Module 9.6: nag_nlp_sparse

Sparse Nonlinear Programming

`nag_nlp_sparse` contains a procedure for solving sparse NLP problems.

Contents

Introduction 9.6.3

Procedures

`nag_nlp_sparse_sol` 9.6.5
Solves a sparse nonlinear programming problem

`nag_nlp_sparse_cntrl_init` 9.6.29
Initialization procedure for `nag_nlp_sparse_cntrl_`*wp*

Derived Types

`nag_nlp_sparse_cntrl_`*wp* 9.6.31
Control parameters for `nag_nlp_sparse_sol`

Examples

Example 1: Nonlinear Programming Problem (with bounds and linear constraints) 9.6.47
Example 2: Nonlinear Programming Problem (with bounds but no general constraints) ... 9.6.53

Additional Examples 9.6.57

Mathematical Background 9.6.59

References 9.6.62

Introduction

This module contains two procedures and a derived type as follows:

- `nag_nlp_sparse_sol` computes a constrained minimum (or maximum) of an arbitrary smooth 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, at non-trivial expense.

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

- `nag_nlp_sparse_cntrl_`*wp* may be used to supply optional parameters to `nag_nlp_sparse_sol`.

Procedure: nag_nlp_sparse_sol

1 Description

nag_nlp_sparse_sol is designed to solve a class of nonlinear programming problems that are assumed to be stated in the following general form:

$$\underset{x \in R^n}{\text{minimize}} \; f(x) \;\; \text{subject to} \;\; l \le \left\{ \begin{array}{c} x \\ F(x) \\ Gx \end{array} \right\} \le u, \tag{1}$$

where $x = (x_1, x_2, \ldots, x_n)^T$ is a set of variables, $f(x)$ is a smooth scalar objective function, l and u are constant lower and upper bounds, $F(x)$ is a vector of smooth nonlinear constraint functions $\{F_i(x)\}$ and G is a *sparse* matrix.

If there are no nonlinear constraints in (1) and F is linear or quadratic, then nag_qp_sol in the module nag_qp (9.1) will generally be more efficient if G is a *dense* matrix. If the problem is dense and does have nonlinear constraints, then either nag_nlp_sol or nag_con_nlin_lsq_sol_1 (as appropriate) should be used instead.

The constraints involving F and Gx are called the *general constraints*. Note that upper and lower bounds are specified for all variables and constraints. This form allows full generality in specifying various types of constraint. In particular, the jth constraint can be defined as an *equality* by setting $l_j = u_j$. If certain bounds are not present, the associated elements of l or u can be set to special values that will be treated as $-\infty$ or $+\infty$.

The procedure converts the upper and lower bounds on the m elements of F and Gx to equalities by introducing a set of *slack variables* s, where $s = (s_1, s_2, \ldots, s_m)^T$. For example, the linear constraint $5 \le 2x_1 + 3x_2 \le +\infty$ is replaced by $2x_1 + 3x_2 - s_1 = 0$, together with the bounded slack $5 \le s_1 \le +\infty$. The problem defined by (1) can therefore be re-written in the following equivalent form:

$$\underset{x \in R^n, s \in R^m}{\text{minimize}} \; f(x) \;\; \text{subject to} \;\; \left\{ \begin{array}{c} F(x) \\ Gx \end{array} \right\} - s = 0, \;\; l \le \left\{ \begin{array}{c} x \\ s \end{array} \right\} \le u. \tag{2}$$

Since the slack variables s are subject to the same upper and lower bounds as the elements of F and Gx, the bounds on F and Gx can simply be thought of as bounds on the combined vector (x, s). The elements of x and s are partitioned into *basic*, *nonbasic* and *superbasic variables* defined as follows:

A basic variable (x_j say) is the jth variable associated with the jth column of the associated basis matrix B.

A nonbasic variable is a variable that is not basic.

A superbasic variable is a nonbasic variable which is not at one of its bounds.

For example, in the simplex method (see Gill *et al.* [5]) the elements of x and s can be partitioned at each vertex into a set of m basic variables (all non-negative) and a set of $(n - m)$ nonbasic variables (all zero). This is equivalent to partitioning the columns of the constraint matrix as $(B \mid N)$, where B contains the m columns that correspond to the basic variables and N contains the $(n - m)$ columns that correspond to the nonbasic variables.

In general, the objective and constraint functions are *structured* in the sense that they are formed from sums of linear and nonlinear functions. This structure can be exploited by the procedure during the solution process as follows.

Consider the following nonlinear optimization problem with four variables (u, v, z, w):

$$\underset{u,v,z,w}{\text{minimize}} \; (u + v + z)^2 + 3z + 5w$$

subject to the constraints

$$\begin{array}{lllllll} u^2 & + & v^2 & + & z & & = 2 \\ u^4 & + & v^4 & & & + \; w & = 4 \\ 2u & + & 4v & & & & \ge 0 \end{array}$$

and to the bounds

$$\begin{array}{rcl} z & \geq & 0 \\ w & \geq & 0. \end{array}$$

This problem has several characteristics that can be exploited by the procedure:

The objective function is nonlinear. It is the sum of a *nonlinear* function of the variables (u,v,z) and a *linear* function of the variables (z,w).

The first two constraints are nonlinear. The third is linear.

Each nonlinear constraint function is the sum of a *nonlinear* function of the variables (u,v) and a *linear* function of the variables (z,w).

The nonlinear terms are defined by the procedures `obj_fun` and `con_fun` (see Section 3.2), which involve only the appropriate subset of variables.

For the objective, we define the function $f(u,v,z) = (u+v+z)^2$ to include only the nonlinear part of the objective. The three variables (u,v,z) associated with this function are known as the *nonlinear objective variables*. The number of them is given by `num_nlin_obj_var` (see Section 3.2), and they are the only variables needed in `obj_fun`. The linear part $3z+5w$ of the objective is stored in row `obj_row` (see Section 3.2) of the (constraint) Jacobian matrix A (see below).

Thus, if x' and y' denote the nonlinear and linear objective variables, respectively, the objective may be re-written in the form

$$f(x') + c^T x' + d^T y',$$

where $f(x')$ is the nonlinear part of the objective; and c and d are constant vectors that form a row of A. In this example, $x' = (u,v,z)$ and $y' = w$.

Similarly for the constraints, we define a vector function $F(u,v)$ to include just the nonlinear terms. In this example, $F_1(u,v) = u^2 + v^2$ and $F_2(u,v) = u^4 + v^4$, where the two variables (u,v) are known as the *nonlinear Jacobian variables*. The number of them is given by `num_nlin_jac_var` (see Section 3.2), and they are the only variables needed in `con_fun`. Thus, if x'' and y'' denote the nonlinear and linear Jacobian variables, respectively, the constraint functions and the linear part of the objective have the form

$$\left(\begin{array}{cc} F(x'') & + A_2 y'' \\ A_3 x'' & + A_4 y'' \end{array} \right), \tag{3}$$

where $x'' = (u,v)$ and $y'' = (z,w)$ in this example. This ensures that the Jacobian is of the form

$$A = \left(\begin{array}{cc} J(x'') & A_2 \\ A_3 & A_4 \end{array} \right),$$

where $J(x'') = \frac{\partial F(x'')}{\partial x}$. Note that $J(x'')$ *always* appears in the *top left-hand corner* of A.

The inequalities $l_1 \leq F(x'') + A_2 y'' \leq u_1$ and $l_2 \leq A_3 x'' + A_4 y'' \leq u_2$ implied by the constraint functions in (3) are known as the *nonlinear* and *linear* constraints, respectively. The nonlinear constraint vector $F(x'')$ in (3) and (optionally) its partial derivative matrix $J(x'')$ are set in `con_fun`. The matrices A_2, A_3 and A_4 contain any (constant) linear terms. Along with the sparsity pattern of $J(x'')$ they are stored in the arrays `a`, `row_index` and `col_ptr` (see Section 3.2).

In general, the vectors x' and x'' have different dimensions, but they *always overlap*, in the sense that the shorter vector is always the beginning of the other. In the above example, the nonlinear Jacobian variables (u,v) are an ordered subset of the nonlinear objective variables (u,v,w). In other cases it could be the other way round (whichever is the most convenient), but the first way keeps $J(x'')$ as small as possible.

Note that the nonlinear objective function $f(x')$ may involve either a subset or superset of the variables appearing in the nonlinear constraint functions $F(x'')$. Thus, `num_nlin_obj_var` $\leq$ `num_nlin_jac_var` (or vice-versa). Sometimes the objective and constraints really involve *disjoint sets of nonlinear variables*. In such cases the variables should be ordered so that `num_nlin_obj_var` $>$ `num_nlin_jac_var` and

$x' = (x'', x''')$, where the objective is nonlinear in just the last vector x'''. The first `num_nlin_jac_var` elements of the gradient array `obj_grad` should also be set to zero in `obj_fun`.

You must supply an initial estimate of the solution to (1), together with a procedure `obj_fun` (if $n_1' > 0$) that defines $f(x')$ and/or (if $n_N > 0$) a procedure `con_fun` which defines $F(x'')$. On every call, these procedures must return values of the nonlinear part of the objective function or the nonlinear constraints, and as many partial derivatives as possible. For maximum reliability, you should provide all partial derivatives (see Chapter 8 of Gill *et al.* [5] for a detailed discussion). Any derivatives which are not provided are approximated by finite differences, at non-trivial expense.

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_nlp_sparse_cntrl_`*wp*.

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

Note: all the input arguments needed to specify the problem to be solved by this procedure are optional. Hence, at least one of the following optional input arguments *must* be present in every call statement: `con_fun`, `obj_fun` or `a` (together with `row_index` and `col_ptr`).

2 Usage

```
USE nag_nlp_sparse

CALL nag_nlp_sparse_sol(x, s, obj_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 '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

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

$n \geq 1$ — the number of variables

$m \geq 1$ — the number of slacks (or general constraints)

$n_Z \leq n \times m$ — the number of non-zeros

3.1 Mandatory Arguments

x(n) — real(kind=*wp*), intent(inout)

Input: the initial values of the variables x. (See also the description for `x_state` in Section 3.2.)

Output: the final values of the variables x.

s(m) — real(kind=*wp*), intent(inout)

Input: if `cold_start` = `.true.` (the default; see Section 3.2), `s` need not be initialized.

If `cold_start` = `.false.`, `s` must contain the initial values of the slacks s.

Output: the final values of the slacks s.

obj_f — real(kind=*wp*), intent(out)

Output: the value of the objective function $f(x)$.

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.

num_nlin_obj_var — integer, intent(in), optional

Input: the number of nonlinear objective variables, n_1'. If the objective function is nonlinear, the leading n_1' columns of A belong to the nonlinear objective variables. (See also the description for `num_nlin_jac_var` below.)

Constraints:

`num_nlin_obj_var` must be present if `obj_fun` is present;

$0 \leq$ `num_nlin_obj_var` $\leq n$.

Default: `num_nlin_obj_var` $= 0$.

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;

$0 \leq$ `num_nlin_con` $\leq m$.

Default: `num_nlin_con` $= 0$.

num_nlin_jac_var — integer, intent(in), optional

Input: the number of nonlinear Jacobian variables, n_1''. If there are any nonlinear constraints, the leading n_1'' columns of A belong to the nonlinear Jacobian variables. If $n_1' > 0$ and $n_1'' > 0$, the nonlinear objective and Jacobian variables overlap. The total number of nonlinear variables is given by $\bar{n} = \max(n_1', n_1'')$.

Constraints:

`num_nlin_jac_var` must be present if `num_nlin_con` is present;

`num_nlin_jac_var` $= 0$ when `num_nlin_con` $= 0$, and $1 \leq$ `num_nlin_jac_var` $\leq n$ otherwise.

Default: `num_nlin_jac_var` $= 0$.

obj_fun — subroutine, optional

The procedure `obj_fun`, supplied by the user, must calculate the nonlinear part of the objective function $f(x)$ and (optionally) its gradient $g(x) = \partial f/\partial x$ for a specified n_1' $(\leq n)$ element vector x.

Its specification is:

```
subroutine obj_fun(first_call, final_call, x, continue, finish, obj_f, &
                   obj_grad, i_comm, r_comm)

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

Input: `first_call` will be `.true.` when `nag_nlp_sparse_sol` 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.

`logical, intent(in) :: final_call`

Input: `final_call` will be `.true.` when `nag_nlp_sparse_sol` calls `obj_fun` for the final time, and `.false.` for all previous calls. It allows you to perform some additional computation on the final solution.

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

Shape: `x` has shape (n_1').

Input: the vector x of nonlinear variables at which the nonlinear part of the objective function and (optionally) elements of its gradient are to be evaluated.

`logical, intent(inout) :: continue`

Input: `continue` will always be `.true.` on entry.

Output: if the nonlinear part of the objective function cannot be calculated at the current x, you should set `continue` to `.false.`. Unless this occurs during the linesearch, `nag_nlp_sparse_sol` will then terminate with `error%code` = 201. Otherwise, the linesearch will shorten the step and try again.

`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 `nag_nlp_sparse_sol` will terminate with `error%code` = 202 regardless of the value of `continue`.

`real(kind=`*wp*`), intent(out) :: obj_f`

Output: the value of the objective function at x.

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

Shape: `obj_grad` has shape (n_1').

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

Output: if `obj_grad` is present, then:

if `obj_deriv` = `.true.` (the default), `obj_grad` must contain *all* the elements of the vector $g(x)$ given by

$$g(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_{n_1'}} \right)^T,$$

where $\partial f/\partial x_i$ is the partial derivative of the nonlinear part of the objective function with respect to the ith nonlinear objective variable evaluated at x, for $i = 1, 2, \ldots, n_1'$. Note that constant elements *must* be loaded into `obj_grad` on every call to this procedure unless `obj_row` > 0.

If `obj_deriv` = `.false.`, any *available* elements of the vector $g(x)$ must be assigned to the corresponding elements of `obj_grad`; the remaining elements *must remain unchanged*. Just before `obj_fun` is called, each element of `obj_grad` 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.

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

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

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

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 for nag_nlp_sparse_cntrl_*wp*).

Constraints: obj_fun must be present if num_nlin_obj_var is present and > 0.

obj_row — integer, intent(in), optional

Input: if obj_row $>$ num_nlin_con, row obj_row of A is a free row containing the non-zero elements of the linear part of the objective function. If obj_row $= 0$, there is no free row. If obj_row $= -1$ (the default), there is a dummy 'free' row and this procedure will assume that your problem either has no general constraints or only upper and lower bounds on the variables.

Constraints:

obj_row ≥ -1;

$m = 1$ when obj_row $= -1$;

num_nlin_con $<$ obj_row $\leq m$ when obj_row > 0.

Default: obj_row $= -1$.

obj_deriv — logical, intent(in), optional

Input: specifies whether or not all elements of the objective gradient are provided by the user.

If obj_deriv = .true. (the default), then all elements of the objective gradient are provided.

If obj_deriv = .false., then it is assumed that some elements of the objective gradient 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 required for each element estimated. Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* [5] 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.)

The setting obj_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: obj_deriv must not be present unless obj_fun is present.

Default: obj_deriv = .true..

i_comm(:) — integer, intent(in), optional

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

Input: these arrays are not used by this procedure, but they are passed directly from the calling (sub)program to the user supplied procedures obj_fun and/or con_fun, and hence may be used to pass information to them.

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 x. 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_nlp_sparse_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_z) — real(kind=*wp*), intent(inout), optional

Input: the non-zero elements of the Jacobian matrix A, ordered by increasing column index. Since the constraint Jacobian matrix $J(x'')$ must always appear in the top left-hand corner of A, those elements in a column associated with any nonlinear constraints must come before any elements belonging to the linear constraint matrix G and the free row (if any; see `obj_row` above).

