

NAG C Library Function Document

nag_opt_nlp_sparse (e04ugc)

1 Purpose

nag_opt_nlp_sparse (e04ugc) solves sparse nonlinear programming problems.

2 Specification

```
#include <nag.h>
#include <nage04.h>

void nag_opt_nlp_sparse(
    void (*confun)(Integer ncnln, Integer njnln, Integer nznjac,
        const double x[], double conf[], double conjac[],
        Nag_Comm *comm),
    void (*objfun)(Integer nonln, const double x[], double *objf,
        double objgrad[], Nag_Comm *comm),
    Integer n, Integer m, Integer ncnln, Integer nonln, Integer njnln,
    Integer iobj, Integer nzn, double a[], const Integer ha[],
    const Integer ka[], double bl[], double bu[], double xs[],
    Integer *ninf, double *sinf, double *objf, Nag_Comm *comm,
    Nag_E04_Opt *options, NagError *fail)
```

3 Description

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

$$\underset{x \in R^n}{\text{minimize}} \quad f(x) \quad \text{subject to} \quad l \leq \begin{Bmatrix} x \\ F(x) \\ Gx \end{Bmatrix} \leq u, \quad (1)$$

where $x = (x_1, x_2, \dots, x_n)^T$ 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.

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 j th 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 $+\infty$. (See the description of the optional parameter **options.inf_bound** in Section 10.2).

nag_opt_nlp_sparse 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 $s = (s_1, s_2, \dots, s_m)^T$. For example, the linear constraint $5 \leq 2x_1 + 3x_2 \leq +\infty$ is replaced by $2x_1 + 3x_2 - s_1 = 0$, together with the bounded slack $5 \leq s_1 \leq +\infty$. The problem defined by (1) can therefore be re-written in the following equivalent form:

$$\underset{x \in R^n, s \in R^m}{\text{minimize}} \quad f(x) \quad \text{subject to} \quad \begin{Bmatrix} F(x) \\ Gx \end{Bmatrix} - s = 0, \quad l \leq \begin{Bmatrix} x \\ s \end{Bmatrix} \leq u. \quad (2)$$

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 (see Section 9 for more details):

A basic variable is a variable associated with a column of a square non-singular basis matrix B .

A nonbasic variable is a variable that is temporarily fixed at its current value (usually its upper or lower bound).

A superbasic variable is a non basic variable which is not at one of its bounds and which is free to move in any desired direction (namely one that will improve the value of the objective function or reduce the sum of infeasibilities). At each step, basic variables are adjusted depending on the values of superbasic variables.

For example, in the simplex method (see Gill *et al.* (1981)) the elements of x 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 $(B|N)$, 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.

The optional parameter **options.direction** (default value = **Nag_Minimize**) may be used to specify an alternative problem in which $f(x)$ is maximized (setting **options.direction** = **Nag_Maximize**), or to only find a feasible point (setting **options.direction** = **Nag_FeasiblePoint**). If the objective function is nonlinear and all the constraints are linear, F is absent and the problem is said to be *linearly constrained*. 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 routine during the solution process as follows.

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

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

subject to the constraints

$$\begin{aligned} u^2 + v^2 + z &= 2 \\ u^4 + v^4 + w &= 4 \\ 2u + 4v &\geq 0 \end{aligned}$$

and to the bounds

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

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

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 user-supplied subroutines **objfun** and **confun** (see Section 4), 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 **nonln** (see Section 4), and they are the only variables needed in **objfun**. The linear part $3z + 5w$ of the objective is stored in row **iobj** (see Section 4) 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 **njln** (see Section 4), and they are the only

variables needed in **confun**. 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

$$\begin{pmatrix} F(x'') + A_2 y'' \\ A_3 x'' + A_4 y'' \end{pmatrix}, \quad (3)$$

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

$$A = \begin{pmatrix} J(x'') & A_2 \\ A_3 & A_4 \end{pmatrix}$$

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 **confun**. 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**, **ha** and **ka** (see Section 4).

In general, the vectors x' and x'' have different dimensions, but they *must* always overlap, in the sense that the shorter vector should always be 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, z) . In other cases it could be the other way round. Note that in some cases it might be necessary to add variables to x' or x'' (whichever is the most convenient), but the first way keeps $J(x'')$ as small as possible. Thus, the nonlinear objective function $f(x')$ may involve either a subset or superset of the variables appearing in the nonlinear constraint functions $F(x'')$, and **nonln** \leq **njnl** (or vice-versa). Sometimes the objective and constraints may really involve *disjoint sets of nonlinear variables*. In such cases the variables should be ordered so that **nonln** $>$ **njnl** and $x' = (x'', x''')$, where the objective is nonlinear in just the last vector x''' . The first **njnl** elements of the gradient array **objgrad** (corresponding to x'') should then be set to zero in **objfun**. This is illustrated in Section 8.

