$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).

6.2 Termination Criteria

This procedure returns with `error%code` = 0 if the iterates have converged to a point x^* that satisfies the first-order Kuhn–Tucker conditions (see Section 1 of the Mathematical Background section of this module document) to the accuracy requested by `control%optim_tol` (default value = (`EPSILON(1.0_`*wp*`)`)$^{0.72}$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*), i.e., the projected gradient and active constraint residuals are negligible at x^*.

6.3 Overflow

If the printed output before the overflow error contains a warning about serious ill conditioning in the working set when adding the jth constraint, it may be possible to avoid the difficulty by increasing the magnitude of `control%nlin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))` or (`EPSILON(1.0_`*wp*`)`)$^{0.33}$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*) and/or `control%lin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`) and rerunning the program. If the message recurs even after this change, the offending linearly dependent constraint (with index 'j') *must* be removed from the problem.

If overflow occurs in one of the user-supplied procedures (e.g., if the subfunctions involve exponentials or singularities), it may help to specify tighter bounds for some of the variables (i.e., reduce the gap between the appropriate l_j and u_j).

7 Description of Printed Output

7.1 Major Iteration Printout

This section describes the intermediate and final printout produced by the major iterations of this procedure (see Section 1 of the Mathematical Background section of this module document). The level of printed output can be controlled via the components `list` and `major_print_level` of the optional argument `control`. 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 are produced only if `control%major_print_level` ≥ 10 (the default).

When `control%major_print_level` ≥ 5 and `control%lt80_char` = `.true.` (the default), the following line of output (< 80 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Maj` is the major iteration count.

`Mnr` is the number of minor iterations required by the feasibility and optimality phases of the QP subproblem. Generally, `Mnr` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see the Mathematical Background section of this module document).
Note that `Mnr` may be greater than `control%minor_iter_lim` (default value = $\max(50, 3\times(n + n_{\mathrm{L}} + n_{\mathrm{N}}))$; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*) if some iterations are required for the feasibility phase.

`Step` is the step taken along the computed search direction. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.

`Merit Function` is the value of the augmented Lagrangian merit function (see Section 3 of the Mathematical Background section of this module document) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters As the solution is approached, `Merit Function` will converge to the value of the objective function at the solution.
If the QP subproblem does not have a feasible point (signified by `I` at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty parameters. During a sequence of major iterations with infeasible subproblems, the sequence of `Merit Function` values will decrease monotonically until either a feasible subproblem is obtained or this procedure terminates with `error%code` = 203 (no feasible point could be found for the nonlinear constraints).
If no nonlinear constraints are present (i.e., $n_N = 0$), this entry contains `Objective`, the value of the objective function $F(x)$. The objective function will decrease monotonically to its optimal value when there are no nonlinear constraints.

`Norm Gz` is $\|Z^T g_{FR}\|$, the Euclidean norm of the projected gradient (see Section 2 of the Mathematical Background section of this module document). `Norm Gz` will be approximately zero in the neighbourhood of a solution.

`Violtn` is the Euclidean norm of the residuals of nonlinear constraints that are violated or in the predicted active set (not printed if $n_N = 0$). `Violtn` will be approximately zero in the neighbourhood of a solution.

`Cond Hz` is a lower bound on the condition number of the projected Hessian approximation H_Z ($H_Z = Z^T H_{FR} Z = R_Z^T R_Z$; see (10) in Section 1 and (15) in Section 2 of the Mathematical Background section of this module document). The larger this number, the more difficult the problem.

`M` is printed if the quasi-Newton update has been modified to ensure that the Hessian approximation is positive definite (see Section 4 of the Mathematical Background section of this module document).

`I` is printed if the QP subproblem has no feasible point.

`C` is printed if central differences have been used to compute the unspecified objective and constraint Jacobians. If the value of `Step` is zero, the switch to central differences was made because no lower point could be found in the linesearch. (In this case, the QP subproblem is re-solved with the central difference gradient and Jacobian.)
If the value of `Step` is non-zero, central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 1 of the Mathematical Background section of this module document).

`L` is printed if the linesearch has produced a relative change in x greater than the value defined by `control%step_limit` (default value = 2.0; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*).
If this output occurs frequently during later iterations of the run, `control%step_limit` should be set to a larger value.

`R` is printed if the approximate Hessian has been refactorized. If the diagonal condition estimator of R indicates that the approximate Hessian is badly conditioned, the approximate Hessian is refactorized using column interchanges. If necessary, R is modified so that its diagonal condition estimator is bounded.

When `control%major_print_level` ≥ 5 and `control%lt80_char` = `.false.`, the following line of output (up to 132 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Maj` (as above)

`Mnr` (as above)

`Step` (as above)

`Nfun` is the cumulative number of evaluations of the objective function needed for the linesearch. Evaluations needed for the estimation of the Jacobians by finite differences are not included. `Nfun` is printed as a guide to the amount of work required for the linesearch.

`Merit Function`	(as above)
`Norm Gz`	(as above)
`Violtn`	(as above)
`Nz`	is the number of columns of Z (see Section 2 of the Mathematical Background section of this module document). The value of `Nz` is the number of variables minus the number of constraints in the predicted active set; i.e., `Nz` $= n - ($ `Bnd` $+$ `Lin` $+$ `Nln`$)$.
`Bnd`	is the number of simple bound constraints in the predicted active set.
`Lin`	is the number of general linear constraints in the predicted active set.
`Nln`	is the number of nonlinear constraints in the predicted active set (not printed if $n_N = 0$).
`Penalty`	is the Euclidean norm of the vector of penalty parameters used in the augmented Lagrangian merit function (not printed if $n_N = 0$).
`Cond H`	is a lower bound on the condition number of the Hessian approximation H.
`Cond Hz`	(as above)
`Cond T`	is a lower bound on the condition number of the matrix of predicted active constraints.
`Conv`	is a three-letter indication of the status of the three convergence tests defined in the description of `control%optim_tol` (see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). Each letter is `T` if the test is satisfied, and `F` otherwise. The three tests indicate whether: (a) the sequence of iterates has converged; (b) the projected gradient (`Norm Gz`) is sufficiently small; and (c) the norm of the residuals of constraints in the predicted active set (`Violtn`) is small enough. If any of these indicators is `F` on termination with `error%level` = 0, you should check the solution carefully.
`M`	(as above)
`I`	(as above)
`C`	(as above)
`L`	(as above)
`R`	(as above)

The final printout includes a listing of the status of every variable and constraint.

The following describes the printout for each variable. A full stop (.) is printed for any numerical value that is zero.

`Varbl` gives the name (`V`) and index j, for $j = 1, 2, \ldots, n$ of the variable.

`State` gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than `control%lin_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*), `State` will be `++` or `--` respectively. (The latter situation can occur only when there is no feasible point for the bounds and linear constraints.)
A key is sometimes printed before `State` to give additional information about the state of a variable.

`A` *Alternative optimum possible.* The variable is active at one of its bounds, but its Lagrange multiplier is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change to the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them would encounter a bound immediately. In either case the values of the Lagrange multipliers might also change.

D	*Degenerate.* The variable is free, but it is equal to (or very close to) one of its bounds.
I	*Infeasible.* The variable is currently violating one of its bounds by more than control%lin_feas_tol.

`Value` is the value of the variable at the final iterate.

`Lower Bound` is the lower bound specified for the variable. `None` indicates that x_lower(j) $\leq$ $-$control%inf_bound (default value $= 10^{20}$; see the type definition for nag_con_nlin_lsq_cntrl_*wp*).

`Upper Bound` is the upper bound specified for the variable. `None` indicates that x_upper(j) $\geq$ control%inf_bound.

`Lagr Mult` is the Lagrange multiplier for the associated bound. This will be zero if `State` is `FR` unless x_lower(j) $\leq$ $-$control%inf_bound and x_upper(j) $\geq$ control%inf_bound, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is `LL`, and non-positive if `State` is `UL`.

`Slack` is the difference between the variable `Value` and the nearer of its (finite) bounds x_lower(j) and x_upper(j). A blank entry indicates that the associated variable is not bounded (i.e., x_lower(j) $\leq$ $-$control%inf_bound and x_upper(j) $\geq$ control%inf_bound).

The meaning of the printout for linear and nonlinear constraints is the same as that given above for variables, with 'variable' replaced by 'constraint', x_lower and x_upper are replaced by either lin_lower and lin_upper, or by nlin_lower and nlin_upper, respectively, control%lin_feas_tol is replaced by control%nlin_feas_tol for the nonlinear constraints and with the following changes in the heading:

`L Con` gives the name (L) and index j, for $j = 1, 2, \ldots, n_{\mathrm{L}}$ of the linear constraint.

`N Con` gives the name (N) and index j, for $j = 1, 2, \ldots, n_{\mathrm{N}}$ of the nonlinear constraint.

Note that movement off a constraint (as opposed to a variable moving away from its bound) can be interpreted as allowing the entry in the `Slack` column to become positive.

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

7.2 Minor Iteration Printout

This section describes the intermediate and final printout produced by the minor iterations of this procedure, which involves solving a QP subproblem of the form

$$\underset{p}{\text{minimize}}\ g^T p + \tfrac{1}{2} p^T H p \quad \text{subject to} \quad \bar{l} \leq \left\{ \begin{array}{c} p \\ A_{\mathrm{L}} p \\ A_{\mathrm{N}} p \end{array} \right\} \leq \bar{u} \tag{4}$$

at every major iteration. (For more details see Section 1 of the Mathematical Background section of this module document.) The level of printed output can be controlled via the component minor_print_level of the optional argument control. Note that the intermediate printout and the final printout are produced only if control%minor_print_level ≥ 10 (default value $= 0$, which produces no output).