In general, A is partitioned into a nonlinear part and a linear part corresponding to the nonlinear variables and linear variables in the problem. Elements in the nonlinear part may be set to any value (e.g., zero) because they are initialized at the first point that satisfies the linear constraints and the upper and lower bounds. The linear part must contain the non-zero elements of G and the free row (if any).

If `con_deriv` = `.true.` (the default), the nonlinear part may also be used to store any constant Jacobian elements. Note that if `con_fun` does not define the constant Jacobian element `con_jac`(i), the missing value will be obtained directly from `a`(j) for some $j \geq i$.

If `con_deriv` = `.false.`, unassigned elements of `con_jac` are *not* treated as constant; they are estimated by finite differences, at non-trivial expense.

Elements with the same row and column indices are not allowed. (See also the descriptions for `row_index` and `col_ptr` below.)

Output: elements in the nonlinear part corresponding to nonlinear Jacobian variables are overwritten.

Constraints: `a` must not be present unless `row_index` and `col_ptr` are present.

Default: the problem contains no general constraints, in which case it may be more appropriate to use either `nag_nlp_sol` or `nag_con_nlin_lsq_sol_1` instead.

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

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

Input: the lower and upper bounds on all the slacks s. To specify a non-existent lower bound (i.e., $l_j = -\infty$), set `s_lower(`j`)` $\leq -$`control%inf_bound`; to specify a non-existent upper bound (i.e., $u_j = +\infty$), set `s_upper(`j`)` $\geq +$`control%inf_bound` (see the type definition for `nag_nlp_sparse_cntrl_`*wp*). Note that the lower (upper) bound corresponding to the free row or dummy 'free' row must be set to $-\infty$ ($+\infty$) and stored in the `ABS(obj_row)`-th element of `s_lower` (`s_upper`).

Constraints:

`s_lower` and `s_upper` must not be present unless `a` or `con_fun` is present;

`s_lower`(j) $\leq$ `s_upper`(j) for $j = 1, 2, \ldots, m$;

$|\beta| <$ `control%inf_bound` when `s_lower`(j) = `s_upper`(j) = β;

`s_lower(ABS(obj_row))` $\leq -$`control%inf_bound` when `obj_row` $\neq 0$;

`s_upper(ABS(obj_row))` $\geq +$`control%inf_bound` when `obj_row` $\neq 0$.

Default: `s_lower` = $-$`control%inf_bound`; `s_upper` = $+$`control%inf_bound`.

row_index(n_z) — integer, intent(in), optional

Input: `row_index`(i) must contain the row index of the non-zero element stored in `a`(i), for $i = 1, 2, \ldots, n_z$. The row indices for a column may be supplied in any order subject to the condition that those elements in a column associated with any nonlinear constraints must appear before those elements associated with any linear constraints (including the free row, if any). Note that `con_fun` must define the Jacobian elements in the same order.

Constraints:

`row_index` must not be present unless `a` and `col_ptr` are present;

$1 \leq$ `row_index`(i) $\leq m$, for $i = 1, 2, \ldots, n_z$.

Default: `row_index` must be present if `a` is present.

col_ptr($n+1$) — integer, intent(in), optional

Input: `col_ptr`(j) must contain the index in `a` of the start of the jth column, for $i = 1, 2, \ldots, n$. To specify the jth column as empty, set `col_ptr`(j) = `col_ptr`($j+1$). Note that the first and last elements of `col_ptr` must be such that `col_ptr`(1) = 1 and `col_ptr`($n+1$) = $n_Z + 1$.

Constraints:

`col_ptr` must not be present unless `a` and `row_index` are present;

`col_ptr`(1) = 1;

`col_ptr`(j) ≥ 1, for $j = 2, 3, \ldots, n$;

`col_ptr`($n+1$) = $n_Z + 1$;

$0 \leq$ `col_ptr`($j+1$) $-$ `col_ptr`(j) $\leq m$, for $j = 1, 2, \ldots, n$.

Default: `col_ptr` must be present if `a` is present.

con_fun — subroutine, optional

The procedure `con_fun`, supplied by the user, must calculate the vector $F(x)$ of nonlinear constraint functions and (optionally) its Jacobian ($= \partial F/\partial x$) for a specified n_1'' ($\leq n$) element vector x.

Its specification is:

```
subroutine con_fun(first_call, final_call, x, continue, finish, con_f, &
                   con_jac, i_comm, r_comm)
```

`logical, intent(in) ::  first_call`

Input: `first_call` will be `.true.` when `nag_nlp_sparse_sol` 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`.

`logical, intent(in) ::  final_call`

Input: `final_call` will be `.true.` when `nag_nlp_sparse_sol` calls `con_fun` for the final time, and `.false.` for all previous calls. It allows you to perform some additional computation on the final solution.

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

Shape: `x` has shape (n_1'').

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

`logical, intent(inout) ::  continue`

Input: `continue` will always be `.true.` on entry.

Output: if the nonlinear constraint functions cannot be calculated at the current x, you should set `continue` to `.false.`. Unless this occurs during the linesearch, `nag_nlp_sparse_sol` will then terminate with `error%code` = 201. Otherwise, the linesearch will shorten the step and try again.

`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 `nag_nlp_sparse_sol` will terminate with `error%code` = 202 regardless of the value of `continue`.

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

Shape: `con_f` has shape (n_N).

Output: `con_f`(i) must contain the value of the ith nonlinear constraint at x, for $i = 1, 2, \ldots, n_N$.

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

Shape: `con_jac` has shape $(n_{\rm N} * n_1'')$.

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

Output: if `con_jac` is present, then it must return the available elements of the constraint Jacobian evaluated at x. These elements must be stored in exactly the same positions as implied by the definitions of `a`, `row_index` and `col_ptr` described above. Note that `nag_nlp_sparse_sol` does not perform any internal checks for consistency, so great care is essential.

If `con_deriv` = `.true.` (the default), the value of any constant Jacobian element not defined by this procedure will be obtained directly from `a`.

If `con_deriv` = `.false.`, each element of `con_jac` is set to a special value just before `con_fun` is called. On return from this procedure, any element that retains the value is estimated by finite differences, at non-trivial expense.

```
integer, intent(in), optional ::  i_comm(:)
real(kind=wp), intent(in), optional ::  r_comm(:)
```

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

Note: if there are any nonlinear constraints, then the first call to `con_fun` will precede the first call to `obj_fun`. `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_nlp_sparse_cntrl_`*wp*).

Constraints: `con_fun` must be present if `num_nlin_con` is present and > 0.

con_deriv — logical, intent(in), optional

Input: specifies whether or not all elements of the constraint Jacobian are provided by the user.

If `con_deriv` = `.true.` (the default), then all elements of the constraint Jacobian are provided.

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 to estimate all unspecified elements (if any) in each column of the Jacobian. For example, if the sparsity pattern of 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 and 4 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.* [5] 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.)

The setting `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` is present.

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

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

Input: a quantity used to estimate the amount of workspace needed to store the basis factors. (The bigger the better, since it is not certain how much workspace the basis factors need.) More precisely, if the minimum amount of workspace required to start solving the problem is denoted by w, then the amount of workspace actually allocated by the procedure will be `work_factor` $\times\ w$.

Constraints: `work_factor` > 1.0.

Default: `work_factor` $= 3.0$.

cold_start — logical, intent(in), optional

Input: indicates how a starting basis is to obtained as follows:

if `cold_start = .true.` (the default), then an internal Crash procedure will be used to choose an initial basis;

if `cold_start = .false.`, then a basis is already defined in `x_state` and `s_state` (probably from a previous call).

Default: `cold_start = .true.`.

names$(n+m)$ — character(len=8), intent(in), optional

Input: the column (i.e., variable) and row (i.e., constraint) names to be used in the printed output. More precisely, the first n elements must contain the names for the columns, the next n_N elements must contain the names for the nonlinear rows (if any) and the next $(m-n_N)$ elements must contain the names for the linear rows (if any). Note that the name for the free row or dummy 'free' row must be stored in `names`(n+`ABS(obj_row)`).

Default: the column and row names will be chosen automatically by the procedure.

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

Input: if `cold_start = .true.` (the default), `x_state` must specify the initial states of the variables x. An internal Crash procedure is then used to select an initial basis matrix B. The initial basis matrix will be triangular (neglecting certain small elements in each column). It is chosen from various rows and columns of $(A\ \ -I)$. Possible values for `x_state`(j) (also used by `s_state`) are as follows:

x_state(j)	**State of x**(j) **during Crash procedure**
0 or 1	Eligible for the basis
2	Ignored
3	Eligible for the basis (given preference over 0 or 1)
4 or 5	Ignored

If nothing special is known about the problem, or there is no wish to provide special information, you may set `x_state` = 0 and `x` = 0.0. All variables will then be eligible for the initial basis. Less trivially, to say that the jth variable will probably be equal to one of its bounds, set `x_state`(j) = 4 and `x`(j) = `x_lower`(j) or `x_state`(j) = 5 and `x`(j) = `x_upper`(j) as appropriate.

Following the Crash procedure, variables for which `x_state`(j) = 2 are made superbasic. Other variables not selected for the basis are then made nonbasic at the value `x`(j) if `x_lower`(j) $\leq$ `x`(j) $\leq$ `x_upper`(j), or at the value `x_lower`(j) or `x_upper`(j) closest to `x`(j).

If `cold_start = .false.`, `x_state` must specify the initial states of the variables x. Note that `x_state` already contains valid values if it was present in a previous call with the same value of n.

Output: the final states of the variables x. The significance of each possible value of `x_state`(j) (also used by `s_state`) is as follows:

x_state(j)	**State of variable** j	**Normal value of x**(j)
0	Nonbasic	`x_lower`(j)
1	Nonbasic	`x_upper`(j)
2	Superbasic	Between `x_lower`(j) and `x_upper`(j)
3	Basic	Between `x_lower`(j) and `x_upper`(j)

If `num_infeas` = 0, basic and superbasic variables may be outside their bounds by as much as the value of `control%minor_feas_tol` (see the type definition for `nag_nlp_sparse_cntrl_`*wp*). Note that if scaling is specified, `control%minor_feas_tol` applies to the variables of the *scaled* problem. In this case, the variables of the original problem may be as much as 0.1 outside their bounds, but this is unlikely unless the problem is very badly scaled.

Very occasionally some nonbasic variables may be outside their bounds by as much as `control%minor_feas_tol`, and there may be some nonbasic variables for which `x`(j) lies strictly between its bounds.

If `num_infeas` > 0, some basic and superbasic variables may be outside their bounds by an arbitrary amount (bounded by `sum_infeas` if scaling was not used).

Constraints:

if `cold_start` = `.true.`, $0 \leq$ `x_state`$(j) \leq 5$ for $j = 1, 2, \ldots, n$;

if `cold_start` = `.false.`, `x_state` must be present and $0 \leq$ `x_state`$(j) \leq 3$ for $j = 1, 2, \ldots, n$.

Default: `x_state` = 0.

s_state(m) — integer, intent(inout), optional

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

If `cold_start` = `.false.`, `s_state` must specify the initial states of the slacks s. Note that `s_state` already contains valid values if it was present in a previous call with the same value of m.

Output: the final states of the slacks s. The significance of each possible value of `s_state`(j) is as follows:

s_state(j)	**State of slack** j	**Normal value of s**(j)
0	Nonbasic	`s_lower`(j)
1	Nonbasic	`s_upper`(j)
2	Superbasic	Between `s_lower`(j) and `s_upper`(j)
3	Basic	Between `s_lower`(j) and `s_upper`(j)

Constraints: if `cold_start` = `.false.`, `s_state` must be present and $0 \leq$ `s_state`$(j) \leq 3$ for $j = 1, 2, \ldots, m$.

Default: `s_state` = 0.

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

Output: the values of the Lagrange multipliers for the bounds on the variables (*reduced costs*).

lin_lambda$(m - n_{\mathrm{N}})$ — real(kind=*wp*), intent(out), optional

Output: the values of the Lagrange multipliers for the bounds on the linear constraints (*shadow costs*).

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 set of Lagrange multiplier estimates for the nonlinear constraints. If nothing special is known about the problem, or there is no wish to provide special information, you may set `nlin_lambda` = 0.0.

Output: the values of the Lagrange multipliers for the bounds on the nonlinear constraints (*shadow costs*).

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

Default: `nlin_lambda` = 0.0.

num_infeas — integer, intent(out), optional

Output: the number of constraints that lie outside their bounds by more than the value of `control%minor_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).

If the *linear* constraints are infeasible, the sum of the infeasibilities of the linear constraints is minimized subject to the upper and lower bounds being satisfied. In this case, `num_infeas` contains the number of elements of Gx that lie outside their upper or lower bounds. Note that the nonlinear constraints are not evaluated.

Otherwise, the sum of the infeasibilities of the *nonlinear* constraints is minimized subject to the linear constraints and the upper and lower bounds being satisfied. In this case, `num_infeas` contains the number of elements of $F(x)$ that lie outside their upper or lower bounds.

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

Output: the sum of the infeasibilities of constraints that lie outside their bounds by more than the value of `control%minor_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).

num_superbasic_vars — integer, intent(inout), optional

Input: the number of superbasics, n_S. It need not be specified if `cold_start = .true.` (the default), but must retain its value from a previous call when `cold_start = .false.`.

Output: the final number of superbasics.

Default: `num_superbasic_vars` = 0.

control — type(nag_nlp_sparse_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_nlp_sparse_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	Objective and/or constraint values could not be calculated. This exit occurs if you have set `continue` to `.false.` in `obj_fun` or `con_fun`.
203	The problem is infeasible. The general constraints cannot all be satisfied simultaneously to within the values of `control%major_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`) and `control%minor_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`).
204	The problem is unbounded (or badly scaled). The objective function is not bounded below (or above in the case of maximization) in the feasible region because a nonbasic variable can apparently be increased or decreased by an arbitrary amount without causing a basic variable to violate a bound. Add an upper or lower bound to the variable (whose index is printed by default) and rerun `nag_nlp_sparse_sol`.
205	The problem may be unbounded. Check that the values of `control%unbounded_obj` (default value = 10^{15}) and `control%unbounded_step_size` (default value = max(`control%inf_bound`, 10^{15})) are not too small. This exit also implies that the objective function is not bounded below (or above in the case of maximization) in the feasible region defined by expanding the bounds by the value of `control%violation_lim` (default value = 10.0).
206	Too many superbasic variables. Increase the value of `control%superbasics_lim` (default value = $\min(n, 500, \bar{n}+1)$) and rerun `nag_nlp_sparse_sol`.
207	The user-provided derivatives of the objective function (computed by `obj_fun`) appear to be incorrect. Check that `obj_fun` has been coded correctly and that all relevant elements of the objective gradient have been assigned their correct values.
208	The user-provided derivatives of the constraint functions (computed by `con_fun`) appear to be incorrect. Check that `con_fun` has been coded correctly and that all relevant elements of the nonlinear constraint Jacobian have been assigned their correct values.
209	The current point cannot be improved upon. Check that `obj_fun` and `con_fun` have been coded correctly and that they are consistent with the values of `obj_deriv` (default value = `.true.`) and `con_deriv` (default value = `.true.`).
210	Numerical error in trying to satisfy the linear constraints (or the linearized nonlinear constraints). The basis is very ill-conditioned.
211	Not enough workspace for the basis factors. Increase the value of `work_factor` (default value = 3.0) and rerun `nag_nlp_sparse_sol`.

212	The basis is singular after 15 attempts to factorize it (and adding slacks where necessary). Either the problem is badly scaled or the value of `control%lu_fac_tol` (default value = 5.0 or 100.0) is too large.
213	An unexpected error has occurred. Please contact NAG.