If there are no nonlinear constraints in (1) and $f(x)$ is linear or quadratic, then it may be simpler and/or more efficient to use `nag_opt_sparse_convex_qp` (e04nkc) to solve the resulting linear or quadratic programming problem, or one of `nag_opt_lp` (e04mfc), `nag_opt_lin_lsq` (e04ncc) or `nag_opt_qp` (e04nfc) if G is a *dense* matrix. If the problem is dense and does have nonlinear constraints, then `nag_opt_nlp` (e04ucc) should be used instead.

You must supply an initial estimate of the solution to (1), together with versions of **objfun** and **confun** that define $f(x')$ and $F(x'')$, respectively, and as many first partial derivatives as possible. Note that if there are any nonlinear constraints, then the *first* call to **confun** will precede the *first* call to **objfun**.

`nag_opt_nlp_sparse` is based on the SNOPT package described in Gill *et al.* (1997), which in turn utilizes routines from the MINOS package (see Murtagh and Saunders (1995)). It incorporates a sequential quadratic programming (SQP) method that obtains search directions from a sequence of quadratic programming (QP) subproblems. Each QP subproblem minimizes a quadratic model of a certain Lagrangian function subject to a linearization of the constraints. An augmented Lagrangian merit function is reduced along each search direction to ensure convergence from any starting point. Further details can be found in Section 9.

Throughout this document the symbol ϵ is used to represent the *machine precision* (see `nag_machine_precision` (X02AJC)).

4 Parameters

1: **confun** – function supplied by user *Function*

confun must calculate the vector $F(x)$ of nonlinear constraint functions and (optionally) its Jacobian ($= \partial F / \partial x$) for a specified **njnl** ($\leq n$) element vector x . If there are no nonlinear constraints (i.e., **ncnln** = 0), **confun** will never be called by `nag_opt_nlp_sparse` and the NAG defined null void function pointer, `NULLFN`, can be supplied in the call to `nag_opt_nlp_sparse`. If there are nonlinear constraints, the first call to **confun** will occur before the first call to **objfun**.

The specification of **confun** is:

```
void confun(Integer ncnln, Integer njnln, Integer nnzjac, const double x[],
            double conf[], double conjac[], Nag_Comm *comm)
```

1: **ncnln** – Integer *Input*
On entry: the number of nonlinear constraints. These must be the first **ncnln** constraints in the problem.

2: **njnln** – Integer *Input*
On entry: the number of nonlinear variables. These must be the first **njnln** variables in the problem.

3: **nnzjac** – Integer *Input*
On entry: the number of non-zero elements in the constraint Jacobian. Note that **nnzjac** will always be less than, or equal to, **ncnln**×**njnln**.

4: **x[njnln]** – const double *Input*
On entry: x , the vector of nonlinear Jacobian variables at which the nonlinear constraint functions and/or all available elements of the constraint Jacobian are to be evaluated.

5: **conf[ncnln]** – double *Output*
On exit: if **comm**→**flag** = 0 or 2, **conf**[$i - 1$] must contain the value of $F_i(x)$, the i th nonlinear constraint at x .

6: **conjac[nnzjac]** – double *Input/Output*
On exit: if **comm**→**flag** = 1 or 2, **conjac** must return the available elements of $J(x)$, the constraint Jacobian evaluated at x . These elements must be stored in **conjac** in exactly the same positions as implied by the definitions of the arrays **a**, **ha** and **ka** described below, remembering that $J(x)$ always appears in the top left-hand corner of A . Note that the routine does not perform any internal checks for consistency (except indirectly via the optional parameter **options.verify_grad**), so great care is essential.

If all elements of the constraint Jacobian are known, i.e., the optional parameter **options.con_deriv** = **TRUE** (the default), any constant elements of the Jacobian may be assigned to **a** at the start of the optimization if desired. If an element of **conjac** is not assigned in **confun**, the corresponding value from **a** is used. See also the description for **a** below.