When control%minor_print_level ≥ 5 and control%lt80_char = .true., the following line of output (< 80 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration of the QP subproblem.

`Itn` is the iteration count.

`Step` is the step taken along the computed search direction. If a constraint is added during the current iteration, `Step` will be the step to the nearest constraint. During the optimality phase, the step can be greater than one only if the factor R_Z is singular (see Section 2 of the Mathematical Background section of this module document).

`Ninf` is the number of violated constraints (infeasibilities). This will be zero during the optimality phase.

`Sinf/Objective`	is the value of the current objective function. If x is not feasible, `Sinf` gives a weighted sum of the magnitudes of the constraint violations. If x is feasible, `Objective` is the value of the QP objective function in (4). The output line for the final iteration of the feasibility phase (i.e., the first iteration for which `Ninf` is zero) will give the value of the true objective at the first feasible point. During the optimality phase, the value of the objective function will be non-increasing. During the feasibility phase, the number of constraint infeasibilities will not increase until either a feasible point is found, or the optimality of the multipliers implies that no feasible point exists. Once optimal multipliers are obtained, the number of infeasibilities can increase, but the sum of infeasibilities will either remain constant or be reduced until the minimum sum of infeasibilities is found.
`Norm Gz`	is $\|Z^T q_{\mathrm{FR}}\|$, the Euclidean norm of the reduced gradient of the QP objective function in (4) with respect to Z (see Section 2 of the Mathematical Background section of this module document). During the optimality phase, this norm will be approximately zero after a unit step.

When `control%minor_print_level` ≥ 5 and `control%lt80_char` $=$ `.false.`, the following line of output (up to 120 characters) is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration of the QP subproblem. The following convention is used for numbering the constraints: indices 1 through n refer to the bounds on the variables, and indices $n+1$ through $n+n_{\mathrm{L}}$ or $n+n_{\mathrm{L}}+n_{\mathrm{N}}$ refer to the general constraints (if any). When the status of a constraint changes, the index of the constraint is printed, along with the designation `L` (lower bound), `U` (upper bound), `E` (equality), `F` (temporarily fixed variable) or `A` (artificial constraint).

`Itn`	(as above)
`Jdel`	is the index of the constraint deleted from the QP working set. If `Jdel` is zero, no constraint was deleted.
`Jadd`	is the index of the constraint added to the QP working set. If `Jadd` is zero, no constraint was added.
`Step`	(as above)
`Ninf`	(as above)
`Sinf/Objective`	(as above)
`Bnd`	is the number of simple bound constraints in the current QP working set.
`Lin`	is the number of general linear constraints in the current QP working set.
`Art`	is the number of artificial constraints in the QP working set.
`Zr`	is the dimension of the sub-space in which the QP objective function in (4) (Section 7.2) is currently being minimized. The value of `Zr` is the number of variables minus the number of constraints in the working set; i.e., `Zr` $= n -$ (`Bnd`+`Lin`+`Art`). The value of n_Z, the number of columns of Z (see Section 1 of the Mathematical Background section of this module document) can be calculated as $n_Z = n -$ (`Bnd`+`Lin`). A zero value of n_Z implies that x lies at a vertex of the feasible region.
`Norm Gz`	(as above)
`Norm Gf`	is $\|q_{\mathrm{FR}}\|$, the Euclidean norm of the gradient of the QP objective function in (4) with respect to the free variables, i.e., variables not currently held at a bound (see Section 2 of the Mathematical Background section of this module document).
`Cond T`	is a lower bound on the condition number of the QP working set.
`Cond Rz`	is a lower bound on the condition number of the triangular factor R_1 (the first `Zr` rows and columns of the factor R_Z; see Section 2 of the Mathematical Background section of this module document). If the estimated rank of the matrix H in (4) is zero, `Cond Rz` is not printed.

The final printout includes a listing of the status of every variable and constraint.

The following describes the printout for each variable. A full stop (.) is printed for any numerical value that is zero.

`Varbl`	gives the name (`V`) and index j, for $j = 1, 2, \ldots, n$ of the variable.
`State`	gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than `control%lin_feas_tol`, `State` will be `++` or `--` respectively. A key is sometimes printed before `State` to give additional information about the state of a variable.

- `A` *Alternative optimum possible.* The variable is active at one of its bounds, but its Lagrange multiplier is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change to the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them would encounter a bound immediately. In either case the values of the Lagrange multipliers might also change.
- `D` *Degenerate.* The variable is free, but it is equal to (or very close to) one of its bounds.
- `I` *Infeasible.* The variable is currently violating one of its bounds by more than `control%lin_feas_tol`.

`Value`	is the value of the variable at the final iterate.
`Lower Bound`	is the lower bound specified for the variable. `None` indicates that $\bar{l}_j \leq -$`control%inf_bound` (default value $= 10^{20}$; see the type definition for `nag_con_nlin_lsq_cntrl_wp`).
`Upper Bound`	is the upper bound specified for the variable. `None` indicates that $\bar{u}_j \geq$ `control%inf_bound`.
`Lagr Mult`	is the Lagrange multiplier for the associated bound. This will be zero if `State` is `FR` unless $\bar{l}_j \leq -$`control%inf_bound` and $\bar{u}_j \geq$`control%inf_bound`, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is `LL`, and non-positive if `State` is `UL`.
`Slack`	is the difference between the variable `Value` and the nearer of its (finite) bounds $\bar{l}_j$ and $\bar{u}_j$. A blank entry indicates that the associated variable is not bounded (i.e., $\bar{l}_j \leq -$`control%inf_bound` and $\bar{u}_j \geq$ `control%inf_bound`).

The meaning of the printout for general constraints is the same as that given above for variables, with 'variable' replaced by 'constraint', $\bar{l}_j$ and $\bar{u}_j$ are replaced by $\bar{l}_{j+n}$ and $\bar{u}_{j+n}$ respectively, `control%lin_feas_tol` is replaced by `control%nlin_feas_tol` for the nonlinear constraints and with the following changes in the heading:

`L Con`	gives the name (`L`) and index j, for $j = 1, 2, \ldots, n_{\mathrm{L}} + n_{\mathrm{N}}$ of the constraint unless an initial feasible point (for the linear constraints and bounds) is being sought, in which case $j = 1, 2, \ldots, n_{\mathrm{L}}$.

Note that movement off a constraint (as opposed to a variable moving away from its bound) can be interpreted as allowing the entry in the `Slack` column to become positive.

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

Procedure: nag_con_nlin_lsq_cntrl_init

1 Description

`nag_con_nlin_lsq_cntrl_init` assigns default values to the components of a structure of the derived type `nag_con_nlin_lsq_cntrl_`*wp*.