Warnings (error%level = 1):

error%code	**Description**
101	Feasible solution found, but requested accuracy not achieved. Check that the value of `control%major_opt_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`) is not too small (say, < `EPSILON(1.0_`*wp*`)`).
102	Too many iterations. Check that the values of `control%major_iter_lim` (default value = 1000) and/or `control%minor_iter_lim` (default value = 500) and/or `control%iter_lim` (default value = 10000) are not too small.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 2 of this module document. These examples could be modified to use some (or all) of the optional arguments described in Section 3.2.

6 Further Comments

6.1 Accuracy

If the value of `control%major_optim_tol` is set to 10^{-d} (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_nlp_sparse_cntrl_`*wp*) and `error%code` = 0 on exit, then the final value of $f(x)$ should have approximately d correct significant digits.

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` $\geq$ 10 (the default).

When `control%major_print_level` $\geq$ 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).
`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 (6) in Section 1 of the Mathematical Background section of this module document) at the current iterate. As the solution is approached, `Merit Function` will converge to the value of the objective function at the solution. In elastic mode (see Section 2 of the Mathematical Background section of this module document), the merit function is a composite function involving the constraint violations weighted by the value of `control%elastic_wt` (default value = 1.0 or 100.0; see the type definition for `nag_nlp_sparse_cntrl_`*wp*). If there are no nonlinear constraints present (i.e., $n_{\mathrm{N}} = 0$), this entry contains `Objective`, the value of the objective function $f(x)$. In this case, $f(x)$ will decrease monotonically to its optimal value.
`Feasibl`	is the value of *rowerr*, the largest element of the scaled nonlinear constraint vector defined in the description of `control%major_feas_tol` (see the type definition for `nag_nlp_sparse_cntrl_`*wp*). The solution is regarded as 'feasible' if `Feasibl` is less than (or equal to) the value of `control%major_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`). `Feasibl` will be approximately zero in the neighbourhood of a solution. If there are no nonlinear constraints present (i.e., $n_{\mathrm{N}} = 0$), all iterates are feasible and this entry is not printed.
`Optimal`	is the value of *maxgap*, the largest element of the maximum complementarity gap vector defined in the description of `control%major_opt_tol` (see the type definition for `nag_nlp_sparse_cntrl_`*wp*). The Lagrange multipliers are regarded as 'optimal' if `Optimal` is less than (or equal to) the value of `control%major_opt_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`). `Optimal` will be approximately zero in the neighbourhood of a solution.
`Cond Hz`	is an estimate of the condition number of the reduced Hessian of the Lagrangian (not printed if n_{N} and n_1' are both zero). It is the square of the ratio between the largest and smallest diagonal elements of the upper triangular matrix R. This constitutes a lower bound on the condition number of the matrix R^TR that approximates the reduced Hessian. The larger this number, the more difficult the problem.
`PD`	is a two-letter indication of the status of the convergence tests involving the feasibility and optimality of the iterates defined in the descriptions of `control%major_feas_tol` and `control%major_opt_tol` (see the type definition for `nag_nlp_sparse_cntrl_`*wp*). Each letter is `T` if the test is satisfied, and `F` otherwise. The tests indicate whether the values of `Feasibl` and `Optimal` are sufficiently small. For example, `TF` or `TT` is printed if there are no nonlinear constraints present (since all iterates are feasible). If either indicator is `F` when the procedure terminates with `error%code` = 0, you should check the solution carefully.
`M`	is printed if an extra evaluation of `obj_fun` and `con_fun` was needed in order to define an acceptable positive-definite quasi-Newton update to the Hessian of the Lagrangian. This modification is only performed when there are nonlinear constraints present (i.e., $n_{\mathrm{N}} > 0$).
`m`	is printed if, in addition, it was also necessary to modify the update to include an augmented Lagrangian term.
`s`	is printed if a self-scaled BFGS (Broyden–Fletcher–Goldfarb–Shanno) update was performed. This update is always used when the Hessian approximation is diagonal, and hence always follows a Hessian reset.
`S`	is printed if, in addition, it was also necessary to modify the self-scaled update in order to maintain positive-definiteness.
`n`	is printed if no positive-definite BFGS update could be found, in which case the approximate Hessian is unchanged from the previous iteration.

`r`	is printed if the approximate Hessian was reset after 10 consecutive major iterations in which no BFGS update could be made. The diagonal elements of the approximate Hessian are retained if at least one update has been performed since the last reset. Otherwise, the approximate Hessian is reset to the identity matrix.
`R`	is printed if the approximate Hessian has been reset by discarding all but its diagonal elements. This reset will be forced periodically by the values of `control%hess_freq` (default value = 99999999) and `control%hess_upd` (default value = 20 or 99999999; see the type definition for `nag_nlp_sparse_cntrl_`*wp*). However, it may also be necessary to reset an ill-conditioned Hessian from time to time.
`l`	is printed if the change in the norm of the variables was greater than the value defined by `control%major_step_lim` (default value = 2.0; see the type definition for `nag_nlp_sparse_cntrl_`*wp*). If this output occurs frequently during later iterations, it may be worthwhile increasing the value of `control%major_step_lim`.
`c`	is printed if central differences have been used to compute the unknown elements of the objective and constraint gradients. A switch to central differences is made if either the linesearch gives a small step, or x is close to being optimal. In some cases, it may be necessary to re-solve the QP subproblem with the central difference gradient and Jacobian.
`u`	is printed if the QP subproblem was unbounded.
`t`	is printed if the minor iterations were terminated after the number of iterations specified by the value of `control%minor_iter_lim` (default value = 500; see the type definition for `nag_nlp_sparse_cntrl_`*wp*) was reached.
`i`	is printed if the QP subproblem was infeasible when the procedure was not in elastic mode. This event triggers the start of nonlinear elastic mode, which remains in effect for all subsequent iterations. Once in elastic mode, the QP subproblems are associated with the elastic problem (see (8) in Section 2 of the Mathematical Background section of this module document). It is also printed if the minimizer of the elastic subproblem does not satisfy the linearized constraints when this procedure is already in elastic mode. (In this case, a feasible point for the usual QP subproblem may or may not exist.)
`w`	is printed if a weak solution of the QP subproblem was found.

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.

`Variable` gives the name of the variable. If `names` (see Section 3.2) is present, the name supplied in `names`(j) is assigned to the jth variable. Otherwise, a default name is used.

`State` gives the state of the variable (`LL` if nonbasic on its lower bound, `UL` if nonbasic on its upper bound, `EQ` if nonbasic and fixed, `FR` if nonbasic and strictly between its bounds, `BS` if basic and `SBS` if superbasic).

A key is sometimes printed before `State` to give additional information about the state of a variable. Note that unless the value of `control%scale_opt` is set to 0 (default value = 1 or 2; see the type definition for `nag_nlp_sparse_cntrl_`*wp*), the tests for assigning a key are applied to the variables of the scaled problem.

`A`	*Alternative optimum possible.* The variable is nonbasic, but its reduced gradient is essentially zero. This means that if the variable were allowed to start moving away from its current value, there would be no change in the value of the objective function. The values of the basic and superbasic variables *might* change, giving a genuine alternative solution. The values of the Lagrange multipliers *might* also change.
`D`	*Degenerate.* The variable is basic, but it is equal to (or very close to) one of its bounds.
`I`	*Infeasible.* The variable is basic and is currently violating one of its bounds by more than the value of `control%minor_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).

N	*Not precisely optimal.* x_j is nonbasic. Its reduced gradient is larger than the value of `control%major_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).

`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_nlp_sparse_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`.
`Residual`	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 general constraints is the same as that given above for variables, with 'variable' replaced by 'constraint', n replaced by m, `names`(j) replaced by `names`$(n+j)$, `x_lower` and `x_upper` are replaced by `s_lower` and `s_upper` respectively, and with the following change in the heading:

`Constrnt`	gives the name of the general constraint.

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

When `control%major_print_level` ≥ 20 and `control%lt80_char = .false.`, the following line of intermediate printout (< 120 characters) is produced at every iteration. Unless stated otherwise, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Major`	(see `Maj` above)
`Minor`	(see `Mnr` above)
`Step`	(as above)
`nObj`	is the number of times `obj_fun` has been called to evaluate the nonlinear part of the objective function. Evaluations needed for the estimation of the gradients by finite differences are not included. `nObj` is printed as a guide to the amount of work required for the linesearch.
`nCon`	is the number of times `con_fun` has been called to evaluate the nonlinear constraint functions (not printed if n_{N} is zero).
`Merit`	(see `Merit Function` above)
`Feasibl`	(as above)
`Optimal`	(as above)
`nS`	is the current number of superbasic variables.
`Penalty`	is the Euclidean norm of the vector of penalty parameters used in the augmented Lagrangian function (not printed if n_{N} is zero).
`LU`	is the number of non-zeros representing the basis factors L and U on completion of the QP subproblem. If there are nonlinear constraints present, the basis factorization $B = LU$ is computed at the start of the first minor iteration. At this stage, `LU` = `lenL` + `lenU`, where `lenL` is the number of subdiagonal elements in the columns of a lower triangular matrix and `lenU` is the number of diagonal and superdiagonal elements in the rows of an upper triangular matrix. As columns of B are replaced during the minor iterations, the value of `LU` may fluctuate up or down (but in general will tend to increase). As the solution is approached and the number of minor iterations required to solve each QP subproblem decreases towards zero, `LU` will reflect the number of non-zeros in the LU factors at the start of each QP subproblem.

	If there are no nonlinear constraints present, refactorization is subject only to the value of `control%fac_freq` (default value = 50 or 100; see the type definition for `nag_nlp_sparse_cntrl_`*wp*) and hence `LU` will tend to increase between factorizations.
`Swp`	is the number of columns of the basis matrix B that were swapped with columns of S in order to improve the condition number of B (not printed if n_{N} is zero). The swaps are determined by an LU factorization of the rectangular matrix $B_S = (B \;\; S)^T$, with stability being favoured more than sparsity.
`Cond Hz`	(as above)
`PD`	(as above)
`M`	(as above)
`m`	(as above)
`s`	(as above)
`S`	(as above)
`n`	(as above)
`r`	(as above)
`R`	(as above)
`l`	(as above)
`c`	(as above)
`u`	(as above)
`t`	(as above)
`i`	(as above)
`w`	(as above)

When `control%major_print_level` $\geq$ 20 and `control%lt80_char` = `.true.` (the default), the following lines of intermediate printout (up to 120 characters) are produced whenever the matrix B or $B_S = (B \;\; S)^T$ is factorized prior to solving the next QP subproblem. Gaussian elimination is used to compute a sparse LU factorization of B or B_S, where PLP^T is a lower triangular matrix and PUQ is an upper triangular matrix for some permutation matrices P and Q. The factorization is stabilized in the manner described under the component `lu_fac_tol` (default value = 5.0 or 100.0) of the optional argument `control`.

`Factorize` is the factorization count.

`Demand` is a code giving the reason for the present factorization as follows:

Code	**Meaning**
0	First LU factorization.
1	The number of updates reached the value of `control%fac_freq` (default value = 50 or 100; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).
2	The number of non-zeros in the updated factors has increased significantly.
7	Not enough storage to update factors.
10	Row residuals too large.
11	Ill-conditioning has caused inconsistent results.

`Iteration`	is the iteration count.
`Nonlinear`	is the number of nonlinear variables in the current basis B (not printed if B_S is factorized).
`Linear`	is the number of linear variables in B (not printed if B_S is factorized).
`Slacks`	is the number of slack variables in B (not printed if B_S is factorized).
`Elems`	is the number of non-zeros in B (not printed if B_S is factorized).
`Density`	is the percentage non-zero density of B (not printed if B_S is factorized). More precisely, `Density` = 100× `Elems`/(`Nonlinear` + `Linear` + `Slacks`)^2.
`Compressns`	is the number of times the data structure holding the partially factorized matrix needed to be compressed, in order to recover unused workspace. Ideally, it should be zero.

`Merit`	is the average Markowitz merit count for the elements chosen to be the diagonals of PUQ. Each merit count is defined to be $(c-1)(r-1)$, where c and r are the number of non-zeros in the column and row containing the element at the time it is selected to be the next diagonal. `Merit` is the average of `m` such quantities. It gives an indication of how much work was required to preserve sparsity during the factorization.
`lenL`	is the number of non-zeros in L.
`lenU`	is the number of non-zeros in U.
`Increase`	is the percentage increase in the number of non-zeros in L and U relative to the number of non-zeros in B. More precisely, `Increase` = 100×(`lenL`+`lenU`−`Elems`)/`Elems`.
`m`	is the number of rows in the problem. Note that `m` = `Ut` + `Lt` + `bp`.
`Ut`	is the number of triangular rows of B at the top of U.
`d1`	is the number of columns remaining when the density of the basis matrix being factorized reached 0.3.
`Lmax`	is the maximum subdiagonal element in the columns of L. This will not exceed the value of `control%lu_fac_tol` (default value = 5.0 or 100.0; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).
`Bmax`	is the maximum non-zero element in B (not printed if B_S is factorized).
`BSmax`	is the maximum non-zero element in B_S (not printed if B is factorized).
`Umax`	is the maximum non-zero element in U, excluding elements of B that remain in U unchanged. (For example, if a slack variable is in the basis, the corresponding row of B will become a row of U without modification. Elements in such rows will not contribute to `Umax`. If the basis is strictly triangular, *none* of the elements of B will contribute, and `Umax` will be zero.) Ideally, `Umax` should not be significantly larger than `Bmax`. If it is several orders of magnitude larger, it may be advisable to reset the value of `control%lu_fac_tol` to some value nearer unity. `Umax` is not printed if B_S is factorized.
`Umin`	is the magnitude of the smallest diagonal element of PUQ.
`Growth`	is the value of the ratio `Umax`/`Bmax`, which should not be too large. Providing `Lmax` is not large (say < 10.0), the ratio max(`Bmax`,`Umax`)/`Umin` is an estimate of the condition number of B. If this number is extremely large, the basis is nearly singular and some numerical difficulties might occur. (However, an effort is made to avoid near-singularity by using slacks to replace columns of B that would have made `Umin` extremely small, and the modified basis is refactorized.)
`Lt`	is the number of triangular columns of B at the left of L.
`bp`	is the size of the 'bump' or block to be factorized nontrivially after the triangular rows and columns of B have been removed.
`d2`	is the number of columns remaining when the density of the basis matrix being factorized has reached 0.6.

When `control%major_print_level` ≥ 20, `control%lt80_char` = `.true.` (the default) and the value of `control%crash_opt` > 0 (default value = 0 or 3; see the type definition for `nag_nlp_sparse_cntrl_`*wp*), the following lines of intermediate printout (< 80 characters) are produced at every iteration. They refer to the number of columns selected by the Crash procedure during each of several passes through A while searching for a triangular basis matrix.

`Slacks`	is the number of slacks selected initially.
`Free cols`	is the number of free columns in the basis, including those whose bounds are rather far apart.
`Preferred`	is the number of 'preferred' columns in the basis (i.e., `x_state`(j) = 3 for some $j \leq n$). It will be a subset of the columns for which `x_state`(j) = 3 was specified.
`Unit`	is the number of unit columns in the basis.
`Double`	is the number of columns in the basis containing 2 non-zeros.
`Triangle`	is the number of triangular columns in the basis with 3 (or more) non-zeros.
`Pad`	is the number of slacks used to pad the basis (to make it a non-singular triangle).

When control%major_print_level $= 1$ or ≥ 10, the following lines of final printout (up to 120 characters) are produced. Note that the final printout includes a listing of the status of every variable and constraint.

Let x_j denote the jth 'column variable', for $j = 1, 2, \ldots, n$. We assume that a typical variable x_j has bounds $\alpha \leq x_j \leq \beta$.