If **options.con_deriv** = **FALSE**, then any available partial derivatives of $c_i(x)$ must be assigned to the elements of **conjac**; the remaining elements *must remain unchanged*. It must be emphasized that, in that case, unassigned elements of **conjac** are not treated as constant; they are estimated by finite differences, at non-trivial expense.

7: **comm** – Nag_Comm *
 Pointer to a structure of type **Nag_Comm**; the following members are relevant to **confun**.

flag – Integer *Input/Output*
On entry: **confun** is called with **comm**→**flag** set to 0, 1 or 2.

If **comm**→**flag** = 0 then only **conf** has to be referenced. If **comm**→**flag** = 1 then only **conjac** has to be referenced. If **comm**→**flag** = 2 then both **conf** and **conjac** are referenced.

On exit: if **confun** resets **comm**→**flag** to -1 , nag_opt_nlp_sparse will terminate with the error indicator **NE_CANNOT_CALCULATE**, unless this occurs during the linesearch; in this case, the linesearch will shorten the step and try again. If **confun** resets **comm**→**flag** to a value smaller or equal to -2 , then nag_opt_nlp_sparse will terminate immediately with the error indicator **NE_USER_STOP**. In both cases, if **fail** is supplied to nag_opt_nlp_sparse **fail.errnum** will be set to the user's setting of **comm**→**flag**.

first – Boolean

Input

On entry: will be set to **TRUE** on the first call to **confun** and **FALSE** for all subsequent calls. This parameter setting allows the user to save computation time if certain data must be read or calculated only once.

last – Boolean

Input

On entry: will be set to **TRUE** on the last call to **confun** and **FALSE** for all other calls. This parameter setting allows the user to perform some additional computation on the final solution.

user – double *

iuser – Integer *

p – Pointer

The type Pointer is void *.

Before calling nag_opt_nlp_sparse these pointers may be allocated memory by the user and initialized with various quantities for use by **confun** when called from nag_opt_nlp_sparse.

Note: **confun** should be tested separately before being used in conjunction with nag_opt_nlp_sparse. The optional parameters **options.verify_grad** and **options.major_iter_lim** can be used to assist this process (see Section 10.2). The array **x** must **not** be changed by **confun**.

If **confun** does not calculate all of the Jacobian constraint elements then the optional parameter **options.con_deriv** should be set to **FALSE**.

2: **objfun** – function supplied by user

Function

objfun must calculate the nonlinear part of the objective $f(x)$ and (optionally) its gradient ($= \partial f / \partial x$) for a specified **nonln** ($\leq n$) element vector x . If there are no nonlinear objective variables (i.e., **nonln** = 0), **objfun** will never be called by nag_opt_nlp_sparse and the NAG defined null void function pointer, NULLFN, can be supplied in the call to nag_opt_nlp_sparse.

The specification for **objfun** is:

```
void objfun(Integer nonln, const double x[], double *objf,
            double objgrad[], Nag_Comm *comm)
```

1: **nonln** – Integer

Input

On entry: the number of nonlinear objective variables. These must be the first **nonln** variables in the problem.

2: **x[nonln]** – const double

Input

On entry: the vector x of nonlinear variables at which the nonlinear part of the objective function and/or all available elements of its gradient are to be evaluated.