2 Usage

```
USE nag_con_nlin_lsq

CALL nag_con_nlin_lsq_cntrl_init(control)
```

3 Arguments

3.1 Mandatory Argument

control — type(nag_con_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_con_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_con_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_con_nlin_lsq_cntrl_dp. For single precision the name is nag_con_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_con_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_con_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_con_nlin_lsq_cntrl_wp
  !  Printing parameters
  logical ::  list
  integer ::  unit
  logical ::  lt80_char
  integer ::  major_print_level
  integer ::  minor_print_level
  !  Derivative verification and approximation
  logical ::  initial_x
  logical ::  cheap_test
  logical ::  obj_verify
  integer ::  start_obj_check
  integer ::  stop_obj_check
  logical ::  con_verify
  integer ::  start_con_check
  integer ::  stop_con_check
  real(kind=wp) ::  diff_int
  real(kind=wp) ::  cent_diff_int
  !  Algorithm choice and tolerances
  integer ::  major_iter_lim
  integer ::  minor_iter_lim
  integer ::  reset_freq
  real(kind=wp) ::  inf_bound
  real(kind=wp) ::  inf_step
  real(kind=wp) ::  lin_feas_tol
  real(kind=wp) ::  nlin_feas_tol
  real(kind=wp) ::  crash_tol
  real(kind=wp) ::  fun_prec
  real(kind=wp) ::  optim_tol
  real(kind=wp) ::  linesearch_tol
  real(kind=wp) ::  step_limit
  logical ::  hessian
  logical ::  jtj_init_hess
end type nag_con_nlin_lsq_cntrl_wp
```

3 Components

3.1 Printing Parameters

list — logical

Controls the printing of the parameter settings in the call to nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1.

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_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1 is sent.

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

Constraints: a valid output unit.

lt80_char — logical

Controls the maximum length of each line of output produced by nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1.

If lt80_char = .true., then the output will not exceed 80 characters per line;

if lt80_char = .false., then the output will not exceed 132 characters per line whenever major_print_level ≥ 5 (the default) or minor_print_level ≥ 5 (default value = 0).

Default: lt80_char = .true..

major_print_level — integer

Controls the amount of output produced by the major iterations of nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1, as indicated below. A detailed description of the printed output is given in Section 7.1 of the appropriate procedure document.

If lt80_char = .true. (the default), the following output is sent to the Fortran unit number defined by unit:

	0	No output.
	1	The final solution only.
$\geq$	5	One line of summary output (< 80 characters) for each major iteration (no printout of the final solution).
$\geq$	10	The final solution and one line of summary output for each major iteration.

If lt80_char = .false., the following output is sent to the Fortran unit number defined by unit:

	0	No output.
	1	The final solution only.
$\geq$	5	One long line of output (up to 132 characters) for each major iteration (no printout of the final solution).
$\geq$	10	The final solution and one long line of output for each major iteration.
$\geq$	20	At each major iteration, the objective function, the Euclidean norm of the nonlinear constraint violations, the values of the nonlinear constraints (the vector c), the values of the linear constraints (the vector $A_{\mathrm{L}}x$), and the current values of the variables (the vector x).
$\geq$	30	At each major iteration, the diagonal elements of the matrix T associated with the TQ factorization (9) of the QP working set, as described in Section 1 of the Mathematical Background section of this module document, and the diagonal elements of R, the triangular factor of the transformed and re-ordered Hessian (10).

Default: major_print_level = 10.

minor_print_level — integer

Controls the amount of output produced by the minor iterations of nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1, as indicated below. A detailed description of the printed output is given in Section 7.2 of the appropriate procedure document.

If lt80_char = .true. (the default), the following output is sent to the Fortran unit number defined by unit:

	0	No output.
	1	The final QP solution only.
$\geq$	5	One line of summary output ($<$ 80 characters) for each minor iteration (no printout of the final QP solution).
$\geq$	10	The final QP solution and one line of summary output for each minor iteration.

If lt80_char = .false., the following output is sent to the Fortran unit number defined by unit:

	0	No output.
	1	The final QP solution only.
$\geq$	5	One long line of output (up to 132 characters) for each minor iteration (no printout of the final QP solution).
$\geq$	10	The final solution and one long line of output for each minor iteration.
$\geq$	20	At each minor iteration, the current estimates of the QP multipliers, the current estimate of the QP search direction, the QP constraint values, and the status of each QP constraint.
$\geq$	30	At each minor iteration, the diagonal elements of the matrix T associated with the TQ factorization (9) of the QP working set, as described in Section 1 of the Mathematical Background section of this module document, and the diagonal elements of the Cholesky factor R of the transformed Hessian (10).

Default: minor_print_level = 0.

3.2 Derivative Verification and Approximation

initial_x — logical

initial_x specifies the point at which the objective and constraint Jacobians are to be checked.

If initial_x = .true., then the check will be made at the user-specified initial value of x;

if initial_x = .false. (the default), then the check will be made at the first point that satisfies the linear constraints and bounds.

Default: initial_x = .false..

cheap_test — logical

cheap_test specifies the level of verification of elements computed by procedures obj_fun and con_fun (see Section 3 of the appropriate procedure document).

If cheap_test = .true. (the default), then only a 'cheap' test will be performed on the objective and constraint Jacobians at the point specified by initial_x (requiring one call to obj_fun and con_fun).

If cheap_test = .false., then a more reliable (but more expensive) check will be made on individual objective and constraint Jacobian elements at the point specified by initial_x (see the descriptions of obj_verify and con_verify).

Default: cheap_test = .true..

obj_verify — logical

Note: obj_verify only takes effect if cheap_test = .false. (default value = .true.).

It specifies whether or not individual elements of the objective Jacobian are to be checked. (Note that unspecified elements are not checked.)

If obj_verify = .true. (the default), then individual objective Jacobian elements within the range specified by start_obj_check (default value = 1) to stop_obj_check (default value = number of variables) will be checked at the point specified by initial_x. If major_print_level > 0 (the default), a result of the form OK or BAD? is printed to indicate whether or not each element appears to be correct.

If obj_verify = .false., then no checks will be performed on the objective Jacobian.

Default: obj_verify = .true..

start_obj_check — integer

Note: start_obj_check only takes effect if obj_verify = .true. (the default).

It specifies the first element of the objective Jacobian to be checked.

Default: start_obj_check = 1.

Constraints: see the description of stop_obj_check.

stop_obj_check — integer

Note: stop_obj_check only takes effect if obj_verify = .true. (the default).

It specifies the last element of the objective Jacobian to be checked.

Default: stop_obj_check = number of variables.

Constraints: 1 ≤ start_obj_check ≤ stop_obj_check ≤ number of variables.

con_verify — logical

Note: con_verify only takes effect if cheap_test = .false. (default value = .true.).

It specifies whether or not individual elements of the constraint Jacobian are to be checked. (Note that unspecified elements are not checked.)

If con_verify = .true. (the default), then individual Jacobian elements in columns start_con_check (default value = 1) to stop_con_check (default value = number of variables) will be checked at the point specified by initial_x. If major_print_level > 0 (the default), a result of the form OK or BAD? is printed to indicate whether or not each element appears to be correct.

If con_verify = .false., then no checks will be performed on the constraint Jacobian.

Default: con_verify = .true..

start_con_check — integer

Note: start_con_check only takes effect if con_verify = .true. (the default).

It specifies the first column of the constraint Jacobian to be checked.

Default: start_con_check = 1.

Constraints: see the description of stop_con_check.

stop_con_check — integer

Note: stop_con_check only takes effect if con_verify = .true. (the default).

It specifies the last column of the constraint Jacobian to be checked.

Default: stop_con_check = number of variables.

Constraints: 1 ≤ start_con_check ≤ stop_con_check ≤ number of variables.

diff_int — real(kind=*wp*)

diff_int defines an interval used to estimate derivatives in the following circumstances:

(a) for verifying the objective and constraint Jacobians (see the descriptions of cheap_test, obj_verify and con_verify);

(b) for estimating unspecified elements of the objective and constraint Jacobians.

In general, a derivative with respect to the jth variable is approximated using the interval δ_j, where δ_j = diff_int$\times(1+|\hat{x}_j|)$, with $\hat{x}$ the first point feasible with respect to the bounds and linear constraints. If the functions are well scaled, the resulting derivative approximation should be accurate to O(diff_int). See Gill *et al.* [10] for a discussion of the accuracy in finite difference approximations.

Default: a finite difference interval is computed automatically for each variable by a procedure that requires up to six calls of obj_fun and con_fun (see Section 3 of the appropriate procedure document). This option is recommended if the function is badly scaled or you wish to have nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1 determine constant elements in the objective and constraint Jacobians.

Constraints: EPSILON(1.0_*wp*) $\leq$ diff_int $<$ 1.0.

cent_diff_int — real(kind=*wp*)

cent_diff_int specifies the difference interval to be used for every element of x whenever the algorithm switches from forward differences to central differences (because the forward-difference approximation is not sufficiently accurate). The switch to central differences is indicated by C at the end of each line of intermediate output produced at the end of a major iteration; see Section 7.1 of the appropriate procedure document.

Default: a finite difference interval is computed automatically for each variable by a procedure that requires up to six calls of obj_fun and con_fun (see the description of diff_int).

Constraints: EPSILON(1.0_*wp*) $\leq$ ent_diff_int $<$ 1.0.

3.3 Algorithm Choice and Tolerances

major_iter_lim — integer

major_iter_lim specifies the maximum number of major iterations allowed before termination.

If you wish to check that a call to nag_con_nlin_lsq_sol or nag_con_nlin_lsq_sol_1 is correct before attempting to solve the problem in full then major_iter_lim may be set to 0. No major iterations will be performed but the initialization stages prior to the first major iteration will be processed and a listing of parameter settings output if list = .true. (the default). Any derivative checking (as specified by cheap_test, obj_verify and con_verify) will also be performed.

Default: major_iter_lim = max(50,3$\times$(number of variables + number of linear constraints) + 10$\times$number of nonlinear constraints).

Constraints: major_iter_lim $\geq$ 0.

minor_iter_lim — integer