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

`Number`	is the column number j. (This is used internally to refer to x_j in the intermediate output.)
`Column`	gives the name of x_j.
`State`	gives the state of x_j relative to the bounds α and β. The various possible states are as follows:

`LL`	x_j is nonbasic at its lower limit, α.
`UL`	x_j is nonbasic at its upper limit, β.
`EQ`	x_j is nonbasic and fixed at the value $\alpha = \beta$.
`FR`	x_j is nonbasic at some value strictly between its bounds: $\alpha < x_j < \beta$.
`BS`	x_j is basic. Usually $\alpha < x_j < \beta$.
`SBS`	x_j is superbasic. $\alpha < x_j < \beta$.

A key is sometimes printed before `State` to give additional information about the state of a variable. Note that unless the value of control%scale_opt is set to 0 (default value = 1 or 2; see the type definition for nag_nlp_sparse_cntrl_*wp*), the tests for assigning a key are applied to the variables of the scaled problem.

`A`	*Alternative optimum possible.* x_j is nonbasic, but its reduced gradient is essentially zero. This means that if x_j were allowed to start moving away from its current value, there would be no change in the value of the objective function. The values of the basic and superbasic variables *might* change, giving a genuine alternative solution. The values of the Lagrange multipliers *might* also change.
`D`	*Degenerate.* x_j is basic, but it is equal to (or very close to) one of its bounds.
`I`	*Infeasible.* x_j is basic and is currently violating one of its bounds by more than the value of control%minor_feas_tol (default value = SQRT(EPSILON(1.0_*wp*)); see the type definition for nag_nlp_sparse_cntrl_*wp*).
`N`	*Not precisely optimal.* x_j is nonbasic. Its reduced gradient is larger than the value of control%major_feas_tol (default value = SQRT(EPSILON(1.0_*wp*)); see the type definition for nag_nlp_sparse_cntrl_*wp*).

`Activity`	is the value of x_j at the final iterate.
`Obj Gradient`	is the value of g_j at the final iterate. (If any x_j is infeasible, g_j is the gradient of the sum of infeasibilities.)
`Lower Bound`	is α, the lower bound specified for x_j. `None` indicates that x_lower(j) $\leq$ $-$control%inf_bound (default value = 10^{20}; see the type definition for nag_nlp_sparse_cntrl_*wp*).
`Upper Bound`	is β, the upper bound specified for x_j. `None` indicates that x_upper(j) $\geq$ control%inf_bound.
`Reduced Gradnt`	is the value of d_j at the final iterate.
`m+j`	is the value of $m + j$.

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

General linear constraints take the form $l \leq Ax \leq u$. The ith constraint is therefore of the form $\alpha \leq a_i^T x \leq \beta$, and the value of $a_i^T x$ is called the *row activity*. Internally, the linear constraints take the form $Ax - s = 0$, where the slack variables s should satisfy the bounds $l \leq s \leq u$. For the ith 'row', it is the slack variable s_i that is directly available, and it is sometimes convenient to refer to its state. Slacks may be basic or nonbasic (but not superbasic).

Nonlinear constraints $\alpha \leq F_i(x) + a_i^T x \leq \beta$ are treated similarly, except that the row activity and degree of infeasibility are computed directly from $F_i(x) + a_i^T x$ rather than from s_i.

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

`Number`	is the value of $n+i$. (This is used internally to refer to s_i in the intermediate output.)
`Row`	gives the name of the ith row.
`State`	gives the state of the ith row relative to the bounds α and β. The various possible states are as follows:

`LL`	The row is at its lower limit, α.
`UL`	The row is at its upper limit, β.
`EQ`	The limits are the same ($\alpha = \beta$).
`BS`	The constraint is not binding. s_i is basic.

A key is sometimes printed before `State` to give additional information about the state of s_i. Note that unless the value of `control%scale_opt` is set to 0 (default value = 1 or 2; see the type definition for `nag_nlp_sparse_cntrl_`*wp*), the tests for assigning a key are applied to the variables of the scaled problem.

`A`	*Alternative optimum possible.* s_i is nonbasic, but its reduced gradient is essentially zero. This means that if s_i were allowed to start moving away from its current value, there would be no change in the value of the objective function. The values of the basic and superbasic variables *might* change, giving a genuine alternative solution. The values of the Lagrange multipliers *might* also change.
`D`	*Degenerate.* s_i is basic, but it is equal to (or very close to) one of its bounds.
`I`	*Infeasible.* s_i is basic and is currently violating one of its bounds by more than the value of `control%minor_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).
`N`	*Not precisely optimal.* s_i is nonbasic. Its reduced gradient is larger than the value of `control%major_feas_tol` (default value = `SQRT(EPSILON(1.0_`*wp*`))`; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).

`Activity`	is the value of $a_i^T x$ (or $F_i(x) + a_i^T x$ for nonlinear rows) at the final iterate.
`Slack Activity`	is the value by which the row differs from its nearest bound. (For the free row (if any), it is set to `Activity`.)
`Lower Bound`	is α, the lower bound specified for s_i. `None` indicates that `s_lower`$(j) \leq$ $-$`control%inf_bound` (default value = 10^{20}; see the type definition for `nag_nlp_sparse_cntrl_`*wp*).
`Upper Bound`	is β, the upper bound specified for s_i. `None` indicates that `s_upper`$(j) \geq$ `control%inf_bound`.
`Dual Activity`	is the value of the dual variable π_i.
`i`	gives the index i of the ith row.

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 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 printout is produced only if `control%minor_print_level` ≥ 1 (default value $= 0$, which produces no output).

When `control%minor_print_level` ≥ 1 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.
`Ninf`	is the number of infeasibilities. This will not increase unless the iterations are in elastic mode. `Ninf` will be zero during the optimality phase.
`Sinf`	is the value of the sum of infeasibilities if `Ninf` is non-zero. This will be zero during the optimality phase.
`Objective`	is the value of the current QP objective function when `Ninf` is zero and the iterations are not in elastic mode. The switch to elastic mode is indicated by a change in the heading to `Composite Obj` (see below).
`Composite Obj`	is the value of the composite objective function (see (8) in Section 2 of the Mathematical Background section of this module document) when the iterations are in elastic mode. This function will decrease monotonically at each iteration.
`Norm rg`	is the Euclidean norm of the reduced gradient of the QP objective function. During the optimality phase, this norm will be approximately zero after a unit step.

When `control%minor_print_level` ≥ 1 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.

In the description below, a ‘pricing’ operation is defined to be the process by which a nonbasic variable is selected to become superbasic (in addition to those already in the superbasic set). If the problem is purely linear, the variable selected will usually become basic immediately (unless it happens to reach its opposite bound and return to the nonbasic set).

`Itn`	(as above)
`pp`	is the partial price indicator. The variable selected by the last pricing operation came from the `pp`-th partition of A and $-I$. Note that `pp` is reset to zero whenever the basis is refactorized.
`dj`	is the value of the reduced gradient (or reduced cost) for the variable selected by the pricing operation at the start of the current iteration.
`+SBS`	is the variable selected by the pricing operation to be added to the superbasic set.
`-SBS`	is the variable chosen to leave the superbasic set. It has become basic if the entry under `-B` is non-zero; otherwise it has become nonbasic.
`-BS`	is the variable removed from the basis (if any) to become nonbasic.
`-B`	is the variable removed from the basis (if any) to swap with a slack variable made superbasic by the latest pricing operation. The swap is done to ensure that there are no superbasic slacks.
`Step`	(as above)
`Pivot`	is the rth element of a vector y satisfying $By = a_q$ whenever a_q (the qth column of the constraint matrix $(A\ \ -I)$) replaces the rth column of the basis matrix B. Wherever possible, `Step` is chosen so as to avoid extremely small values of `Pivot` (since they may cause the basis to be nearly singular). In extreme cases, it may be necessary to increase the value of `control%pivot_tol` (default value $=$ (`EPSILON`(1.0_*wp*))$^{0.67}$; see the type definition for `nag_nlp_sparse_cntrl_`*wp*) to exclude very small elements of y from consideration during the computation of `Step`.
`Ninf`	(as above)

`Sinf/Objective`	is the value of the current objective function. If x is infeasible, `Sinf` gives the value of the sum of infeasibilities at the start of the current iteration. It will usually decrease at each non-zero value of `Step`, but may occasionally increase if the value of `Ninf` decreases by a factor of 2 or more. However, in elastic mode this entry gives the value of the composite objective function (see (8) in Section 2 of the Mathematical Background section of this module document), which will decrease monotonically at each iteration. If x is feasible, `Objective` is the value of the current QP objective function.
`L`	is the number of non-zeros in the basis factor L. Immediately after a basis factorization $B = LU$, this entry contains `lenL`. Further non-zeros are added to `L` when various columns of B are later replaced. (Thus, `L` increases monotonically.)
`U`	is the number of non-zeros in the basis factor U. Immediately after a basis factorization $B = LU$, this entry contains `lenU`. As columns of B are replaced, the matrix U is maintained explicitly (in sparse form). The value of `U` may fluctuate up or down; in general, it will tend to increase.
`Ncp`	is the number of compressions required to recover workspace in the data structure for U. This includes the number of compressions needed during the previous basis factorization. Normally, `Ncp` should increase very slowly.

The following items are printed only if the problem is nonlinear or the superbasic set is non-empty (i.e., if the current solution is nonbasic).

`Norm rg`	(as above)
`nS`	is the current number of superbasic variables.
`Cond Hz`	(as above)

Procedure: nag_nlp_sparse_cntrl_init

1 Description

`nag_nlp_sparse_cntrl_init` assigns default values to the components of a structure of the derived type `nag_nlp_sparse_cntrl_`*wp*.

2 Usage

```
USE nag_nlp_sparse
CALL nag_nlp_sparse_cntrl_init(control)
```

3 Arguments

3.1 Mandatory Argument

control — type(nag_nlp_sparse_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_nlp_sparse_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_nlp_sparse_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_nlp_sparse_cntrl_dp. For single precision the name is nag_nlp_sparse_cntrl_sp. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

A structure of type nag_nlp_sparse_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 nag_nlp_sparse_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_nlp_sparse_cntrl_wp
  !  Printing parameters
  logical ::  list
  integer ::  unit
  logical ::  lt80_char
  integer ::  major_print_level
  integer ::  minor_print_level
  !  Derivative verification and approximation
  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) ::  fwd_diff_int
  real(kind=wp) ::  cent_diff_int
  !  Tolerances and limits
  integer ::  check_freq
  integer ::  crash_opt
  integer ::  expand_freq
  integer ::  fac_freq
  integer ::  hess_freq
  integer ::  hess_upd
  integer ::  iter_lim
  integer ::  major_iter_lim
  integer ::  minor_iter_lim
  integer ::  part_price
  integer ::  scale_opt
  integer ::  superbasics_lim
  real(kind=wp) ::  crash_tol
  real(kind=wp) ::  elastic_wt
  real(kind=wp) ::  fun_prec
  real(kind=wp) ::  inf_bound
  real(kind=wp) ::  linesearch_tol
  real(kind=wp) ::  lu_den_tol
  real(kind=wp) ::  lu_fac_tol
  real(kind=wp) ::  lu_sing_tol
```

```
    real(kind=wp) ::  lu_upd_tol
    real(kind=wp) ::  major_feas_tol
    real(kind=wp) ::  major_opt_tol
    real(kind=wp) ::  major_step_lim
    real(kind=wp) ::  minor_feas_tol
    real(kind=wp) ::  minor_opt_tol
    real(kind=wp) ::  pivot_tol
    real(kind=wp) ::  scale_tol
    real(kind=wp) ::  unbounded_obj
    real(kind=wp) ::  unbounded_step_size
    real(kind=wp) ::  violation_lim
    logical ::  deriv_linesearch
    logical ::  feas_exit
    logical ::  hess_lim_mem
    logical ::  minimize
  end type nag_nlp_sparse_cntrl_wp
```

3 Components

3.1 Printing Parameters

list — logical

Controls the printing of the parameter settings in the call to `nag_nlp_sparse_sol`.

If `list` = `.true.` (the default), 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_nlp_sparse_sol` 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_nlp_sparse_sol`.

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

if `lt80_char` = `.false.`, then the output will not exceed 120 characters per line whenever `major_print_level` = 5 or ≥ 10 (the default) or `minor_print_level` ≥ 1 (default value = 0).

Default: `lt80_char` = `.true.`.

major_print_level — integer

Controls the amount of output produced by the major iterations of `nag_nlp_sparse_sol`, as indicated below. A detailed description of the printed output is given in Section 7.1 of the procedure document for `nag_nlp_sparse_sol`.

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.
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.
	5	One long line of output (< 120 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.

Default: `major_print_level` = 10.

minor_print_level — integer

Controls the amount of output produced by the minor iterations of `nag_nlp_sparse_sol`, as indicated below. A detailed description of the printed output is given in Section 7.2 of the procedure document for `nag_nlp_sparse_sol`.

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

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

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

	0	No output.
$\geq$	1	One long line of output (< 120 characters) for each minor iteration (no printout of the final QP solution).

Default: `minor_print_level` = 0.

3.2 Derivative Verification and Approximation

Let $\hat{x}$ denote the first point that satisfies the linear constraints and bounds on the variables.

cheap_test — logical

`cheap_test` specifies the level of verification of elements computed by the user-supplied procedures `obj_fun` and `con_fun` (see Section 3 of the procedure document for `nag_nlp_sparse_sol`).

If `cheap_test` = `.true.` (the default), then only a 'cheap' test will be performed on the objective gradient and constraint Jacobian at the point $\hat{x}$ (requiring three calls to `obj_fun` and two calls to `con_fun`). Note that no checks are carried out if every column of the constraint Jacobian contains a missing element.

If `cheap_test` = `.false.`, then a more reliable (but more expensive) check will be made on individual objective gradient and constraint Jacobian elements at the point $\hat{x}$ (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 gradient are to be checked. (Note that unspecified elements are not checked, and hence they result in no overhead.)

If `obj_verify` = `.true.` (the default), then individual objective gradient elements within the range specified by `start_obj_check` (default value = 1) to `stop_obj_check` (default value = the number of variables) will be checked at the point $\hat{x}$. 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 gradient.

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 gradient 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 gradient to be checked.

Default: `stop_obj_check` = the number of nonlinear objective variables.

Constraints: 1 ≤ `start_obj_check` ≤ `stop_obj_check` ≤ the 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, and hence they result in no overhead.)

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 $\hat{x}$. 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` = the number of nonlinear constraint variables.

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

fwd_diff_int — real(kind=*wp*)

`fwd_diff_int` defines an interval used to estimate derivatives by forward differences in the following circumstances:

(a) for verifying the objective gradient and/or constraint Jacobian (see the descriptions of `cheap_test`, `obj_verify` and `con_verify`);

(b) for estimating unspecified elements of the objective gradient and/or constraint Jacobian.

A derivative with respect to x_j is estimated by perturbing that element of x to the value x_j + `fwd_diff_int` × $(1 + |x_j|)$, and then evaluating $f(x)$ and/or $F(x)$ (as appropriate) at the perturbed point. The resulting gradient estimates should be accurate to O(`fwd_diff_int`), unless the functions are badly scaled. Judicious alteration of `fwd_diff_int` may sometimes lead to greater accuracy. See Gill *et al.* [5] for a discussion of the accuracy in finite difference approximations.

Default: `fwd_diff_int` = `SQRT(fun_prec)`.

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

cent_diff_int — real(kind=*wp*)

`cent_diff_int` specifies the difference interval to be used near an optimal solution in order to obtain more accurate (but more expensive) estimates of gradients. This requires twice as many function evaluations as compared to using forward differences (see the description of `fwd_diff_int`). The interval used for the jth variable is $h_j = \texttt{cent\_diff\_int} \times (1+|x_j|)$. The resulting gradient estimates should be accurate to $O((\texttt{cent\_diff\_int})^2)$, unless the functions are badly scaled. The switch to central differences is indicated by `c` at the end of each line of intermediate printout produced by the major iterations (see Section 7.1 of the procedure document for `nag_nlp_sparse_sol`). See Gill *et al.* [5] for a discussion of the accuracy in finite difference approximations.

Default: $\texttt{cent\_diff\_int} = (\texttt{fun\_prec})^{\frac{1}{3}}$.

Constraints: $\texttt{EPSILON(1.0\_}wp\texttt{)} \le \texttt{cent\_diff\_int} < 1.0$.

3.3 Algorithm Choice and Tolerances

check_freq — integer

`check_freq` specifies how often the current solution (x, s) is to be tested to see whether it satisfies the general linear constraints (including any linearized nonlinear constraints). The numerical test is performed every `check_freq`-th minor iteration after the most recent basis factorization. The constraints are of the form $Ax - s = b$, where s is the set of slack variables. If the largest element of the residual vector $r = b - Ax + s$ is judged to be too large, the current basis is refactorized and the basic variables recomputed to satisfy the general constraints more accurately.

If $\texttt{check\_freq} = 0$, the value 99999999 is used instead and effectively no checks are made.

Default: $\texttt{check\_freq} = 60$.

Constraints: $\texttt{check\_freq} \ge 0$.

crash_opt — integer

`crash_opt` is used in conjunction with the optional argument `cold_start` (see Section 3.2 of the procedure document for `nag_nlp_sparse_sol`) in order to select an initial basis.

If `cold_start = .true.` (the default), an internal Crash procedure is used to select an initial basis from various rows and columns of the constraint matrix $(A \ \ -I)$. The value of `crash_opt` determines which rows and columns of A are initially eligible for the basis, and how many times the Crash procedure is called. Columns of $-I$ are used to pad the basis where necessary. The possible choices for `crash_opt` are the following.

0	The initial basis contains only slack variables: $B = I$.
1	The Crash procedure is called once (looking for a triangular basis in all rows and columns of A).
2	The Crash procedure is called twice (if there are any nonlinear constraints). The first call looks for a triangular basis in linear rows, and the iteration proceeds with simplex iterations until the linear constraints are satisfied. The Jacobian is then evaluated for the first major iteration and the Crash procedure is called again to find a triangular basis in the nonlinear rows (whilst retaining the current basis for linear rows).
3	The Crash procedure is called up to three times (if there are any nonlinear constraints). The first two calls treat linear *equality* constraints and linear *inequality* constraints separately. The Jacobian is then evaluated for the first major iteration and the Crash procedure is called again to find a triangular basis in the nonlinear rows (whilst retaining the current basis for linear rows).

If $\texttt{crash\_opt} \ge 1$, certain slacks on inequality rows are selected for the basis first. (If $\texttt{crash\_opt} \ge 2$, numerical values are used to exclude slacks that are close to a bound.) The Crash procedure then makes several passes through the columns of A, searching for a basis matrix that is essentially triangular. A column is assigned to 'pivot' on a particular row if the column contains a suitably large element in a row that has not yet been assigned. (The pivot elements ultimately form the diagonals of the triangular basis.) For remaining unassigned rows, slack variables are inserted to complete the basis.

Default: `crash_opt` $= 0$ if there are any nonlinear constraints, and 3 otherwise.

Constraints: $0 \leq$ `crash_opt` ≤ 3.

crash_tol — real(kind=*wp*)

`crash_tol` is used in conjunction with the optional argument `cold_start` (see Section 3.2 of the procedure document for `nag_nlp_sparse_sol`) in order to select an initial basis.

It allows the Crash procedure to ignore certain 'small' non-zero elements in the columns of A while searching for a triangular basis. If a_{max} is the largest element in the jth column, other non-zeros a_{ij} in the column are ignored if $|a_{ij}| \leq a_{max} \times$ `crash_tol`.

The basis obtained by the Crash procedure may not be strictly triangular when `crash_tol` > 0.0, but it is likely to be non-singular and almost triangular. The intention is to obtain a starting basis containing more columns of A and fewer (arbitrary) slacks. A feasible solution may be reached earlier on some problems.

Default: `crash_tol` $= 0.1$.

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

deriv_linesearch — logical

`deriv_linesearch` specifies the tolerance to be used at every major iteration in order to improve the value of the Lagrangian merit function (6) during the linesearch (see Section 1 of the Mathematical Background section of this module document).

If `deriv_linesearch = .true.` (the default), a linesearch based upon safeguarded cubic interpolation (which requires both function and gradient values in order to compute estimates of the step α_k) is used.

If `deriv_linesearch = .false.`, a linesearch based upon safeguarded quadratic interpolation (which does not require the evaluation or approximation of any gradients) is used. This setting should also be used if some analytic derivatives are not provided.

A nonderivative linesearch can be slightly less robust on difficult problems, and it is recommended that the default be used if the functions and their derivatives can be computed at approximately the same cost. If the gradients are very expensive to compute relative to the functions however, a nonderivative linesearch may result in a significant decrease in the total run-time.

Default: `deriv_linesearch = .true.`.

elastic_wt — real(kind=*wp*)

`elastic_wt` specifies the initial weight γ associated with problem (8) (see Section 2 of the Mathematical Background section of this module document).

At any given major iteration k, elastic mode is entered if the QP subproblem is infeasible or the QP dual variables (Lagrange multipliers) are larger in magnitude than `elastic_wt` $\times (1 + \| g(x_k) \|_2)$, where g is the objective gradient. In either case, the QP subproblem is re-solved in elastic mode with $\gamma =$ `elastic_wt` $\times (1 + \| g(x_k) \|_2)$.

Thereafter, γ is increased (subject to a maximum allowable value) at any point that is optimal for problem (8) but not feasible for problem (1). After the pth increase, $\gamma = (1 + \| g(x_{k_1}) \|_2) \times 10^p \times$ `elastic_wt`, where x_{k_1} is the iterate at which γ was first needed.

Default: `elastic_wt` $= 100.0$ if there are any nonlinear constraints, and 1.0 otherwise.

Constraints: `elastic_wt` ≥ 0.0.

expand_freq — integer

`expand_freq` is part of the EXPAND anti-cycling procedure due to Gill *et al.* [4], which is designed to make progress even on highly degenerate problems.

For linear models, the strategy is to force a positive step at every iteration, at the expense of violating the constraints by a small amount. Suppose that the value of `minor_feas_tol` is δ. Over a period of `expand_freq` iterations, the feasibility tolerance actually used by this procedure (i.e., the *working* feasibility tolerance) increases from 0.5δ to δ (in steps of $0.5\delta/$`expand_freq`).

For nonlinear models, the same procedure is used for iterations in which there is only one superbasic variable. (Cycling can only occur when the current solution is at a vertex of the feasible region.) Thus, zero steps are allowed if there is more than one superbasic variable, but otherwise positive steps are enforced.

Increasing the value of `expand_freq` helps reduce the number of slightly infeasible nonbasic basic variables (most of which are eliminated during the resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see the description of `pivot_tol`).

If `expand_freq` $= 0$, the value 99999999 is used instead and effectively no anti-cycling procedure is invoked.

Default: `expand_freq` $= 10000$.

Constraints: `expand_freq` ≥ 0.

fac_freq — integer

`fac_freq` specifies the maximum number of basis changes that will occur between factorizations of the basis matrix.

For linear problems, the basis factors are usually updated at every iteration. The default value ($=$ 100) is reasonable for typical problems, particularly those that are extremely sparse or well-scaled.

When the objective function is nonlinear, fewer basis updates will occur as the solution is approached. The number of iterations between basis factorizations will therefore increase. During these iterations a test is made regularly according to the value of `check_freq` to ensure that the general constraints are satisfied. If necessary, the basis will be refactorized before the limit of `fac_freq` updates is reached.

Default: `fac_freq` $= 50$ if there are any nonlinear constraints, and 100 otherwise.

Constraints: `fac_freq` ≥ 1.

feas_exit — logical

Note: `feas_exit` only takes effect if the linear constraints are feasible, or the value of `major_iter_lim` is not exceeded.

It specifies whether additional iterations be performed when termination is about to occur at a point that does not satisfy the nonlinear constraints.

> If `feas_exit = .true.`, additional iterations are performed in order to find a feasible point (if any) for the nonlinear constraints. This involves solving a feasible point problem in which the objective function is omitted.
>
> If `feas_exit = .false.` (the default), no additional iterations are performed.

Default: `feas_exit = .false.`.

fun_prec — real(kind=*wp*)

`fun_prec` defines the *relative function precision* ε_{R}, which is intended to be a measure of the relative accuracy with which the nonlinear functions can be computed. For example, if $f(x)$ (or $F_i(x)$) is computed as 1000.56789 for some relevant x and the first 6 significant digits are known to be correct, the appropriate value for ε_{R} would be 10^{-6}.

Ideally the functions $f(x)$ or $F_i(x)$ should have magnitude of order 1. If all functions are substantially *less* than 1 in magnitude, ε_{R} should be the *absolute* precision. For example, if $f(x)$

(or $F_i(x)$) is computed as $1.23456789 \times 10^{-4}$ for some relevant x and the first 6 significant digits are known to be correct, the 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.* [5] for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy.

In some cases the function values will be the result of extensive computation, possibly involving an iterative procedure that can provide few digits of precision at reasonable cost. Specifying an appropriate value of fun_prec may therefore lead to savings, by allowing the linesearch procedure to terminate when the difference between function values along the search direction becomes as small as the absolute error in the values.

Default: fun_prec = (EPSILON(1.0_*wp*))$^{0.8}$.

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

hess_freq — integer

hess_freq specifies the maximum number of BFGS updates allowed between resetting the approximate Hessian to the identity matrix upon completion of a major iteration. It is intended to be used in conjunction with hess_lim_mem.

Default: hess_freq = 99999999 and effectively no resets occur.

Constraints: hess_freq $\geq$ 1.

hess_lim_mem — logical

hess_lim_mem specifies the method for storing and updating the quasi-Newton approximation to the Hessian of the Lagrangian function.

If hess_lim_mem = .true. (the default), a limited memory procedure is used to update a diagonal Hessian approximation H_r a limited number of times. (Updates are accumulated as a list of vector pairs. They are discarded at regular intervals after H_r has been reset to their diagonal.)

If hess_lim_mem = .false., the approximate Hessian is treated as a dense matrix, and BFGS quasi-Newton updates are applied explicitly. This is most efficient when the total number of nonlinear variables is not too large (say, $<$ 75). In this case, you can expect 1-step Q-superlinear convergence to the solution.

Note that if hess_freq = 20 (default value = 99999999) is used in conjunction with hess_lim_mem = .false., the effect will be similar to using hess_lim_mem = .true. in conjunction with hess_upd = 20 (the default), except that the latter will retain the current diagonal during resets.

Default: hess_lim_mem = .true..

hess_upd — integer

Note: hess_upd only takes effect if hess_lim_mem = .true. (the default).

It defines the maximum number of pairs of Hessian update vectors that are to be used to define the quasi-Newton approximate Hessian.

Once the limit of hess_upd updates is reached, all but the diagonal elements of the accumulated updates are discarded and the process starts again. Broadly speaking, the more updates that are stored, the better the quality of the approximate Hessian. On the other hand, the more vectors that are stored, the greater the cost of each QP iteration.

The default value is likely to give a robust algorithm without significant expense, but faster convergence may be obtained with far fewer updates (say, $<$ 10).

Default: hess_upd = 20 if hess_lim_mem = .true. and 99999999 otherwise, in which case effectively no updates are performed.

Constraints: hess_upd $\geq$ 0.

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.

iter_lim — integer

`iter_lim` specifies the maximum number of minor iterations allowed (i.e., iterations of the simplex method or the QP algorithm), summed over all major iterations. (See the description of `minor_iter_lim` and `major_iter_lim`.)

Default: `iter_lim` $= 10000$.

Constraints: `iter_lim` ≥ 1.

linesearch_tol — real(kind=*wp*)

`linesearch_tol` controls the accuracy with which a steplength will be located along the direction of search at each iteration. At the start of each linesearch a target directional derivative for the Lagrangian merit function is identified. The value of `linesearch_tol` therefore determines the accuracy to which this target value is approximated.

The default value ($= 0.9$) requests an inaccurate search, and is appropriate for most problems, particularly those with any nonlinear constraints.

If the nonlinear functions are expensive to evaluate, a less accurate search may be appropriate. If the optional arguments `obj_deriv` and `con_deriv` are both set to `.true.` (the default; see Section 3.2 of the procedure document for `nag_nlp_sparse_sol`), try setting `linesearch_tol` to 0.99. (The number of major iterations required to solve the problem might increase, but the total number of function evaluations may decrease enough to compensate.)

If `obj_deriv` and/or `con_deriv` are set to `.false.`, a moderately accurate search may be appropriate; try setting `linesearch_tol` to 0.5. Each search will (typically) require only $1 - 5$ function values, but many function calls will then be needed to estimate the missing gradients for the next iteration.

If the nonlinear functions are cheap to evaluate, a more accurate search may be appropriate; try setting `linesearch_tol` to 0.1, 0.01 or 0.001. The number of major iterations required to solve the problem might decrease.

Default: `linesearch_tol` $= 0.9$.

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

lu_den_tol — real(kind=*wp*)

lu_sing_tol — real(kind=*wp*)

`lu_den_tol` defines the density tolerance to be used during the LU factorization of the basis matrix. Columns of L and rows of U are formed one at a time, and the remaining rows and columns of the basis are altered appropriately. At any stage, if the density of the remaining matrix exceeds `lu_den_tol`, the Markowitz strategy for choosing pivots is terminated. The remaining matrix is then factorized using a dense LU procedure. Increasing the value of `lu_den_tol` towards unity may give slightly sparser LU factors, with a slight increase in factorization time.

`lu_sing_tol` defines the singularity tolerance to be used to guard against ill-conditioned basis matrices. Whenever the basis is refactorized, the diagonal elements of U are tested as follows. If $|u_{jj}| \leq$ `lu_sing_tol` or $|u_{jj}| <$ `lu_sing_tol` $\times \max_i |u_{ij}|$, the jth column of the basis is replaced by the corresponding slack variable. This is most likely to occur when the optional argument `cold_start` is set to `.false.` (see Section 3.2 of the procedure document for `nag_nlp_sparse_sol`), or at the start of a major iteration.

In some cases, the Jacobian matrix may converge to values that make the basis exactly singular (e.g., a whole row of the Jacobian matrix could be zero at an optimal solution). Before exact singularity occurs, the basis could become very ill-conditioned and the optimization could progress very slowly (if at all). Setting `lu_sing_tol` to 0.00001 (say) may therefore help cause a judicious change of basis in such situations.