minor_iter_lim specifies the maximum number of iterations for finding a feasible point with respect to the linear constraints and bounds on the variables. It also specifies the maximum number of minor iterations for the optimality phase of each QP subproblem.

Default: minor_iter_lim = max(50,3$\times$(number of variables + number of linear constraints + number of nonlinear constraints)).

Constraints: minor_iter_lim $\geq$ 1.

reset_freq — integer

Every reset_freq iterations the approximate Hessian matrix is reset to J^TJ, where J is the objective Jacobian matrix $\nabla f(x)$ (see the description of jtj_init_hess).

At any point where there are no nonlinear constraints active and the values of $f(x)$ are small in magnitude compared to the norm of J, J^TJ will be a good approximation to the objective Hessian matrix $\nabla^2 F(x)$. Under these circumstances, frequent resetting can significantly improve the convergence rate of nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1. Resetting is suppressed at any iteration during which there are nonlinear constraints active.

Default: reset_freq = 2.

Constraints: reset_freq ≥ 1.

inf_bound — real(kind=*wp*)

inf_bound defines the 'infinite' bound size in the definition of the problem constraints. Any upper bound greater than or equal to inf_bound will be regarded as $+\infty$ (and similarly any lower bound less than or equal to $-$inf_bound will be regarded as $-\infty$).

Default: inf_bound = 10^{20}.

Constraints: inf_bound > 0.0.

inf_step — real(kind=*wp*)

inf_step specifies the magnitude of the change in variables that will be considered a step to an unbounded solution. If the change in x during an iteration would exceed the value of inf_step, the objective function is considered to be unbounded below in the feasible region.

Default: inf_step = max(inf_bound, 10^{20}).

Constraints: inf_step $\geq$ inf_bound.

lin_feas_tol — real(kind=*wp*)

lin_feas_tol defines the maximum acceptable *absolute* violation in linear constraints at a 'feasible' point; i.e., a linear constraint is considered satisfied if its violation does not exceed lin_feas_tol.

On entry to nag_con_nlin_lsq_sol or nag_con_nlin_lsq_sol_1, an iterative procedure is executed in order to find a point that satisfies the linear constraints and bounds on the variables to within the tolerance specified by lin_feas_tol. All subsequent iterates will satisfy the linear constraints to within the same tolerance (unless lin_feas_tol is comparable to the finite difference interval).

lin_feas_tol should reflect the precision of the linear constraints. For example, if the variables and the coefficients in the linear constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify lin_feas_tol as 10^{-6}.

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

Constraints: EPSILON(1.0_*wp*) $\leq$ lin_feas_tol < 1.0.

nlin_feas_tol — real(kind=*wp*)

nlin_feas_tol defines the maximum acceptable *absolute* violation in nonlinear constraints at a 'feasible' point; i.e., a nonlinear constraint is considered satisfied if its violation does not exceed nlin_feas_tol.

nlin_feas_tol defines the largest constraint violation that is acceptable at an optimal point. Since nonlinear constraints are not generally satisfied until the final iterate, the value of nlin_feas_tol acts as a partial termination criterion for the iterative sequence generated by nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1 (see the description of optim_tol).

nlin_feas_tol should reflect the precision of the nonlinear constraints. For example, if the variables and the coefficients in the nonlinear constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify nlin_feas_tol as 10^{-6}.

Default: nlin_feas_tol = SQRT(EPSILON(1.0_*wp*)) if con_deriv = .true. (the default; see Section 3.2 of the appropriate procedure document), and $(\text{EPSILON(1.0\_}wp))^{0.33}$ otherwise.

Constraints: EPSILON(1.0_*wp*) $\leq$ nlin_feas_tol < 1.0.

crash_tol — real(kind=*wp*)

crash_tol is used in conjunction with the optional argument cold_start (see Section 3.2 of the appropriate procedure document) in order to select an initial working set.

If cold_start = .true. (the default), the initial working set will include (if possible) bounds or general inequality constraints that lie within crash_tol of their bounds. In particular, a constraint of the form $a_j^T x \geq l$ will be included in the working set if $|a_j^T x - l| \leq$ crash_tol $\times (1 + |l|)$.

Default: crash_tol = 0.01.

Constraints: 0.0 ≤ crash_tol ≤ 1.0.

fun_prec — real(kind=*wp*)

fun_prec defines ε_{R}, which is intended to be a measure of the accuracy with which the problem functions $F(x)$ and $c(x)$ can be computed.

The value of ε_{R} should reflect the relative precision of $1+|F(x)|$; i.e., ε_{R} acts as a relative precision when $|F|$ is large, and as an absolute precision when $|F|$ is small. For example, if $F(x)$ is typically of order 1000 and the first six significant digits are known to be correct, an appropriate value for ε_{R} would be 10^{-6}. In contrast, if $F(x)$ is typically of order 10^{-4} and the first six significant digits are known to be correct, an appropriate value for ε_{R} would be 10^{-10}.

The choice of ε_{R} can be quite complicated for badly scaled problems; see Chapter 8 of Gill *et al.* [10] for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However, when the accuracy of the computed function values is known to be significantly worse than full precision, the value of ε_{R} should be large enough so that nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1 will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

Default: fun_prec = (EPSILON(1.0_*wp*))$^{0.9}$.

Constraints: EPSILON(1.0_*wp*) ≤ fun_prec < 1.0.

optim_tol — real(kind=*wp*)

optim_tol specifies the accuracy to which you wish the final iterate to approximate a solution of the problem. Broadly speaking, optim_tol indicates the number of correct figures desired in the objective function at the solution. For example, if optim_tol is set to 10^{-6}, the final value of F should have approximately six correct figures whenever nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1 terminate successfully, i.e., if the iterative sequence of x-values is judged to have converged and the final point satisfies the first-order Kuhn–Tucker conditions (see Section 1 of the Mathematical Background section of this module document).

The sequence of iterates is considered to have converged at x if

$$\alpha||p|| \leq \sqrt{\texttt{optim\_tol}} \times (1 + ||x||),$$

where p is the search direction and α the step length from (7) (see Section 1 of the Mathematical Background section of this module document). An iterate is considered to satisfy the first-order conditions for a minimum if

$$||Z^T g_{\mathrm{FR}}|| \leq \sqrt{\texttt{optim\_tol}} \times (1 + \max\,(1 + |F(x)|, ||g_{\mathrm{FR}}||))$$

and

$$|res_j| \leq \texttt{nlin\_feas\_tol} \quad \text{for all} \quad j,$$

where $Z^T g_{\mathrm{FR}}$ is the projected gradient, g_{FR} is the gradient of $F(x)$ with respect to the free variables, and res_j is the violation of the jth active nonlinear constraint.

Default: optim_tol = (EPSILON(1.0_*wp*))$^{0.72}$.

Constraints: fun_prec ≤ optim_tol < 1.0.

linesearch_tol — real(kind=*wp*)

linesearch_tol controls the accuracy with which the step α taken during each iteration approximates a minimum of the merit function along the search direction (the smaller the value of linesearch_tol, the more accurate the linesearch). The default value (= 0.9) requests an inaccurate search, and is appropriate for most problems, particularly those with any nonlinear constraints.

If there are no nonlinear constraints, a more accurate search may be appropriate when it is desirable to reduce the number of major iterations (for example, if the objective function is cheap to evaluate, or if a substantial number of derivatives are unspecified).

Default: linesearch_tol = 0.9.

Constraints: 0.0 ≤ linesearch_tol < 1.0.

step_limit — real(kind=*wp*)

step_limit specifies the maximum change in variables at the first step of the linesearch. It is used to encourage evaluation of the problem functions at 'meaningful' points only, since in some cases, such as $F(x) = ae^{bx}$ or $F(x) = ax^b$, even a moderate change in the elements of x can lead to floating-point overflow. Given any major iterate x, the first point $\tilde{x}$ at which F and c are evaluated during the linesearch is restricted so that

$$||\tilde{x} - x||_2 \leq \texttt{step\_limit} \times (1 + ||x||_2).$$

The linesearch may go on and evaluate F and c at points further from x if this will result in a lower value of the merit function (indicated by L at the end of each line of intermediate printout produced by the major iterations; see Section 7.1 of the appropriate procedure document). If L is printed for most of the iterations, step_limit should be set to a larger value.

Wherever possible, upper and lower bounds on x should be used to prevent evaluation of the subfunctions at wild values. The default value (= 2.0) should not affect progress on well-behaved functions, but values such as 0.1 or 0.01 may be helpful when rapidly varying functions are present. If a small value of step_limit is selected, a good starting point may be required.

Default: step_limit = 2.0.

Constraints: step_limit > 0.0.

hessian — logical

hessian is used in conjunction with the optional argument r (see Section 3.2 of the appropriate procedure document). It controls the contents of the upper triangular Cholesky factor R (see Section 1 of the Mathematical Background section of this module document). Note that nag_con_nlin_lsq_sol and nag_con_nlin_lsq_sol_1 work exclusively with the *transformed and re-ordered* Hessian H_Q, and hence extra computation is required to form the Hessian H explicitly.

If hessian = .true. and r is present, the upper triangular Cholesky factor R of the approximate (untransformed) Hessian H is formed and stored in r.

If hessian = .false. and r is present, the upper triangular Cholesky factor R of the transformed and re-ordered Hessian H_Q is formed and stored in r.

The default value (= .true.) should be used if the optional argument cold_start (see Section 3.2 of the appropriate procedure document) will be .false. on the next call to nag_con_nlin_lsq_sol or nag_con_nlin_lsq_sol_1.

Default: hessian = .true..

jtj_init_hess — logical

jtj_init_hess controls the initial value of the upper triangular Cholesky factor R (see Section 1 of the Mathematical Background section of this module document).

If jtj_init_hess = .true., then R is initialized to $J^T J$ (which is often a good approximation to the objective Hessian matrix $\nabla^2 F(x)$; see the description of reset_freq).

If jtj_init_hess = .false., then R is initialized to the identity matrix.

Default: jtj_init_hess = .true..

Example 1: Nonlinear Least-squares Programming Problem (with bounds but no linear constraints)

This is Problem 57 from Hock and Schittkowski [11] and involves the minimization of the nonlinear sum of squares function

$$F(x) = \tfrac{1}{2}\sum_{i=1}^{44}(f_i(x) - y_i)^2,$$

where $f_i(x) = x_1 + (0.49 - x_1)e^{-x_2(a_i-8)}$ and

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a_i	8	8	10	10	10	10	12	12	12	12	14	14	14	16	16
y_i	0.49	0.49	0.48	0.47	0.48	0.47	0.46	0.46	0.45	0.43	0.45	0.43	0.43	0.44	0.43
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
a_i	16	18	18	20	20	20	22	22	22	24	24	24	26	26	26
y_i	0.43	0.46	0.45	0.42	0.42	0.43	0.41	0.41	0.40	0.42	0.40	0.40	0.41	0.40	0.41
i	31	32	33	34	35	36	37	38	39	40	41	42	43	44	
a_i	28	28	30	30	30	32	32	34	36	36	38	38	40	42	
y_i	0.41	0.40	0.40	0.40	0.38	0.41	0.40	0.40	0.41	0.38	0.40	0.40	0.39	0.39	

subject to the bounds

$$\begin{aligned} x_1 &\geq 0.4 \\ x_2 &\geq -4.0 \end{aligned}$$

and to the nonlinear constraint

$$0.49x_2 - x_1x_2 \geq 0.09.$$

The initial point, which is infeasible, is

$$x^{(0)} = (0.4,\ 0.0)^T.$$

The optimal solution (to five figures) is

$$x^* = (0.41995,\ 1.28485)^T,$$

and $F(x^*) = 0.01423$. The nonlinear constraint is active at the 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.

```
MODULE con_nlin_lsq_ex01_mod
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND, PRESENT
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  REAL (wp), PARAMETER :: pt49 = 0.49_wp

CONTAINS

  SUBROUTINE obj_fun(first_call,x,finish,f,f_jac)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
```

```
    ! .. Intrinsic Functions ..
    INTRINSIC EXP
    ! .. Parameters ..
    REAL (wp), PARAMETER :: eight = 8.0_wp
    REAL (wp), PARAMETER :: one = 1.0_wp
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    LOGICAL, INTENT (IN) :: first_call
    ! .. Array Arguments ..
    REAL (wp), INTENT (OUT) :: f(:)
    REAL (wp), OPTIONAL, INTENT (INOUT) :: f_jac(:,:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Local Arrays ..
    REAL (wp) :: a(44) = (/ 8.0_wp, 8.0_wp, 10.0_wp, 10.0_wp, 10.0_wp, &
     10.0_wp, 12.0_wp, 12.0_wp, 12.0_wp, 12.0_wp, 14.0_wp, 14.0_wp, &
     14.0_wp, 16.0_wp, 16.0_wp, 16.0_wp, 18.0_wp, 18.0_wp, 20.0_wp, &
     20.0_wp, 20.0_wp, 22.0_wp, 22.0_wp, 22.0_wp, 24.0_wp, 24.0_wp, &
     24.0_wp, 26.0_wp, 26.0_wp, 26.0_wp, 28.0_wp, 28.0_wp, 30.0_wp, &
     30.0_wp, 30.0_wp, 32.0_wp, 32.0_wp, 34.0_wp, 36.0_wp, 36.0_wp, &
     38.0_wp, 38.0_wp, 40.0_wp, 42.0_wp/)
    ! .. Executable Statements ..

    f = x(1) + (pt49-x(1))*EXP(-x(2)*(a-eight))

    IF (PRESENT(f_jac)) THEN
      f_jac(:,1) = one - EXP(-x(2)*(a-eight))
      f_jac(:,2) = -(pt49-x(1))*(a-eight)*EXP(-x(2)*(a-eight))
    END IF

  END SUBROUTINE obj_fun

  SUBROUTINE con_fun(first_call,x,finish,needc,con_f,con_jac)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Parameters ..
    REAL (wp), PARAMETER :: zero = 0.0_wp
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    LOGICAL, INTENT (IN) :: first_call
    ! .. Array Arguments ..
    INTEGER, INTENT (IN) :: needc(:)
    REAL (wp), INTENT (INOUT) :: con_f(:)
    REAL (wp), OPTIONAL, INTENT (INOUT) :: con_jac(:,:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Executable Statements ..

    IF (needc(1)>0) con_f(1) = -x(1)*x(2) + pt49*x(2)

    IF (PRESENT(con_jac)) THEN
      IF (needc(1)>0) THEN
        con_jac(1,1) = -x(2)
        con_jac(1,2) = -x(1) + pt49
      END IF
    END IF
  END SUBROUTINE con_fun

END MODULE con_nlin_lsq_ex01_mod

PROGRAM nag_con_nlin_lsq_ex01

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

```
      ! .. Use Statements ..
      USE nag_examples_io, ONLY : nag_std_in, nag_std_out
      USE nag_con_nlin_lsq, ONLY : nag_con_nlin_lsq_sol
      USE con_nlin_lsq_ex01_mod, ONLY : con_fun, obj_fun, wp
      ! .. Implicit None Statement ..
      IMPLICIT NONE
      ! .. Parameters ..
      INTEGER, PARAMETER :: m = 44, n = 2, num_nlin_con = 1
      ! .. Local Scalars ..
      REAL (wp) :: obj_f
      ! .. Local Arrays ..
      REAL (wp) :: f(m), nlin_lower(num_nlin_con), x(n), x_lower(n), y(m)
      ! .. Executable Statements ..
      WRITE (nag_std_out,*) &
       'Example Program Results for nag_con_nlin_lsq_ex01'

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

      ! Read in problem data
      READ (nag_std_in,*) x_lower
      READ (nag_std_in,*) nlin_lower
      READ (nag_std_in,*) x
      READ (nag_std_in,*) y

      ! Solve the problem

      CALL nag_con_nlin_lsq_sol(obj_fun,x,obj_f,f,num_nlin_con=num_nlin_con, &
       con_fun=con_fun,y=y,x_lower=x_lower,nlin_lower=nlin_lower)

    END PROGRAM nag_con_nlin_lsq_ex01
```

2 Program Data

```
Example Program Data for nag_con_nlin_lsq_ex01
 0.40  -4.00                                                : x_lower
 0.09                                                       : nlin_lower
 0.40   0.00                                                : x
 0.49   0.49   0.48   0.47   0.48   0.47   0.46   0.46   0.45
 0.43   0.45   0.43   0.43   0.44   0.43   0.43   0.46   0.45
 0.42   0.42   0.43   0.41   0.41   0.40   0.42   0.40   0.40
 0.41   0.40   0.41   0.41   0.40   0.40   0.40   0.38   0.41
 0.40   0.40   0.41   0.38   0.40   0.40   0.39   0.39         : y
```

3 Program Results

```
 Example Program Results for nag_con_nlin_lsq_ex01

 Parameters
 ----------

 list...................    .true.     lt80_char..............    .true.
 unit...................         6

 subfunctions...........        44     variables..............         2
 linear constraints.....         0     nonlinear constraints..         1

 inf_bound.............. 1.00E+20     cold_start.............    .true.
 inf_step............... 1.00E+20     eps (machine precision)  2.22E-16
 step_limit............. 2.00E+00     hessian................    .true.
```

```
major_print_level......        10     minor_print_level......         0
major_iter_lim.........        50     minor_iter_lim.........        50

nlin_feas_tol.......... 1.49E-08      optim_tol............. 5.36E-12
linesearch_tol......... 9.00E-01      fun_prec.............. 8.16E-15
crash_tol.............. 1.00E-02

jtj_init_hess..........   .true.      reset_freq............         2

f_deriv................   .true.      con_deriv.............    .true.
cheap_test.............   .true.      initial_x.............   .false.


Verification of the constraint gradients.
-----------------------------------------

The constraint Jacobian seems to be ok.

The largest relative error was    2.74E-08  in constraint    1



Verification of the objective gradients.
----------------------------------------

The objective Jacobian seems to be ok.

The largest relative error was    1.54E-07  in subfunction   44



 Maj  Mnr    Step Merit Function Norm Gz  Violtn Cond Hz
   0    1 0.0E+00   7.424030E+01 0.0E+00 9.0E-02 1.0E+00
   1    1 1.0E+00   2.169044E-02 6.7E-02 2.8E-17 1.0E+00
   2    1 1.0E+00   2.168663E-02 6.6E-02 2.9E-10 1.0E+00
   3    1 2.5E-01   1.853144E-02 4.6E-02 2.6E-04 1.0E+00
   4    1 1.0E+00   1.439518E-02 1.7E-03 3.1E-03 1.0E+00
   5    1 1.0E+00   1.422984E-02 2.4E-05 2.1E-05 1.0E+00
   6    1 1.0E+00   1.422984E-02 1.7E-05 1.5E-09 1.0E+00
   7    1 1.0E+00   1.422983E-02 5.3E-06 6.2E-10 1.0E+00
   8    1 1.0E+00   1.422983E-02 2.5E-10 1.2E-10 1.0E+00
   9    0 1.0E+00   1.422983E-02 1.5E-16 8.3E-17 1.0E+00

Exit from nag_con_nlin_lsq after     9 major iterations,
                                     9 minor iterations.


Varbl State     Value       Lower Bound   Upper Bound    Lagr Mult   Slack

V   1    FR   0.419953      0.400000          None           .      1.9953E-02
V   2    FR    1.28485      -4.00000          None           .       5.285


N Con State     Value       Lower Bound   Upper Bound    Lagr Mult   Slack

N   1    LL   9.000000E-02  9.000000E-02      None      3.3358E-02  8.3267E-17

Exit nag_con_nlin_lsq_sol - Optimal solution found.

Final objective value =   0.1422983E-01
```

Example 2: Nonlinear least-squares Programming Problem (with bounds and linear constraints)

This is Problem 23 from Hock and Schittkowski [11] and involves the minimization of the nonlinear sum of squares function

$$F(x) = \frac{1}{2}(x_1^2 + x_2^2)$$

subject to the bounds

$$\begin{array}{rcccl} -50 & \leq & x_1 & \leq & 50 \\ -50 & \leq & x_2 & \leq & 50 \end{array}$$

to the linear constraint

$$x_1 + x_2 \geq 1,$$

and to the nonlinear constraints

$$\begin{array}{l} x_1^2 + x_2^2 \geq 1, \\ 9x_1^2 + x_2^2 \geq 9, \\ x_1^2 - x_2 \geq 0, \\ x_2^2 - x_1 \geq 0. \end{array}$$