Default: `lu_den_tol` = 0.6; `lu_sing_tol` = (`EPSILON(1.0_`*wp*`)`)$^{0.67}$.

Constraints: `lu_den_tol` $\geq$ 0.0; `lu_sing_tol` $>$ 0.0.

lu_fac_tol — real(kind=*wp*)

lu_upd_tol — real(kind=*wp*)

`lu_fac_tol` and `lu_upd_tol` specify tolerances which affect the stability and sparsity of the basis factorization $B = LU$, during refactorization and updating, respectively. The lower triangular matrix L is a product of matrices of the form

$$\begin{pmatrix} 1 & \\ \mu & 1 \end{pmatrix},$$

where the multipliers μ satisfy $|\mu| \leq$ `lu_fac_tol` and $|\mu| \leq$ `lu_upd_tol`. Smaller values of `lu_fac_tol` and `lu_upd_tol` favour stability, while larger values favour sparsity. The default values usually strike a good compromise. For large and relatively dense problems, setting `lu_fac_tol` to 10.0 or 5.0 (say) may give a marked improvement in sparsity without impairing stability to a serious degree. Note that for problems involving band matrices, it may be necessary to reduce `lu_fac_tol` and/or `lu_upd_tol` in order to achieve stability.

Default: `lu_fac_tol` = 5.0 if there are any nonlinear constraints, and 100.0 otherwise; `lu_upd_tol` = 5.0 if there are any nonlinear constraints, and 10.0 otherwise.

Constraints: `lu_fac_tol` $\geq$ 1.0; `lu_upd_tol` $\geq$ 1.0.

major_feas_tol — real(kind=*wp*)

`major_feas_tol` specifies how accurately the nonlinear constraints should be satisfied. The default value is appropriate when the linear and nonlinear constraints contain data to approximately that accuracy. A larger value may be appropriate if some of the problem functions are known to be of low accuracy.

Let *rowerr* be defined as the maximum nonlinear constraint violation normalized by the size of the solution. It is required to satisfy

$$rowerr = \max_i \frac{viol_i}{\| (x,s) \|} \quad \leq \quad \texttt{major\_feas\_tol},$$

where $viol_i$ is the violation of the ith nonlinear constraint.

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

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

major_opt_tol — real(kind=*wp*)

`major_opt_tol` specifies the final accuracy of the dual variables π. If `nag_nlp_sparse_sol` terminates with `error%code` = 0, a primal and dual solution (x, s, π) will have been computed such that

$$maxgap = \max_j \frac{gap_j}{\| \pi \|} \quad \leq \quad \texttt{major\_opt\_tol},$$

where gap_j is an estimate of the complementarity gap for the jth variable and $\| \pi \|$ is a measure of the size of the QP dual variables (or Lagrange multipliers) given by

$$\| \pi \| = \max\left(\frac{\sigma}{\sqrt{m}}, 1\right), \quad \text{where } \sigma = \sum_{i=1}^{m} |\pi_i|.$$

It is included to make the tests independent of a scale factor on the objective function. Specifically, gap_j is computed from the final QP solution using the reduced gradients $d_j = g_j - \pi^T a_j$, where g_j is the jth element of the objective gradient and a_j is the associated column of the constraint matrix $(A \ -I)$:

$$gap_j = \begin{cases} d_j \min(x_j - l_j, 1) & \text{if } d_j \geq 0; \\ -d_j \min(u_j - x_j, 1) & \text{if } d_j < 0. \end{cases}$$

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

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

major_iter_lim — integer

`major_iter_lim` specifies the maximum number of major iterations allowed before termination. It is intended to guard against an excessive number of linearizations of the nonlinear constraints.

If you wish to check that a call to `nag_nlp_sparse_sol` 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` = 1000.

Constraints: `major_iter_lim` ≥ 0.

major_step_lim — real(kind=*wp*)

`major_step_lim` limits the change in x during a linesearch. It applies to all nonlinear problems once a 'feasible solution' or 'feasible subproblem' has been found.

A linesearch determines a step α in the interval $0 < \alpha \leq \beta$, where $\beta = 1$ if there are any nonlinear constraints, or the step to the nearest upper or lower bound on x if all the constraints are linear. Normally, the first step attempted is $\alpha_1 = \min(1, \beta)$.

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. The value of `major_step_lim` is therefore used to define a step limit $\bar{\beta}$ given by

$$\bar{\beta} = \frac{\texttt{major\_step\_lim} \times (1 + \|x\|_2)}{\|p\|_2},$$

where p is the search direction and the first evaluation of $f(x)$ is made at the (potentially) smaller step length $\alpha_1 = \min(1, \bar{\beta}, \beta)$.

Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at meaningless points. 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 `major_step_lim` is selected, a 'good' starting point may be required. An important application is to the class of nonlinear least-squares problems.

Default: `major_step_lim` = 2.0.

Constraints: `major_step_lim` > 0.0.

minimize — logical

`minimize` specifies the required direction of the optimization. It applies to both linear and nonlinear terms (if any) in the objective function $f(x)$.

If `minimize = .true.` (the default), $f(x)$ is minimized.

If `minimize = .false.`, $f(x)$ is maximized.

Note that if two problems are the same except that one minimizes $f(x)$ and the other maximizes $-f(x)$, their solutions will be the same but the signs of the dual variables π_i and the reduced gradients d_j will be reversed.

Default: `minimize = .true.`.

minor_feas_tol — real(kind=*wp*)

minor_feas_tol specifies the tolerance within which all variables eventually satisfy their upper and lower bounds. Since this includes slack variables, general linear constraints should also be satisfied to within minor_feas_tol. Note that feasibility with respect to nonlinear constraints is judged by the value of major_feas_tol (and not by minor_feas_tol).

If the bounds and linear constraints cannot be satisfied to within minor_feas_tol, the problem is declared *infeasible*. Let Sinf be the corresponding sum of infeasibilities (see Section 7.1 of the procedure document for nag_nlp_sparse_sol). If Sinf is quite small, it may be appropriate to raise minor_feas_tol by a factor of 10 or 100. Otherwise, some error in the data should be suspected.

If scale_opt > 1, feasibility is defined in terms of the *scaled* problem (since it is more likely to be meaningful). (See the description of scale_opt.)

Nonlinear functions will only be evaluated at points that satisfy the bounds and linear constraints. If there are regions where a function is undefined, every effort should be made to eliminate these regions from the problem. For example, if $f(x_1, x_2) = \sqrt{x_1} + \log(x_2)$, it is essential to place lower bounds on both x_1 and x_2. If the bounds are specified as $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$, it might be appropriate to specify minor_feas_tol as 10^{-6}. (The log singularity is more serious; in general, you should attempt to keep x as far away from singularities as possible.)

In reality, the value of minor_feas_tol is used as a feasibility tolerance for satisfying the bounds on x and s in each QP subproblem. If the sum of infeasibilities cannot be reduced to zero, the QP subproblem is declared infeasible and the procedure is then in *elastic mode* thereafter (with only the linearized nonlinear constraints defined to be elastic). (See the description of elastic_wt.)

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

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

minor_iter_lim — integer

minor_iter_lim specifies the maximum number of iterations allowed between successive linearizations of the nonlinear constraints. Values in the range 10 to 50 prevent excessive effort being expended on early major iterations, but allow later QP subproblems to be solved to completion. Note that an extra m minor iterations are allowed if the first QP subproblem to be solved starts with the all-slack basis $B = I$. (See the description of crash_tol.)

In general, it is unsafe to specify values as small as 1 or 2 for minor_iter_lim (because even when an optimal solution has been reached, a few minor iterations may be needed for the corresponding QP subproblem to be recognised as optimal).

Default: minor_iter_lim = 500.

Constraints: minor_iter_lim ≥ 1.

minor_opt_tol — real(kind=*wp*)

minor_opt_tol is used to judge optimality for each QP subproblem. Let the QP reduced gradients be $d_j = g_j - \pi^T a_j$, where g_j is the jth element of the QP gradient, a_j is the associated column of the QP constraint matrix and π is the set of QP dual variables.

By construction, the reduced gradients for basic variables are always zero. The QP subproblem will be declared optimal if the reduced gradients for nonbasic variables at their upper or lower bounds satisfy

$$\frac{d_j}{\| \pi \|} \geq -\texttt{minor\_opt\_tol} \quad \text{or} \quad \frac{d_j}{\| \pi \|} \leq \texttt{minor\_opt\_tol}$$

respectively, and if $\frac{|d_j|}{\| \pi \|} \leq$ minor_opt_tol for superbasic variables.

Note that $\| \pi \|$ is a measure of the size of the dual variables. It is included to make the tests independent of a scale factor on the objective function. (The value of $\| \pi \|$ actually used is defined in the description of major_opt_tol.)

If the objective is scaled down to be very *small*, the optimality test reduces to comparing d_j against `minor_opt_tol`.

Default: `minor_opt_tol` = `SQRT(EPSILON(1.0_wp))`.

Constraints: `EPSILON(1.0_wp)` $\leq$ `minor_opt_tol` < 1.0.

part_price — integer

`part_price` is recommended for large problems that have significantly more variables than constraints (i.e., $n \gg m$). It reduces the work required for each 'pricing' operation (i.e., when a nonbasic variable is selected to become superbasic). The possible choices for `part_price` are the following.

1	All columns of the constraint matrix $(A \ \ -I)$ are searched.
≥ 2	Both A and I are partitioned to give `part_price` roughly equal segments A_j, I_j, for $j = 1, 2, \ldots, p$ (modulo p). If the previous pricing search was successful on A_j, I_j, the next search begins on the segments A_{j+1}, I_{j+1}. If a reduced gradient is found that is larger than some dynamic tolerance, the variable with the largest such reduced gradient (of appropriate sign) is selected to enter the basis. If nothing is found, the search continues on the next segments A_{j+2}, I_{j+2}, and so on.

Default: `part_price` = 1 if there are any nonlinear constraints, and 10 otherwise.

Constraints: `part_price` ≥ 1.

pivot_tol — real(kind=*wp*)

`pivot_tol` specifies the tolerance to be used during the solution of QP subproblems in order to prevent columns entering the basis if they would cause the basis to become almost singular.

When x changes to $x + \alpha p$ for some specified search direction p, a 'ratio test' is used to determine which element of x reaches an upper or lower bound first. The corresponding element of p is called the *pivot element*. Elements of p that are smaller than `pivot_tol` are ignored (and therefore cannot be pivot elements).

It is common in practice for two (or more) variables to reach a bound at essentially the same time. In such cases, the value of `minor_feas_tol` provides some freedom to maximize the pivot element and thereby improve numerical stability. Excessively *small* values of `minor_feas_tol` should therefore not be specified. To a lesser extent, the value of `expand_freq` also provides some freedom to maximize the pivot element. Excessively *large* values of `expand_freq` should therefore not be specified. (See the description of `minor_feas_tol` and `expand_freq`.)

Default: `pivot_tol` = $($`EPSILON(1.0_wp)`$)^{0.67}$.

Constraints: `pivot_tol` > 0.0.

scale_opt — integer

`scale_opt` enables you to scale the variables and constraints using an iterative procedure due to Fourer [9], which attempts to compute row scales r_i and column scales c_j such that the scaled matrix coefficients $\bar{a}_{ij} = a_{ij} \times (c_j / r_i)$ are as close as possible to unity. (The lower and upper bounds on the variables and slacks for the scaled problem are redefined as $\bar{l}_j = l_j / c_j$ and $\bar{u}_j = u_j / c_j$ respectively, where $c_j \equiv r_{j-n}$ if $j > n$.) The possible choices for `scale_opt` are the following.

0	No scaling is performed. This is recommended if it is known that the elements of x and the constraint matrix A (along with its Jacobian) never become large (say, $>$ 1000).
1	All linear constraints and variables are scaled. This may improve the overall efficiency of the procedure on some problems.
2	All constraints and variables are scaled. Also, an additional scaling is performed that takes into account columns of $(A \ \ -I)$ that are fixed or have positive lower bounds or negative upper bounds.

If there are any nonlinear constraints present, the scale factors depend on the Jacobian at the first point that satisfies the linear constraints and the upper and lower bounds. The setting scale_opt = 2 should therefore be used only if a 'good' starting point is available and the problem is not highly nonlinear.

Default: scale_opt = 1 if there are any nonlinear constraints, and 2 otherwise.

Constraints: $0 \le$ scale_opt ≤ 2.

scale_tol — real(kind=*wp*)

Note: scale_tol only takes effect if scale_opt = 1 or 2 (the default).

It is used to control the number of scaling passes to be made through the constraint matrix A. At least 3 (and at most 10) passes will be made. More precisely, let a_p denote the largest column ratio (i.e., $\frac{\text{'biggest' element}}{\text{'smallest' element}}$ in some sense) after the pth scaling pass through A. The scaling procedure is terminated if $a_p \ge a_{p-1} \times$ scale_tol for some $p \ge 3$. Thus, increasing the value of scale_tol from 0.9 to 0.99 (say) will probably increase the number of passes through A.

Default: scale_tol = 0.9.

Constraints: $0.0 <$ scale_tol < 1.0.

superbasics_lim — integer

Note: superbasics_lim only takes effect if the problem is nonlinear.

It places a limit on the storage allocated for superbasic variables. Ideally, it should be set to a value slightly larger than the 'number of degrees of freedom' expected at the solution.

The number of degrees of freedom is often called the 'number of independent variables'. Normally, the value of superbasics_lim need not be greater than the total number of nonlinear variables plus one, but for many problems it may be considerably smaller.

Default: superbasics_lim = min(500, the number of variables, the total number of nonlinear variables + 1).

Constraints: superbasics_lim ≥ 1.

unbounded_obj — real(kind=*wp*)

unbounded_step_size — real(kind=*wp*)

unbounded_obj and unbounded_step_size attempt to detect unboundedness in nonlinear problems. During the linesearch, the objective function f is evaluated at points of the form $x + \alpha p$, where x and p are fixed and α varies. If $|f|$ exceeds unbounded_obj or α exceeds unbounded_step_size, the iterations are terminated and the procedure terminates with error%code = 204.

If singularities are present, unboundedness in $f(x)$ may manifest itself by a floating-point overflow during the evaluation of $f(x + \alpha p)$, before the test against unbounded_obj can be made.

Unboundedness in x is best avoided by placing finite upper and lower bounds on the variables.

Default: unbounded_obj = 10^{15}; unbounded_step_size = max(inf_bound, 10^{20}).

Constraints: unbounded_obj > 0.0; unbounded_step_size > 0.0.

violation_lim — real(kind=*wp*)

violation_lim specifies an absolute limit on the magnitude of the maximum constraint violation after the linesearch. Upon completion of the linesearch, the new iterate x_{k+1} satisfies the condition

$$v_i(x_{k+1}) \le \texttt{violation\_lim} \times \max(1, v_i(x_0)),$$

where x_0 is the point at which the nonlinear constraints are first evaluated and $v_i(x)$ is the ith nonlinear constraint violation $v_i(x) = \max(0, l_i - F_i(x), F_i(x) - u_i)$.

The effect of the violation limit is to restrict the iterates to lie in an *expanded* feasible region whose size depends on the magnitude of violation_lim. This makes it possible to keep the iterates within a region where the objective function is expected to be well-defined and bounded below (or above in the case of maximization). If the objective function is bounded below (or above in the case of maximization) for all values of the variables, then `violation_lim` may be any large positive value.

Default: `violation_lim` $= 10.0$.

Constraints: `violation_lim` > 0.0.

Example 1: Nonlinear Programming Problem (with bounds and linear constraints)

This is a reformulation of Problem 74 from Hock and Schittkowski [10] and involves the minimization of the nonlinear function

$$3x_3 + 10^{-6}x_3^3 + 2x_4 + \frac{2}{3} \times 10^{-6}x_4^3$$

subject to the bounds

$$\begin{array}{rcccl} -0.55 & \leq & x_1 & \leq & 0.55 \\ -0.55 & \leq & x_2 & \leq & 0.55 \\ 0 & \leq & x_3 & \leq & 1200 \\ 0 & \leq & x_4 & \leq & 1200 \end{array}$$

to the linear constraints

$$\begin{aligned} -x_1 + x_2 &\geq -0.55, \\ x_1 - x_2 &\geq -0.55, \end{aligned}$$

and to the nonlinear constraints

$$\begin{aligned} &1000\sin(-x_1 - 0.25) + 1000\sin(-x_2 - 0.25) - x_3 = -894.8, \\ &1000\sin(x_1 - 0.25) + 1000\sin(x_1 - x_2 - 0.25) - x_4 = -894.8, \\ &1000\sin(x_2 - 0.25) + 1000\sin(x_2 - x_1 - 0.25) = -1294.8. \end{aligned}$$

The initial point, which is infeasible, is

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

The optimal solution (to five figures) is

$$x^* = (0.11887,\ -0.39623,\ 679.94,\ 1026.0)^T,$$

and $F(x^*) = 5126.4$. All the 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 nlp_sparse_ex01_mod
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  SUBROUTINE obj_fun(first_call,final_call,x,continue,finish,obj_f, &
     obj_grad,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC PRESENT
    ! .. Scalar Arguments ..
    REAL (wp), INTENT (OUT) :: obj_f
    LOGICAL, INTENT (INOUT) :: continue, finish
    LOGICAL, INTENT (IN) :: final_call, first_call
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), OPTIONAL, INTENT (INOUT) :: obj_grad(:)
    REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Executable Statements ..

    obj_f = 1.0E-6_wp*x(3)**3 + (2.0E-6_wp/3.0_wp)*x(4)**3

    IF (PRESENT(obj_grad)) THEN
      obj_grad(1:2) = 0.0_wp
      obj_grad(3) = 3.0E-6_wp*x(3)**2
      obj_grad(4) = 2.0E-6_wp*x(4)**2
    END IF
  END SUBROUTINE obj_fun

  SUBROUTINE con_fun(first_call,final_call,x,continue,finish,con_f, &
     con_jac,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC COS, PRESENT, SIN
    ! .. Parameters ..
    REAL (wp), PARAMETER :: quarter = 0.25_wp
    REAL (wp), PARAMETER :: thousand = 1000.0_wp
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: continue, finish
    LOGICAL, INTENT (IN) :: final_call, first_call
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), INTENT (OUT) :: con_f(:)
    REAL (wp), OPTIONAL, INTENT (INOUT) :: con_jac(:)
    REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Executable Statements ..

    con_f(1) = SIN(-x(1)-quarter) + SIN(-x(2)-quarter)
    con_f(2) = SIN(x(1)-quarter) + SIN(x(1)-x(2)-quarter)
    con_f(3) = SIN(x(2)-quarter) + SIN(x(2)-x(1)-quarter)
    con_f = thousand*con_f

    IF (PRESENT(con_jac)) THEN
      con_jac(1) = -COS(-x(1)-quarter)
      con_jac(2) = COS(x(1)-x(2)-quarter) + COS(x(1)-quarter)
      con_jac(3) = -COS(x(2)-x(1)-quarter)
      con_jac(4) = -COS(-x(2)-quarter)
      con_jac(5) = -COS(x(1)-x(2)-quarter)
      con_jac(6) = COS(x(2)-quarter) + COS(x(2)-x(1)-quarter)
      con_jac = thousand*con_jac
    END IF
  END SUBROUTINE con_fun

END MODULE nlp_sparse_ex01_mod

PROGRAM nag_nlp_sparse_ex01

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nlp_sparse_ex01_mod, ONLY : con_fun, obj_fun, wp
  USE nag_nlp_sparse, ONLY : nag_nlp_sparse_sol
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Parameters ..
  INTEGER, PARAMETER :: idummy = -11111, m = 6, n = 4
  INTEGER, PARAMETER :: nname = n + m
  INTEGER, PARAMETER :: nnz = 14, num_nlin_con = 3, num_nlin_jac_var = 2, &
   num_nlin_obj_var = 4, obj_row = 6
  ! .. Local Scalars ..
  INTEGER :: i, icol, jcol
  REAL (wp) :: obj_f
  ! .. Local Arrays ..
  INTEGER :: col_ptr(n+1) = idummy
  INTEGER :: row_index(nnz)
  REAL (wp) :: a(nnz), s(m), s_lower(m), s_upper(m), x(n), x_lower(n), &
   x_upper(n)
  CHARACTER (8) :: names(n+m) = (/ 'Varble 1', 'Varble 2', 'Varble 3', &
   'Varble 4', 'NlnCon 1', 'NlnCon 2', 'NlnCon 3', 'LinCon 1', 'LinCon 2', &
   'Free Row'/)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_nlp_sparse_ex01'

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

  jcol = 1
  col_ptr(jcol) = 1
  DO i = 1, nnz

    ! Element ( row_index(i), icol ) is stored in a( i )

    READ (nag_std_in,*) a(i), row_index(i), icol

    IF (icol<jcol) THEN

      ! Elements not ordered by increasing column index.

      WRITE (nag_std_out,*) 'Element in column', icol, &
       ' found after element in column', jcol, '. Problem abandoned.'
      STOP

    ELSE IF (icol==jcol+1) THEN

      ! Index in a of the start of the icol-th column equals i.

      col_ptr(icol) = i
      jcol = icol

    ELSE IF (icol>jcol+1) THEN

      ! Index in a of the start of the icol-th column equals i,
      ! but columns jcol+1,jcol+2,...,icol-1 are empty. Set the
      ! corresponding elements of col_ptr to i.

      col_ptr(jcol+1:icol) = i
      jcol = icol
    END IF
  END DO

  col_ptr(n+1) = nnz + 1

  IF (n>icol) THEN
```

```
        ! Columns n,n-1,...,icol+1 are empty. Set the corresponding
        ! elements of col_ptr accordingly.

        DO i = n, icol + 1, -1
          IF (col_ptr(i)==idummy) col_ptr(i) = col_ptr(i+1)
        END DO

      END IF

      READ (nag_std_in,*) (x_lower(i),i=1,n)
      READ (nag_std_in,*) (x_upper(i),i=1,n)
      READ (nag_std_in,*) (s_lower(i),i=1,m)
      READ (nag_std_in,*) (s_upper(i),i=1,m)
      READ (nag_std_in,*) (x(i),i=1,n)

      ! Solve the problem

      CALL nag_nlp_sparse_sol(x,s,obj_f,obj_fun=obj_fun, &
       num_nlin_con=num_nlin_con,num_nlin_obj_var=num_nlin_obj_var, &
       num_nlin_jac_var=num_nlin_jac_var,obj_row=obj_row,con_fun=con_fun,a=a, &
       row_index=row_index,col_ptr=col_ptr,names=names,x_lower=x_lower, &
       x_upper=x_upper,s_lower=s_lower,s_upper=s_upper)

    END PROGRAM nag_nlp_sparse_ex01
```