The initial point, which is infeasible, is

$$x^{(0)} = (3.0,\ 0.6)^T.$$

The optimal solution is

$$x^* = (1,\ 1)^T,$$

and $F(x^*) = 1$. Two nonlinear constraints are active at the 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.

```
MODULE con_nlin_lsq_ex06_mod
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND, PRESENT
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  REAL (wp), PARAMETER :: one = 1.0_wp

CONTAINS

  SUBROUTINE obj_fun(first_call,x,finish,f,f_jac,needf)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIZE
    ! .. Parameters ..
    REAL (wp), PARAMETER :: zero = 0.0_wp
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    LOGICAL, INTENT (IN) :: first_call
```

```
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: needf(:)
    REAL (wp), INTENT (OUT) :: f(:)
    REAL (wp), OPTIONAL, INTENT (INOUT) :: f_jac(:,:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Local Scalars ..
    INTEGER :: i
    ! .. Executable Statements ..

    IF (PRESENT(needf)) THEN
      DO i = 1, SIZE(f)
        IF (needf(i)>0) f(i) = x(i)
      END DO
    ELSE
      f = x
    END IF

    IF (PRESENT(f_jac)) THEN
      IF (first_call) THEN
        f_jac(1,1) = one
        f_jac(1,2) = zero
        f_jac(2,1) = zero
        f_jac(2,2) = one
      END IF
    END IF

  END SUBROUTINE obj_fun

  SUBROUTINE con_fun(first_call,x,finish,needc,con_f,con_jac)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Parameters ..
    REAL (wp), PARAMETER :: nine = 9.0_wp
    REAL (wp), PARAMETER :: two = 2.0_wp
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    LOGICAL, INTENT (IN) :: first_call
    ! .. Array Arguments ..
    INTEGER, INTENT (IN) :: needc(:)
    REAL (wp), INTENT (INOUT) :: con_f(:)
    REAL (wp), OPTIONAL, INTENT (INOUT) :: con_jac(:,:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Executable Statements ..

    IF (needc(1)>0) con_f(1) = x(1)**2 + x(2)**2
    IF (needc(2)>0) con_f(2) = nine*x(1)**2 + x(2)**2
    IF (needc(3)>0) con_f(3) = x(1)**2 - x(2)
    IF (needc(4)>0) con_f(4) = x(2)**2 - x(1)

    IF (PRESENT(con_jac)) THEN
      IF (needc(1)>0) con_jac(1,:) = two*x
      IF (needc(2)>0) THEN
        con_jac(2,1) = two*nine*x(1)
        con_jac(2,2) = two*x(2)
      END IF
      IF (needc(3)>0) THEN
        con_jac(3,1) = two*x(1)
        IF (first_call) con_jac(3,2) = -one
      END IF
      IF (needc(4)>0) THEN
        IF (first_call) con_jac(4,1) = -one
        con_jac(4,2) = two*x(2)
```

```
      END IF
    END IF
  END SUBROUTINE con_fun

END MODULE con_nlin_lsq_ex06_mod

PROGRAM nag_con_nlin_lsq_ex06

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_con_nlin_lsq, ONLY : nag_con_nlin_lsq_sol_1, &
   nag_con_nlin_lsq_cntrl_init
  USE nag_con_nlin_lsq_types, ONLY : nag_con_nlin_lsq_cntrl_wp => &
   nag_con_nlin_lsq_cntrl_dp
  USE con_nlin_lsq_ex06_mod, ONLY : con_fun, obj_fun, wp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Parameters ..
  INTEGER, PARAMETER :: m = 2, n = 2, nl = 1, num_nlin_con = 4
  ! .. Local Scalars ..
  INTEGER :: i
  REAL (wp) :: obj_f
  TYPE (nag_con_nlin_lsq_cntrl_wp) :: control
  ! .. Local Arrays ..
  REAL (wp) :: a(nl,n), f(m), lin_lower(nl), nlin_lower(num_nlin_con), &
   x(n), x_lower(n), x_upper(n)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) &
   'Example Program Results for nag_con_nlin_lsq_ex06'

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

  ! Read in problem data
  READ (nag_std_in,*) (a(i,:),i=1,nl)
  READ (nag_std_in,*) lin_lower
  READ (nag_std_in,*) x_lower
  READ (nag_std_in,*) x_upper
  READ (nag_std_in,*) nlin_lower
  READ (nag_std_in,*) x

  ! Initialize control structure and set required control parameters

  CALL nag_con_nlin_lsq_cntrl_init(control)

  control%major_iter_lim = 25
  control%minor_iter_lim = 10
  control%step_limit = 5.0_wp
  control%initial_x = .TRUE.

  ! Solve the problem

  CALL nag_con_nlin_lsq_sol_1(obj_fun,x,obj_f,f,num_nlin_con=num_nlin_con, &
   con_fun=con_fun,x_lower=x_lower,x_upper=x_upper,lin_lower=lin_lower, &
   a=a,nlin_lower=nlin_lower,control=control)

END PROGRAM nag_con_nlin_lsq_ex06
```

2 Program Data

```
Example Program Data for nag_con_nlin_lsq_ex06
   1.0    1.0                                                  : a
   1.0                                                         : lin_lower
 -50.0  -50.0                                                  : x_lower
  50.0   50.0                                                  : x_upper
   1.0    9.0    0.0    0.0                                    : nlin_lower
   3.0    0.6                                                  : x
```

3 Program Results

```
 Example Program Results for nag_con_nlin_lsq_ex06

 Parameters
 ----------

 list..................    .true.     lt80_char.............    .true.
 unit..................         6

 subfunctions..........         2     variables.............         2
 linear constraints.....        1     nonlinear constraints..        4

 inf_bound............. 1.00E+20     cold_start............    .true.
 inf_step.............. 1.00E+20     eps (machine precision)  2.22E-16
 step_limit............ 5.00E+00     hessian...............    .true.

 major_print_level......       10     minor_print_level......         0
 major_iter_lim.........       25     minor_iter_lim.........        10

 lin_feas_tol.......... 1.49E-08     crash_tol............. 1.00E-02
 nlin_feas_tol......... 1.49E-08     optim_tol............. 5.36E-12
 linesearch_tol........ 9.00E-01     fun_prec.............. 8.16E-15

 jtj_init_hess.........    .true.     reset_freq............         2

 f_deriv...............    .true.     con_deriv.............    .true.
 cheap_test............    .true.     initial_x.............    .true.


 Verification of the constraint gradients.
 -----------------------------------------

 The constraint Jacobian seems to be ok.

 The largest relative error was    3.25E-08  in constraint    2



 Verification of the objective gradients.
 ----------------------------------------

 The objective Jacobian seems to be ok.

 The largest relative error was    1.34E-10  in subfunction    1



  Maj  Mnr    Step Merit Function Norm Gz  Violtn Cond Hz
    0    2 0.0E+00   6.642581E+00 0.0E+00 8.8E+00 1.0E+00
    1    1 1.0E+00   2.882568E+00 2.2E+00 1.4E+00 1.0E+00
    2    1 1.0E+00   2.279671E+00 0.0E+00 1.4E+00 1.0E+00
```

```
   3    0 1.0E+00   1.072307E+00 0.0E+00 2.6E-01 1.0E+00
   4    0 1.0E+00   1.000912E+00 0.0E+00 2.4E-02 1.0E+00
   5    0 1.0E+00   1.000000E+00 0.0E+00 3.4E-04 1.0E+00
   6    0 1.0E+00   1.000000E+00 0.0E+00 7.6E-08 1.0E+00
   7    0 1.0E+00   1.000000E+00 0.0E+00 4.0E-15 1.0E+00

Exit from nag_con_nlin_lsq after     7 major iterations,
                                     4 minor iterations.


Varbl State     Value       Lower Bound   Upper Bound    Lagr Mult   Slack

V   1    FR    1.00000      -50.0000       50.0000           .       49.00
V   2    FR    1.00000      -50.0000       50.0000           .       49.00


L Con State     Value       Lower Bound   Upper Bound    Lagr Mult   Slack

L   1    FR    2.00000       1.00000          None           .       1.000


N Con State     Value       Lower Bound   Upper Bound    Lagr Mult   Slack

N   1    FR    2.00000       1.00000          None           .       1.000
N   2    FR    10.0000       9.00000          None           .       1.000
N   3    LL   3.108624E-15      .             None       1.000      3.1086E-15
N   4    LL   2.442491E-15      .             None       1.000      2.4425E-15

Exit nag_con_nlin_lsq_sol_1 - Optimal solution found.

Final objective value =    1.000000
```

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_con_nlin_lsq_ex02`

Solves a nonlinear least-squares programming problem with bounds and linear constraints.

`nag_con_nlin_lsq_ex03`

Solves a nonlinear least-squares programming problem in which there are no bounds or linear constraints.

`nag_con_nlin_lsq_ex04`

Solves an unconstrained non-linear least-squares problem.

`nag_con_nlin_lsq_ex05`

Solves a nonlinear least-squares programming problem with bounds but no linear constraints.

`nag_con_nlin_lsq_ex07`

Solves a nonlinear least-squares programming problem in which there are no bounds or linear constraints.

`nag_con_nlin_lsq_ex08`

Solves an unconstrained non-linear least-squares problem.

Mathematical Background

1 Overview

`nag_con_nlin_lsq_sol` and `nag_con_nlin_lsq_sol_1` are based on the procedure NPSOL described in Gill *et al.* [6], which implements a sequential quadratic programming (SQP) method. For an overview of SQP methods, see for example, Fletcher [4], Gill *et al.* [10] and Powell [13].

At a solution of

$$\underset{x \in R^n}{\text{minimize}} \; F(x) = \tfrac{1}{2} \sum_{i=1}^{m} (f_i(x) - y_i)^2 \;\; \text{subject to} \;\; l \le \left\{ \begin{array}{c} x \\ A_{\text{L}} x \\ c(x) \end{array} \right\} \le u, \tag{5}$$

some of the constraints will be *active*, i.e., satisfied exactly. An active simple bound constraint implies that the corresponding variable is *fixed* at its bound, and hence the variables are partitioned into *fixed* and *free* variables. Let C denote the m by n matrix of gradients of the active general linear and nonlinear constraints. The number of fixed variables will be denoted by n_{FX}, with n_{FR} ($n_{\text{FR}} = n - n_{\text{FX}}$) the number of free variables. The subscripts 'FX' and 'FR' on a vector or matrix will denote the vector or matrix composed of the elements corresponding to fixed or free variables.

A point x is a *a first-order Kuhn–Tucker point* for (5) (see Powell [12]) if the following conditions hold:

(a) x is feasible;

(b) there exist vectors ξ and λ (*the Lagrange multiplier vectors for the bound and general constraints*) such that

$$g = C^T \lambda + \xi, \tag{6}$$

where g is the gradient of F evaluated at x, and $\xi_j = 0$ if the jth variable is free.

(c) The Lagrange multiplier corresponding to an inequality constraint active at its lower bound must be non-negative, and non-positive for an inequality constraint active at its upper bound.

Let Z denote a matrix whose columns form a basis for the set of vectors orthogonal to the rows of C_{FR}; i.e., $C_{\text{FR}} Z = 0$. An equivalent statement of the condition (6) in terms of Z is

$$Z^T g_{\text{FR}} = 0.$$

The vector $Z^T g_{\text{FR}}$ is termed the *projected gradient* of F at x. Certain additional conditions must be satisfied in order for a first-order Kuhn–Tucker point to be a solution of (5) (see Powell [12]).

The basic structure of `nag_con_nlin_lsq_sol` and `nag_con_nlin_lsq_sol_1` involves *major* and *minor* iterations. The major iterations generate a sequence of iterates $\{x_k\}$ that converge to x^*, a first-order Kuhn–Tucker point of (5). At a typical major iteration, the new iterate $\bar{x}$ is defined by

$$\bar{x} = x + \alpha p, \tag{7}$$

where x is the current iterate, the non-negative scalar α is the *step length*, and p is the *search direction*. (For simplicity, we shall always consider a typical iteration and avoid reference to the index of the iteration.) Also associated with each major iteration are estimates of the Lagrange multipliers and a prediction of the active set.

The search direction p in (7) is the solution of a quadratic programming subproblem of the form

$$\underset{p}{\text{minimize}} \; g^T p + \tfrac{1}{2} p^T H p \;\; \text{subject to} \;\; \bar{l} \le \left\{ \begin{array}{c} p \\ A_{\text{L}} p \\ A_{\text{N}} p \end{array} \right\} \le \bar{u}, \tag{8}$$

where g is the gradient of F at x, the matrix H is a positive definite quasi-Newton approximation to the Hessian of the Lagrangian function (see Section 4), and A_{N} is the Jacobian matrix of c evaluated at x. (Finite difference estimates may be used for g and $A_{\rm N}$; see the optional parameters `f_deriv` and `con_deriv` in Section 3.2 of the appropriate procedure document.) Let l in (5) be partitioned into three sections: $l_{\rm B}$, $l_{\rm L}$ and $l_{\rm N}$, corresponding to the bound, linear and nonlinear constraints. The vector $\bar{l}$ in (8) is similarly partitioned, and is defined as

$$\bar{l}_{\rm B} = l_{\rm B} - x, \;\; \bar{l}_{\rm L} = l_{\rm L} - A_{\rm L}x, \quad \text{and } \bar{l}_{\rm N} = l_{\rm N} - c,$$

where c is the vector of nonlinear constraints evaluated at x. The vector $\bar{u}$ is defined in an analogous fashion.

The estimated Lagrange multipliers at each major iteration are the Lagrange multipliers from the subproblem (8) (and similarly for the predicted active set). (The numbers of bounds, general linear and nonlinear constraints in the QP active set are the quantities `Bnd`, `Lin` and `Nln` in the printed output.) The subproblem (8) is solved using procedures derived from LSSOL described in Gill *et al.* [5]. Since solving a quadratic program is itself an iterative procedure, the *minor* iterations of `nag_con_nlin_lsq_sol` and `nag_con_nlin_lsq_sol_1` are the iterations within this process. (More details about solving the subproblem are given in Section 2.)

Certain matrices associated with the QP subproblem are relevant in the major iterations. Let the subscripts 'FX' and 'FR' refer to the *predicted* fixed and free variables, and let C denote the m by n matrix of gradients of the general linear and nonlinear constraints in the predicted active set. First, we have available the TQ factorization of $C_{\rm FR}$:

$$C_{\rm FR}Q_{\rm FR} = (0 \;\; T), \tag{9}$$

where T is a non-singular m by m reverse-triangular matrix (i.e., $t_{ij} = 0$ if $i + j < m$), and the non-singular $n_{\rm FR}$ by $n_{\rm FR}$ matrix $Q_{\rm FR}$ is the product of orthogonal transformations (see Gill *et al.* [7]). Second, we have the upper triangular Cholesky factor R of the *transformed and re-ordered* Hessian matrix

$$R^TR = H_Q \equiv Q^T\tilde{H}Q, \tag{10}$$

where $\tilde{H}$ is the Hessian H with rows and columns permuted so that the free variables are first, and Q is the n by n matrix

$$Q = \begin{pmatrix} Q_{\rm FR} & \\ & I_{\rm FX} \end{pmatrix}, \tag{11}$$

with $I_{\rm FX}$ the identity matrix of order $n_{\rm FX}$. If the columns of $Q_{\rm FR}$ are partitioned so that

$$Q_{\rm FR} = (Z \;\; Y),$$

the n_Z ($n_Z \equiv n_{\rm FR} - m$) columns of Z form a basis for the null space of $C_{\rm FR}$. The matrix Z is used to compute the projected gradient $Z^Tg_{\rm FR}$ at the current iterate. (The values of n_Z and the norms of $g_{\rm FR}$ and $Z^Tg_{\rm FR}$ are the quantities `Nz`, `Norm Gf` and `Norm Gz` in the printed output.)

A theoretical characteristic of SQP methods is that the predicted active set from the QP subproblem (8) is identical to the correct active set in a neighbourhood of x^*. This feature is exploited by using the QP active set from the previous iteration as a prediction of the active set for the next QP subproblem, which leads in practice to optimality of the subproblems in only one iteration as the solution is approached. Separate treatment of bound and linear constraints also saves computation in factorizing $C_{\rm FR}$ and H_Q.

Once p has been computed, the major iteration proceeds by determining a step length α that produces a 'sufficient decrease' in an augmented Lagrangian *merit function* (see Section 3). Finally, the approximation to the transformed Hessian matrix H_Q is updated using a modified BFGS (Broyden–Fletcher–Goldfarb–Shanno) quasi-Newton update (see Section 4) to incorporate new curvature information obtained in the move from x to $\bar{x}$.

Starting from the user-provided initial point, an iterative procedure is executed to find a point that is feasible with respect to the bounds and linear constraints (using the tolerance specified by `control%lin_feas_tol`; see the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). If no feasible point exists for the bound and linear constraints, (5) has no solution and termination occurs with `error%code` = 202 (no feasible point was found for the linear constraints and bounds). Otherwise, the problem functions will thereafter be evaluated only at points that are feasible with respect to the bounds and linear constraints. The only exception involves variables whose bounds differ by an amount comparable to the finite difference interval (see the discussion of `control%diff_int` in the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). In contrast to the bounds and linear constraints, it must be emphasised that *the nonlinear constraints will not generally be satisfied until an optimal point* is reached.

Facilities are provided to check whether the user-provided Jacobians appear to be correct (see the discussion of `control%cheap_test`, `control%obj_verify` and `control%con_verify` in the type definition for `nag_con_nlin_lsq_cntrl_`*wp*). In general, the check is provided at the first point that is feasible with respect to the linear constraints and bounds. However, you may request that the check be performed at the initial point.

In summary, the method of `nag_con_nlin_lsq_sol` and `nag_con_nlin_lsq_sol_1` first determines a point that satisfies the bound and linear constraints. Thereafter, each iteration includes:

(a) the solution of a quadratic programming subproblem (see Section 2);

(b) a linesearch with an augmented Lagrangian merit function (see Section 3); and

(c) a quasi-Newton update of the approximate Hessian of the Lagrangian merit function (see Section 4).

2 Solution of the Quadratic Programming Subproblem

The search direction p is obtained by solving (8) using procedures derived from LSSOL described in Gill *et al.* [5], which was specifically designed to be used within an SQP algorithm for nonlinear programming.

The method used by `nag_con_nlin_lsq_sol` and `nag_con_nlin_lsq_sol_1` is a two-phase (primal) quadratic programming method. The two phases of the method are: finding an initial feasible point by minimizing the sum of infeasibilities (the *feasibility phase*), and minimizing the quadratic objective function within the feasible region (the *optimality phase*). The computations in both phases are performed by the same procedures. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function.