2 Program Data

```
Example Program Data for nag_nlp_sparse_ex01
 1.0E+25   1   1
 1.0E+25   2   1
 1.0E+25   3   1
     1.0   5   1
    -1.0   4   1
 1.0E+25   1   2
 1.0E+25   2   2
 1.0E+25   3   2
    -1.0   5   2
     1.0   4   2
     3.0   6   3
    -1.0   1   3
    -1.0   2   4
     2.0   6   4                                       : End of a
  -0.55   -0.55     0.0     0.0                        : End of x_lower
   0.55    0.55  1200.0  1200.0                        : End of x_upper
-894.8  -894.8  -1294.8    -0.55    -0.55    -1.0E+25 : End of s_lower
-894.8  -894.8  -1294.8     1.0E+25  1.0E+25  1.0E+25 : End of s_upper
   0.0     0.0      0.0     0.0                        : End of x
```

3 Program Results

```
 Example Program Results for nag_nlp_sparse_ex01

 Parameters
 ----------

 Printing.
 list..................    .true.     lt80_char.............    .true.
 unit..................         6     major_print_level......       10
 minor_print_level......        0
```

```
Derivative approximation.
obj_deriv.............    .true.     con_deriv.............    .true.
fwd_diff_int.......... 5.48E-07     cent_diff_int......... 6.69E-05

Derivative verification.
cheap_test............    .true.

Frequencies.
check_freq............        60     expand_freq...........     10000
fac_freq..............        50

QP subproblems.
scale_tol............. 9.00E-01     minor_feas_tol........ 1.49E-08
scale_opt.............         1     minor_opt_tol......... 1.49E-08
part_price............         1     crash_tol............. 1.00E-01
pivot_tol............. 3.25E-11     elastic_wt............ 1.00E+02
crash_opt.............         0

The SQP method.
minimize..............    .true.     superbasics_lim.......         4
num_nlin_obj_var......         4     major_opt_tol......... 1.49E-08
func_prec............. 3.00E-13     unbounded_step_size... 1.00E+20
deriv_linesearch......    .true.     unbounded_obj......... 1.00E+15
major_step_lim........ 2.00E+00     major_iter_lim........      1000
linesearch_tol........ 9.00E-01     minor_iter_lim........       500
inf_bound............. 1.00E+20     iter_lim..............     10000

Hessian approximation.
hess_lim_mem..........    .true.     hess_upd..............        20
hess_freq............. 99999999

Nonlinear constraints.
num_nlin_con..........         3     major_feas_tol........ 1.49E-08
num_nlin_jac_var......         2     violation_lim......... 1.00E+01

Miscellaneous.
variables.............         4     linear constraints....         3
nonlinear variables...         4     linear variables......         0
lu_den_tol............ 6.00E-01     lu_fac_tol............ 5.00E+00
lu_sing_tol........... 3.25E-11     lu_upd_tol............ 5.00E+00
eps (machine precision) 2.22E-16     cold_start............    .true.
feas_exit.............   .false.     obj_row...............         6
work_factor........... 3.00E+00

Itn      0 -- scale_opt reduced from                1 to                0.

Itn      0 -- Feasible linear rows.

Itn      0 -- Norm(x-x0) minimized. Sum of infeasibilities = 0.00E+00.

con_fun  sets       6   out of       6   constraint gradients.
obj_fun  sets       4   out of       4   objective  gradients.

Cheap test on con_fun...

The Jacobian seems to be OK.

The largest discrepancy was    5.83E-08  in constraint     3.

Cheap test on obj_fun...

The objective gradients seem to be OK.
```

```
Gradient projected in two directions   0.00000000000E+00   0.00000000000E+00
Difference approximations              3.03146585985E-19   7.81305633981E-21

Itn      0 -- All-slack basis B = I selected.

Itn      7 -- Large multipliers.
              Elastic mode started with weight =  2.0E+02.


 Maj  Mnr    Step Merit Function Feasibl Optimal Cond Hz PD
   0   12 0.0E+00   3.199952E+05 1.7E+00 8.0E-01 2.1E+06 FF  R   i
   1    2 1.0E+00   2.463016E+05 1.2E+00 3.2E+03 4.5E+00 FF s
   2    1 1.0E+00   1.001802E+04 3.3E-02 9.2E+01 4.5E+00 FF
   3    1 1.0E+00   5.253418E+03 6.6E-04 2.5E+01 4.8E+00 FF
   4    1 1.0E+00   5.239444E+03 2.0E-06 2.8E+01 1.0E+02 FF
   5    1 1.0E+00   5.126208E+03 6.0E-04 5.9E-01 1.1E+02 FF
   6    1 1.0E+00   5.126498E+03 4.7E-07 2.9E-02 1.0E+02 FF
   7    1 1.0E+00   5.126498E+03 5.9E-10 1.5E-03 1.1E+02 TF
   8    1 1.0E+00   5.126498E+03 1.2E-12 7.6E-09 1.1E+02 TT

Exit from nag_nlp_sparse_sol after     8 major iterations,
                                      21 minor iterations.

Variable State     Value       Lower Bound  Upper Bound   Lagr Mult  Residual

Varble 1    BS   0.118876     -0.55000      0.55000     -1.2529E-07  0.4311
Varble 2    BS  -0.396234     -0.55000      0.55000      1.9243E-08  0.1538
Varble 3    BS    679.945          .         1200.0      1.7001E-10   520.1
Varble 4   SBS    1026.07          .         1200.0     -2.1918E-10   173.9

Constrnt State     Value       Lower Bound  Upper Bound   Lagr Mult  Residual

NlnCon 1    EQ   -894.800      -894.80      -894.80      -4.387      3.3646E-09
NlnCon 2    EQ   -894.800      -894.80      -894.80      -4.106      6.0049E-10
NlnCon 3    EQ   -1294.80      -1294.8      -1294.8      -5.463      3.3556E-09
LinCon 1    BS  -0.515110     -0.55000         None           .      3.4890E-02
LinCon 2    BS   0.515110     -0.55000         None           .       1.065
Free Row    BS    4091.97          None         None     -1.000       4092.

Optimal solution found.

Final objective value =     5126.498
```

Example 2: Nonlinear Programming Problem (with bounds but no general constraints)

This is Problem 45 from Hock and Schittkowski [10] and involves the minimization of the nonlinear function

$$\frac{1}{120} \times x_1x_2x_3x_4x_5$$

subject to the bounds

$$\begin{array}{ccccc} 0 & \leq & x_1 & \leq & 1 \\ 0 & \leq & x_2 & \leq & 2 \\ 0 & \leq & x_3 & \leq & 3 \\ 0 & \leq & x_4 & \leq & 4 \\ 0 & \leq & x_5 & \leq & 5. \end{array}$$

The initial point, which is infeasible, is

$$x^{(0)} = (2,\ 2,\ 2,\ 2,\ 2)^T.$$

The optimal solution is

$$x^* = (1,\ 2,\ 3,\ 4,\ 5)^T,$$

and $F(x^*) = 1$. All the bounds 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 nlp_sparse_ex02_mod
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  SUBROUTINE obj_fun(first_call,final_call,x,continue,finish,obj_f, &
     obj_grad,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC PRESENT, PRODUCT
    ! .. Scalar Arguments ..
    REAL (wp), INTENT (OUT) :: obj_f
    LOGICAL, INTENT (INOUT) :: continue, finish
    LOGICAL, INTENT (IN) :: final_call, first_call
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), OPTIONAL, INTENT (INOUT) :: obj_grad(:)
    REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Local Scalars ..
    REAL (wp) :: sixty = 60.0_wp
    REAL (wp) :: two = 2.0_wp
    ! .. Executable Statements ..

    obj_f = two - PRODUCT(x)/(two*sixty)

    IF (PRESENT(obj_grad)) THEN
      obj_grad(1) = -PRODUCT(x(2:5))
      obj_grad(2) = -x(1)*PRODUCT(x(3:5))
      obj_grad(3) = -x(1)*x(2)*x(4)*x(5)
      obj_grad(4) = -x(5)*PRODUCT(x(1:3))
      obj_grad(5) = -PRODUCT(x(1:4))
      obj_grad = obj_grad/(two*sixty)
    END IF
  END SUBROUTINE obj_fun

END MODULE nlp_sparse_ex02_mod

PROGRAM nag_nlp_sparse_ex02

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

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nlp_sparse_ex02_mod, ONLY : obj_fun, wp
  USE nag_nlp_sparse, ONLY : nag_nlp_sparse_sol, &
   nag_nlp_sparse_cntrl_init, nag_nlp_sparse_cntrl_wp => &
   nag_nlp_sparse_cntrl_dp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Parameters ..
  INTEGER, PARAMETER :: m = 1, n = 5
  INTEGER, PARAMETER :: nname = n + m
  INTEGER, PARAMETER :: num_nlin_obj_var = 5
  ! .. Local Scalars ..
  INTEGER :: i
  REAL (wp) :: obj_f
  TYPE (nag_nlp_sparse_cntrl_wp) :: control
  ! .. Local Arrays ..
  REAL (wp) :: s(m), x(n), x_lower(n), x_upper(n)
  CHARACTER (8) :: names(n+m) = (/ 'Varble 1', 'Varble 2', 'Varble 3', &
   'Varble 4', 'Varble 5', 'DummyRow'/)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_nlp_sparse_ex02'

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

  READ (nag_std_in,*) (x_lower(i),i=1,n)
  READ (nag_std_in,*) (x_upper(i),i=1,n)
  READ (nag_std_in,*) (x(i),i=1,n)

  ! Initialize control structure and set required control parameters

  CALL nag_nlp_sparse_cntrl_init(control)

  control%major_iter_lim = 25
  control%minor_iter_lim = 10
  control%major_step_lim = 5.0_wp

  ! Solve the problem

  CALL nag_nlp_sparse_sol(x,s,obj_f,obj_fun=obj_fun, &
   num_nlin_obj_var=num_nlin_obj_var,names=names,x_lower=x_lower, &
```

```
      x_upper=x_upper,control=control)

   END PROGRAM nag_nlp_sparse_ex02
```

2 Program Data

```
Example Program Data for nag_nlp_sparse_ex02
 0.0  0.0  0.0  0.0  0.0 : End of x_lower
 1.0  2.0  3.0  4.0  5.0 : End of x_upper
 2.0  2.0  2.0  2.0  2.0 : End of x
```

3 Program Results

```
 Example Program Results for nag_nlp_sparse_ex02

 Parameters
 ----------

 Printing.
 list..................    .true.     lt80_char.............    .true.
 unit..................         6     major_print_level......       10
 minor_print_level......        0

 Derivative approximation.
 obj_deriv.............    .true.     con_deriv.............    .true.
 fwd_diff_int..........  5.48E-07     cent_diff_int.........  6.69E-05

 Derivative verification.
 cheap_test............    .true.