In general, a quadratic program must be solved by iteration. Let p denote the current estimate of the solution of (8); the new iterate $\bar{p}$ is defined by

$$\bar{p} = p + \sigma d, \tag{12}$$

where, as in (7), σ is a non-negative step length and d is a search direction.

At the beginning of each iteration of the procedure , a *working* set is defined of constraints (general and bound) that are satisfied exactly. The vector d is then constructed so that the values of the constraints in the working set remain *unaltered* for any move along d. For a bound constraint in the working set, this property is achieved by setting the corresponding element of d to zero, i.e., by fixing the variable at its bound. As before, the subscripts 'FX' and 'FR' denote selection of the elements associated with the fixed and free variables.

Let C denote the sub-matrix of rows of

$$\begin{pmatrix} A_{\mathrm{L}} \\ A_{\mathrm{N}} \end{pmatrix}$$

corresponding to general constraints in the working set. The general constraints in the working set will remain unaltered if

$$C_{\mathrm{FR}} d_{\mathrm{FR}} = 0, \tag{13}$$

which is equivalent to defining d_{FR} as

$$d_{\mathrm{FR}} = Z d_Z \tag{14}$$

for some vector d_Z, where Z is the matrix associated with the TQ factorization (9) of C_{FR}.

The definition of d_Z in (14) depends on whether the current p is feasible. If not, d_Z is zero except for an element γ in the jth position, where j and γ are chosen so that the sum of infeasibilities is decreasing along d. (For further details, see Gill *et al.* [5].) In the feasible case, d_Z satisfies the equations

$$R_Z^T R_Z d_Z = -Z^T q_{\text{FR}}, \tag{15}$$

where R_Z is the Cholesky factor of $Z^T H_{\text{FR}} Z$ and q is the gradient of the quadratic objective function ($q = g + Hp$). (The vector $Z^T q_{\text{FR}}$ is the projected gradient of the QP.) With (15), $p + d$ is the minimizer of the quadratic objective function subject to treating the constraints in the working set as equalities.

If the QP projected gradient is zero, the current point is a constrained stationary point in the sub-space defined by the working set. During the feasibility phase, the projected gradient will usually be zero only at a vertex (although it may vanish at non-vertices in the presence of constraint dependencies). During the optimality phase, a zero projected gradient implies that p minimizes the quadratic objective function when the constraints in the working set are treated as equalities. In either case, Lagrange multipliers are computed. Given a positive constant δ of the order of EPSILON(1.0_*wp*), the Lagrange multiplier μ_j corresponding to an inequality constraint in the working set at its upper bound is said to be *optimal* if $\mu_j \leq \delta$ when the jth constraint is at its *upper bound*, or if $\mu_j \geq -\delta$ when the associated constraint is at its *lower bound*. If any multiplier is non-optimal, the current objective function (either the true objective or the sum of infeasibilities) can be reduced by deleting the corresponding constraint from the working set.

If optimal multipliers occur during the feasibility phase and the sum of infeasibilities is non-zero, no feasible point exists. The QP algorithm will then continue iterating to determine the minimum sum of infeasibilities. At this point, the Lagrange multiplier μ_j will satisfy $-(1+\delta) \leq \mu_j \leq \delta$ for an inequality constraint at its upper bound, and $-\delta \leq \mu_j \leq (1+\delta)$ for an inequality at its lower bound. The Lagrange multiplier for an equality constraint will satisfy $|\,\mu_j\,| \leq 1 + \delta$.

The choice of step length σ in the QP iteration (12) is based on remaining feasible with respect to the satisfied constraints. During the optimality phase, if $p + d$ is feasible, σ will be taken as unity. (In this case, the projected gradient at $\bar{p}$ will be zero.) Otherwise, σ is set to σ_{M}, the step to the 'nearest' constraint, which is added to the working set at the next iteration.

Each change in the working set leads to a simple change to C_{FR}: if the status of a general constraint changes, a *row* of C_{FR} is altered; if a bound constraint enters or leaves the working set, a *column* of C_{FR} changes. Explicit representations are recurred of the matrices T, Q_{FR} and R, and of the vectors $Q^T q$ and $Q^T g$.

3 The Merit Function

After computing the search direction as described in Section 2, each major iteration proceeds by determining a step length α in (7) that produces a 'sufficient decrease' in the augmented Lagrangian merit function

$$L(x, \lambda, s) = F(x) - \sum_i \lambda_i (c_i(x) - s_i) + \tfrac{1}{2} \sum_i \rho_i (c_i(x) - s_i)^2, \tag{16}$$

where x, λ and s vary during the *linesearch*. The summation terms in (16) involve only the *nonlinear* constraints. The vector λ is an estimate of the Lagrange multipliers for the nonlinear constraints of (5). The non-negative *slack variables* $\{s_i\}$ allow nonlinear inequality constraints to be treated without introducing discontinuities. The solution of the QP subproblem (8) provides a vector triple that serves as a direction of search for the three sets of variables. The non-negative vector ρ of *penalty parameters* is initialized to zero at the beginning of the first major iteration. Thereafter, selected elements are increased whenever necessary to ensure descent for the merit function. Thus, the sequence of norms of ρ (the quantity Penalty in the printed output) is generally non-decreasing, although each ρ_i may be reduced a limited number of times.

The merit function (16) and its global convergence properties are described in Gill *et al.* [9].

4 The Quasi-Newton Update

The matrix H in (8) is a *positive definite quasi-Newton* approximation to the Hessian of the Lagrangian function. (For a review of quasi-Newton methods, see Dennis Jr and Schnabel [3].) At the end of each major iteration, a new Hessian approximation $\bar{H}$ is defined as a rank-two modification of H (using the BFGS quasi-Newton update):

$$\bar{H} = H - \frac{1}{s^T H s} H s s^T H + \frac{1}{y^T s} y y^T, \tag{17}$$

where $s = \bar{x} - x$ (the change in x).

Note that H is required to be positive definite. If H is positive definite, $\bar{H}$ defined by (17) will be positive definite if and only if $y^T s$ is positive (see Dennis Jr and Moré [1]). Ideally, y in (17) would be taken as y_L, the change in gradient of the Lagrangian function

$$y_L = \bar{g} - \bar{A}_{\rm N}^T \mu_{\rm N} - g + A_{\rm N}^T \mu_{\rm N}, \tag{18}$$

where $\mu_{\rm N}$ denotes the QP multipliers associated with the nonlinear constraints of the original problem. If $y_L^T s$ is not sufficiently positive, an attempt is made to perform the update with a vector y of the form

$$y = y_L + \sum_i \omega_i (a_i(\bar{x}) c_i(\bar{x}) - a_i(x) c_i(x)),$$

where $\omega_i \geq 0$. If no such vector can be found, the update is performed with a scaled y_L; in this case an `M` is printed to indicate that the update was modified.

Rather than modifying H itself, the Cholesky factor of the *transformed Hessian* H_Q (10) is updated, where Q is the matrix from (9) associated with the active set of the QP subproblem. The update (17) is equivalent to the following update to H_Q:

$$\bar{H}_Q = H_Q - \frac{1}{s_Q^T H_Q s_Q} H_Q s_Q s_Q^T H_Q + \frac{1}{y_Q^T s_Q} y_Q y_Q^T, \tag{19}$$

where $y_Q = Q^T y$, and $s_Q = Q^T s$. This update may be expressed as a *rank-one* update to R (see Dennis Jr and Schnabel [2]).

References

[1] Dennis J E Jr and Moré J J (1977) Quasi-Newton methods, motivation and theory *SIAM Rev.* **19** 46–89

[2] Dennis J E Jr and Schnabel R B (1981) A new derivation of symmetric positive-definite secant updates *Nonlinear Programming* (ed O L Mangasarian, R R Meyer and S M Robinson) **4** Academic Press 167–199

[3] Dennis J E Jr and Schnabel R B (1983) *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* Prentice-Hall

[4] Fletcher R (1987) *Practical Methods of Optimization* Wiley (2nd Edition)

[5] Gill P E, Hammarling S, Murray W, Saunders M A and Wright M H (1986) User's guide for LSSOL (Version 1.0) *Report SOL 86-1* Department of Operations Research, Stanford University

[6] Gill P E, Murray W, Saunders M A and Wright M H (1986) User's guide for NPSOL (Version 4.0) *Report SOL 86-2* Department of Operations Research, Stanford University

[7] Gill P E, Murray W, Saunders M A and Wright M H (1984) User's guide for SOL/QPSOL version 3.2 *Report SOL 84–5* Department of Operations Research, Stanford University

[8] Gill P E, Murray W, Saunders M A and Wright M H (1984) Procedures for optimization problems with a mixture of bounds and general linear constraints *ACM Trans. Math. Software* **10** 282–298

[9] Gill P E, Murray W, Saunders M A and Wright M H (1986) Some theoretical properties of an augmented Lagrangian merit function *Report SOL 86–6R* Department of Operations Research, Stanford University

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

[11] Hock W and Schittkowski K (1981) *Test Examples for Nonlinear Programming Codes. Lecture Notes in Economics and Mathematical Systems* **187** Springer-Verlag

[12] Powell M J D (1974) Introduction to constrained optimization *Numerical Methods for Constrained Optimization* (ed P E Gill and W Murray) Academic Press 1–28

[13] Powell M J D (1983) Variable metric methods in constrained optimization *Mathematical Programming: The State of the Art* (ed A Bachem, M Grötschel and B Korte) Springer-Verlag 288–311