 Frequencies.
 check_freq............        60     expand_freq...........     10000
 fac_freq..............       100

 QP subproblems.
 scale_tol.............  9.00E-01     minor_feas_tol........  1.49E-08
 scale_opt.............         2     minor_opt_tol.........  1.49E-08
 part_price............        10     crash_tol.............  1.00E-01
 pivot_tol.............  3.25E-11     elastic_wt............  1.00E+00
 crash_opt.............         3

 The SQP method.
 minimize..............    .true.     superbasics_lim........        5
 num_nlin_obj_var.......        5     major_opt_tol.........  1.49E-08
 func_prec.............  3.00E-13     unbounded_step_size....  1.00E+20
 deriv_linesearch.......    .true.     unbounded_obj.........  1.00E+15
 major_step_lim.........  5.00E+00     major_iter_lim.........       25
 linesearch_tol.........  9.00E-01     minor_iter_lim.........       10
 inf_bound.............  1.00E+20     iter_lim..............     10000

 Hessian approximation.
 hess_lim_mem..........    .true.     hess_upd..............        20
 hess_freq.............  99999999

 Nonlinear constraints.
 num_nlin_con..........         0     num_nlin_jac_var.......        0

 Miscellaneous.
 variables.............         5     linear constraints.....        1
 nonlinear variables....        5     linear variables.......        0
 lu_den_tol............  6.00E-01     lu_fac_tol............  1.00E+02
```

```
lu_sing_tol...........  3.25E-11     lu_upd_tol............  1.00E+01
eps (machine precision)  2.22E-16     cold_start............    .true.
feas_exit.............   .false.     obj_row...............        -1
work_factor...........  3.00E+00

Itn      0 -- part_price reduced from              10 to                1.

Itn      0 -- Feasible linear rows.

Itn      0 -- Norm(x-x0) minimized. Sum of infeasibilities = 0.00E+00.

obj_fun  sets       5   out of       5   objective  gradients.

Cheap test on obj_fun...

The objective gradients seem to be OK.
Gradient projected in two directions   1.16666666667E-01  -3.46944695195E-18
Difference approximations             1.16666602172E-01   1.46827081870E-08


 Maj  Mnr    Step      Objective Optimal Cond Hz PD
   0    3 0.0E+00   1.866667E+00 3.3E-02 1.0E+00 TF  R
   1    2 1.5E+01   1.550000E+00 7.5E-02 1.0E+00 TF n
   2    2 6.7E+00   1.200000E+00 1.0E-01 1.0E+00 TF n
   3    1 5.0E+00   1.000000E+00 0.0E+00 1.0E+00 TT n

Exit from nag_nlp_sparse_sol after     3 major iterations,
                                      8 minor iterations.

Variable State     Value       Lower Bound  Upper Bound   Lagr Mult  Residual

Varble 1    UL    1.00000           .        1.0000      -1.000         .
Varble 2    UL    2.00000           .        2.0000     -0.5000         .
Varble 3    UL    3.00000           .        3.0000     -0.3333         .
Varble 4    UL    4.00000           .        4.0000     -0.2500         .
Varble 5    UL    5.00000           .        5.0000     -0.2000         .

Constrnt State     Value       Lower Bound  Upper Bound   Lagr Mult  Residual

DummyRow    BS    0.00000          None         None     -1.000         .

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_nlp_sparse_ex03`

Solves the nonlinear programming problem described in Section 1 of this module document.

Mathematical Background

1 Overview

`nag_nlp_sparse_sol` is based on the SNOPT package described in Gill *et al.* [1], which in turn utilizes routines from the MINOS package (written in Fortran 77; see Murtagh and Saunders [11]).

At a solution of (1), some of the constraints will be *active*, i.e., satisfied exactly. Let

$$r(x) = \begin{pmatrix} x \\ F(x) \\ Gx \end{pmatrix}$$

and $\mathcal{G}$ denote the set of indices of $r(x)$ corresponding to active constraints at an arbitrary point x. Let $r_j'(x)$ denote the usual *derivative* of $r_j(x)$, which is the row vector of first partial derivatives of $r_j(x)$ (see Ortega and Rheinboldt [12]). The vector $r_j'(x)$ comprises the jth row of $r'(x)$ so that

$$r'(x) = \begin{pmatrix} I \\ J(x) \\ G \end{pmatrix},$$

where $J(x)$ is the Jacobian of $F(x)$.

A point x is a *first-order Kuhn–Karesh–Tucker (KKT) point* for (1) (see, e.g., Powell [13]) if the following conditions hold:

(a) x is feasible;

(b) there exists a vector λ (*the Lagrange multiplier vector for the bound and general constraints*) such that

$$g(x) = r'(x)^T\lambda = (I \;\; J(x)^T \;\; G^T)\lambda, \tag{4}$$

where g is the gradient of f evaluated at x;

(c) the Lagrange multiplier λ_j associated with the jth constraint satisfies $\lambda_j = 0$ if $l_j < r_j(x) < u_j$; $\lambda_j \geq 0$ if $l_j = r_j(x)$; $\lambda_j \leq 0$ if $r_j(x) = u_j$; and λ_j can have any value if $l_j = u_j$.

An equivalent statement of the condition (4) is

$$Z^Tg(x) = 0,$$

where Z is a matrix defined as follows. Consider the set N of vectors orthogonal to the gradients of the active constraints, i.e.,

$$N = \left\{ z \;\mid\; r_j'(x)z = 0 \;\text{ for all }\; j \in \mathcal{G} \right\}.$$

The columns of Z may then be taken as any basis for the vector space N. The vector Z^Tg is termed the *reduced gradient* of f at x. Certain additional conditions must be satisfied in order for a first-order KKT point to be a solution of (1) (see, e.g., Powell [13]).

The basic structure of `nag_nlp_sparse_sol` involves *major* and *minor* iterations. The major iterations generate a sequence of iterates $\{x_k\}$ that satisfy the linear constraints and converge to a point x^* that satisfies the first-order KKT optimality conditions. At each iterate a QP subproblem is used to generate a search direction towards the next iterate (x_{k+1}). The constraints of the subproblem are formed from the linear constraints $Gx - s_L = 0$ and the nonlinear constraint linearization

$$F(x_k) + F'(x_k)(x - x_k) - s_N = 0,$$

where $F'(x_k)$ denotes the *Jacobian matrix*, whose rows are the first partial derivatives of $F(x)$ evaluated at the point x_k. The QP constraints therefore comprise the m linear constraints

$$\begin{array}{rcccl} F'(x_k)x & - & s_N & & = & -F(x_k) + F'(x_k)x_k, \\ Gx & & & - \; s_L & = & 0, \end{array}$$

where x and $s = (s_N, s_L)^T$ are bounded above and below by u and l as before. If the m by n matrix A and m element vector b are defined as

$$A = \begin{pmatrix} F'(x_k) \\ G \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} -F(x_k) + F'(x_k)x_k \\ 0 \end{pmatrix},$$

then the QP subproblem can be written as

$$\underset{x,s}{\text{minimize}}\ q(x) \ \ \text{subject to} \ \ Ax - s = b, \ \ l \le \begin{Bmatrix} x \\ s \end{Bmatrix} \le u, \tag{5}$$

where $q(x)$ is a quadratic approximation to a modified Lagrangian function (see Gill *et al.* [1]).

The linear constraint matrix A is stored in the arrays `a`, `row_index` and `col_ptr` (see Section 3.2). This allows you to specify the sparsity pattern of non-zero elements in $F'(x)$ and G, and identify any non-zero elements that remain constant throughout the minimization.

Solving the QP subproblem is itself an iterative procedure, with the *minor* iterations of an SQP method being the iterations of the QP method. At each minor iteration, the constraints $Ax - s = b$ are (conceptually) partitioned into the form

$$Bx_B + Sx_S + Nx_N = b,$$

where the *basis matrix* B is square and non-singular. The elements of x_B, x_S and x_N are called the *basic*, *superbasic* and *nonbasic* variables respectively; they are a permutation of the elements of x and s. At a QP solution, the basic and superbasic variables will lie somewhere between their bounds, while the nonbasic variables will be equal to one of their upper or lower bounds. At each minor iteration, x_S is regarded as a set of independent variables that are free to move in any desired direction, namely one that will improve the value of the QP objective function $q(x)$ or sum of infeasibilities (as appropriate). The basic variables are then adjusted in order to ensure that (x, s) continues to satisfy $Ax - s = b$. The number of superbasic variables (n_S say) therefore indicates the number of degrees of freedom remaining after the constraints have been satisfied. In broad terms, n_S is a measure of *how nonlinear* the problem is. In particular, n_S will always be zero if there are no nonlinear constraints in (1) and $f(x)$ is linear.

If it appears that no improvement can be made with the current definition of B, S and N a nonbasic variable is selected to be added to S and the process is repeated with the value of n_S increased by one. At all stages, if a basic or superbasic variable encounters one of its bounds, the variable is made nonbasic and the value of n_S decreased by one.

Associated with each of the m equality constraints $Ax - s = b$ is a *dual variable* π_i. Similarly, each variable in (x, s) has an associated *reduced gradient* d_j (also known as a *reduced cost*). The reduced gradients for the variables x are the quantities $g - A^T\pi$, where g is the gradient of the QP objective function $q(x)$; and the reduced gradients for the slack variables s are the dual variables π. The QP subproblem (5) is optimal if $d_j \ge 0$ for all nonbasic variables at their lower bounds, $d_j \le 0$ for all nonbasic variables at their upper bounds and $d_j = 0$ for other variables (including superbasics). In practice, an *approximate* QP solution is found by slightly relaxing these conditions on d_j (see the description of `control%minor_opt_tol` in the type definition for **nag_nlp_sparse_cntrl_*wp***).

After a QP subproblem has been solved, new estimates of the solution to (1) are computed using a linesearch on the augmented Lagrangian merit function

$$\mathcal{M}(x, s, \pi) = f(x) - \pi^T(F(x) - s_N) + \tfrac{1}{2}(F(x) - s_N)^T D(F(x) - s_N), \tag{6}$$

where D is a diagonal matrix of penalty parameters. If (x_k, s_k, π_k) denotes the current estimate of the solution and $(\hat{x}, \hat{s}, \hat{\pi})$ denotes the optimal QP solution, the linesearch determines a step α_k (where $0 < \alpha_k \le 1$) such that the new point

$$\begin{pmatrix} x_{k+1} \\ s_{k+1} \\ \pi_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ s_k \\ \pi_k \end{pmatrix} + \alpha_k \begin{pmatrix} \hat{x}_k - x_k \\ \hat{s}_k - s_k \\ \hat{\pi}_k - \pi_k \end{pmatrix}$$

produces a *sufficient decrease* in the merit function (6). When necessary, the penalties in D are increased by the minimum-norm perturbation that ensures descent for $\mathcal{M}$ (see Gill *et al.* [2]). As in **nag_nlp_sol**, s_N is adjusted to minimize the merit function as a function of s prior to the solution of the QP subproblem. Further details can be found in Eldersveld [7] and Gill *et al.* [3].

2 Treatment of Constraint Infeasibilities

`nag_nlp_sparse_sol` makes explicit allowance for infeasible constraints. Infeasible linear constraints are detected first by solving a problem of the form

$$\underset{x,v,w}{\text{minimize}}\ e^T(v+w)\ \text{ subject to }\ l \le \begin{Bmatrix} x \\ Gx - v + w \end{Bmatrix} \le u,\ \ v \ge 0,\ \ w \ge 0, \tag{7}$$

where $e = (1, 1, \ldots, 1)^T$. This is equivalent to minimizing the sum of the general linear constraint violations subject to the simple bounds. (In the linear programming literature, the approach is often called *elastic programming*.)

If the linear constraints are infeasible (i.e., $v \neq 0$ or $w \neq 0$), the procedure terminates without computing the nonlinear functions.

If the linear constraints are feasible, all subsequent iterates will satisfy the linear constraints. (Such a strategy allows linear constraints to be used to define a region in which $f(x)$ and $F(x)$ can be safely evaluated.) The procedure then proceeds to solve (1) as given, using search directions obtained from a sequence of QP subproblems (5). Each QP subproblem minimizes a quadratic model of a certain Lagrangian function subject to linearized constraints. An augmented Lagrangian merit function (6) is reduced along each search direction to ensure convergence from any starting point.

The procedure enters 'elastic' mode if the QP subproblem proves to be infeasible or unbounded (or if the dual variables π for the nonlinear constraints become 'large') by solving a problem of the form

$$\underset{x,v,w}{\text{minimize}}\ \bar{f}(x,v,w)\ \text{ subject to }\ l \le \begin{Bmatrix} x \\ F(x) - v + w \\ Gx \end{Bmatrix} \le u,\ \ v \ge 0,\ \ w \ge 0, \tag{8}$$

where

$$\bar{f}(x,v,w) = f(x) + \gamma e^T(v+w) \tag{9}$$

is called a *composite objective* and γ is a non-negative parameter (the *elastic weight*). If γ is sufficiently large, this is equivalent to minimizing the sum of the nonlinear constraint violations subject to the linear constraints and bounds. A similar l_1 formulation of (1) is fundamental to the Sl_1QP algorithm of Fletcher [8]. See also Conn [6].

References

[1] Gill P E, Murray W and Saunders M A (1997) SNOPT: An SQP algorithm for large-scale constrained optimization *Numerical Analysis Report 97–2* Department of Mathematics, University of California, San Diego

[2] Gill P E, Murray W, Saunders M A and Wright M H (1992) Some theoretical properties of an augmented Lagrangian merit function *Advances in Optimization and Parallel Computing* (ed P M Pardalos) North Holland 101–128

[3] 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

[4] Gill P E, Murray W, Saunders M A and Wright M H (1989) A practical anti-cycling procedure for linearly constrained optimization *Math. Programming* **45** 437–474

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

[6] Conn A R (1973) Constrained optimization using a nondifferentiable penalty function *SIAM J. Numer. Anal.* **10** 760–779

[7] Eldersveld S K (1991) Large-scale sequential quadratic programming algorithms *PhD Thesis* Department of Operations Research, Stanford University, Stanford

[8] Fletcher R (1984) An ℓ_1 penalty method for nonlinear constraints *Numerical Optimization 1984* (ed P T Boggs, R H Byrd and R B Schnabel) SIAM Philadelphia 26–40

[9] Fourer R (1982) Solving staircase linear programs by the simplex method *Math. Prgramming* **23** 274–313

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

[11] Murtagh B A and Saunders M A (1995) MINOS 5.4 User's Guide *Report SOL 83-20* Department of Operations Research, Stanford University

[12] Ortega J M and Rheinboldt W C (1970) *Iterative Solution of Nonlinear Equations in Several Variables* Academic Press

[13] 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