- 3: **objf** – double * *Output*
On exit: if **comm**→**flag** = 0 or 2, **objfun** must set **objf** to the value of the nonlinear part of the objective function at x . If it is not possible to evaluate the objective function at x , then **objfun** should assign -1 to **comm**→**flag**; nag_opt_nlp_sparse will then terminate, unless this occurs during the linesearch; in this case, the linesearch will shorten the step and try again.
- 4: **objgrad[nonln]** – double *Output*
On exit: if **comm**→**flag** = 1 or 2, **objgrad** must return the available elements of the gradient $\partial f/\partial x$ evaluated at the current point x .
 If the optional parameter **options.obj_deriv** = **TRUE** (the default), all elements of **objgrad** must be set; if **options.obj_deriv** = **FALSE**, any available elements of the Jacobian matrix must be assigned to the elements of **objgrad**; the remaining elements *must remain unchanged*.
- 5: **comm** – Nag_Comm *
 Pointer to structure of type Nag_Comm; the following members are relevant to **objfun**.
- flag** – Integer *Input/Output*
On entry: **objfun** is called with **comm**→**flag** set to 0, 1 or 2.
 If **comm**→**flag** = 0 then only **objf** has to be referenced. If **comm**→**flag** = 1 then only **objgrad** has to be referenced. If **comm**→**flag** = 2 then both **objf** and **objgrad** are referenced.
On exit: if **objfun** resets **comm**→**flag** to -1 , then nag_opt_nlp_sparse will terminate with the error indicator **NE_CANNOT_CALCULATE**, unless this occurs during the linesearch; in this case, the linesearch will shorten the step and try again. If **objfun** resets **comm**→**flag** to a value smaller or equal to -2 , then nag_opt_nlp_sparse will terminate immediately with the error indicator **NE_USER_STOP**. In both cases, if **fail** is supplied to nag_opt_nlp_sparse **fail.errnum** will then be set to the user's setting of **comm**→**flag**.
- first** – Boolean *Input*
On entry: will be set to **TRUE** on the first call to **objfun** and **FALSE** for all subsequent calls. This parameter setting allows the user to save computation time if certain data must be read or calculated only once.
- last** – Boolean *Input*
On entry: will be set to **TRUE** on the last call to **objfun** and **FALSE** for all other calls. This parameter setting allows the user to perform some additional computation on the final solution.
- nf** – Integer *Input*
On entry: the number of evaluations of the objective function; this value will be equal to the number of calls made to **objfun** including the current one.
- user** – double *
iuser – Integer *
p – Pointer
 The type Pointer is void *.
- Before calling nag_opt_nlp_sparse these pointers may be allocated memory by the user and initialized with various quantities for use by **objfun** when called from nag_opt_nlp_sparse.

Note: **objfun** should be tested separately before being used in conjunction with `nag_opt_nlp_sparse`. The optional parameters **options.verify_grad** and **options.major_iter_lim** can be used to assist this process (see Section 10.2). The array **x** must **not** be changed by **objfun**.

If the function **objfun** does not calculate all of the Jacobian elements then the optional parameter **options.obj_deriv** should be set to **FALSE**.

- 3: **n** – Integer *Input*
On entry: n , the number of variables (excluding slacks). This is the number of columns in the full Jacobian matrix A .
Constraint: $n \geq 1$.
- 4: **m** – Integer *Input*
On entry: m , the number of general constraints (or slacks). This is the number of rows in A , including the free row (if any; see **iobj** below). Note that A must contain at least one row. If your problem has no constraints, or only upper and lower bounds on the variables, then you must include a dummy ‘free’ row consisting of a single (zero) element subject to ‘infinite’ upper and lower bounds. Further details can be found under the descriptions for **iobj**, **nnz**, **a**, **ha**, **ka**, **bl** and **bu** below.
Constraint: $m \geq 1$.
- 5: **ncnln** – Integer *Input*
On entry: the number of nonlinear constraints. These correspond to the leading **ncnln** rows of A .
Constraint: $0 \leq \text{ncnln} \leq m$.
- 6: **nonln** – Integer *Input*
On entry: the number of nonlinear objective variables. If the objective function is nonlinear, the leading **nonln** columns of A belong to the nonlinear objective variables. (See also the description for **njln** below.)
Constraint: $0 \leq \text{nonln} \leq n$
- 7: **njln** – Integer *Input*
On entry: the number of nonlinear Jacobian variables. If there are any nonlinear constraints, the leading **njln** columns of A belong to the nonlinear Jacobian variables. If **nonln** $>$ 0 and **njln** $>$ 0, the nonlinear objective and Jacobian variables overlap. The total number of nonlinear variables is given by $\bar{n} = \max(\text{nonln}, \text{njln})$.
Constraints:

$$\text{njln} = 0 \text{ when } \text{ncnln} = 0,$$

$$1 \leq \text{njln} \leq n \text{ when } \text{ncnln} > 0.$$
- 8: **iobj** – Integer *Input*
On entry: if **iobj** $>$ **ncnln**, row **iobj** of A is a free row containing the non-zero elements of the linear part of the objective function. If **iobj** = 0, there is no free row. If **iobj** = -1, there is a dummy ‘free’ row.
Constraints:

$$\text{iobj} \geq -1,$$

$$\text{ncnln} < \text{iobj} \leq m \text{ when } \text{iobj} > 0.$$

- 9: **nnz** – Integer *Input*
- On entry:* the number of non-zero elements in A (including the Jacobian for any nonlinear constraints, J). If **iobj** = -1, set **nnz** = 1.
- Constraint:* $1 \leq \mathbf{nnz} \leq \mathbf{n} \times \mathbf{m}$.
- 10: **a[nnz]** – double *Input/Output*
- On entry:* the non-zero elements of the Jacobian matrix A , ordered by increasing column index. Note that elements with the same row and column index are not allowed. 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 **iobj** 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. If the optional parameter **options.con_deriv** = **TRUE** (the default), the nonlinear part may also be used to store any constant Jacobian elements. Note that if **confun** does not define the constant Jacobian element **conjac**[i], the missing value will be obtained directly from the corresponding element of **a**. The linear part must contain the non-zero elements of G and the free row (if any). If **iobj** = -1, set **a**[0] = β , say, where $|\beta| < \mathit{bigbnd}$ and bigbnd is the value of the optional parameter **options.inf_bound** (default value = 10^{20}). Elements with the same row and column indices are not allowed. (See also the descriptions for **ha** and **ka** below.)
- On exit:* elements in the nonlinear part corresponding to nonlinear Jacobian variables are overwritten.
- 11: **ha[nnz]** – const Integer *Input*
- On entry:* **ha**[i] must contain the row index of the non-zero element stored in **a**[i], for $i = 0, 1, \dots, \mathbf{nnz}-1$. 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 **confun** must define the Jacobian elements in the same order. If **iobj** = -1, set **ha**[0] = 1.
- Constraint:* $1 \leq \mathbf{ha}[i] \leq \mathbf{m}$, for $i = 0, 1, \dots, \mathbf{nnz}-1$.
- 12: **ka[n+1]** – const Integer *Input*
- On entry:* **ka**[$j-1$] must contain the index in **a** of the start of the j th column, for $j = 1, 2, \dots, \mathbf{n}$. To specify the j th column as empty, set **ka**[j] = **ka**[$j-1$]. Note that the first and last elements of **ka** must be such that **ka**[0] = 0 and **ka**[\mathbf{n}] = **nnz**. If **iobj** = -1, set **ka**[j] = 1 for $j = 1, 2, \dots, \mathbf{n}$.
- Constraints:*
- $$\mathbf{ka}[0] = 0,$$
- $$\mathbf{ka}[j-1] \geq 0 \text{ for } j = 2, 3, \dots, \mathbf{n},$$
- $$\mathbf{ka}[\mathbf{n}] = \mathbf{nnz},$$
- $$0 \leq \mathbf{ka}[j] - \mathbf{ka}[j-1] \leq \mathbf{m}, \text{ for } j = 1, 2, \dots, \mathbf{n}.$$
- 13: **bl[n+m]** – double *Input/Output*
- 14: **bu[n+m]** – double *Input/Output*
- On entry:* **bl** must contain the lower bounds l and **bu** the upper bounds u , for all the variables and general constraints, in the following order. The first \mathbf{n} elements of **bl** must contain the bounds on the variables \mathbf{x} , the next **ncnl** elements the bounds for the nonlinear constraints $F(x)$ (if any) and the next $(\mathbf{m}-\mathbf{ncnl})$ elements the bounds for the linear constraints Gx and the free row (if any). To specify a non-existent lower bound (i.e., $l_j = -\infty$), set **bl**[$j-1$] \leq -**options.inf_bound**, and to specify a non-existent upper bound (i.e., $u_j = +\infty$), set **bu**[$j-1$] \geq **options.inf_bound**, where **options.inf_bound** is one of the optional parameters (default value 10^{20} , see Section 10.2). To specify the j th constraint as an *equality*, set **bl**[$j-1$] = **bu**[$j-1$] = β , say, where

$|\beta| < \mathbf{options.inf_bound}$. Note that the lower bound corresponding to $\mathbf{iobj} \neq 0$ must be set to $-\infty$ and stored in $\mathbf{bl}[n+|\mathbf{iobj}|-1]$; similarly, the upper bound must be set to $+\infty$ and stored in $\mathbf{bu}[n+|\mathbf{iobj}|-1]$.

On exit: the elements of \mathbf{bl} and \mathbf{bu} may have been modified internally, but they are restored on exit.

Constraints:

$$\mathbf{bl}[j] \leq \mathbf{bu}[j], \text{ for } j = 0, 1, \dots, n+m-1,$$

$$|\beta| < \mathbf{options.inf_bound} \text{ when } \mathbf{bl}[j] = \mathbf{bu}[j] = \beta,$$

$$\mathbf{bl}[n+|\mathbf{iobj}|-1] \leq -\mathbf{options.inf_bound} \text{ and } \mathbf{bu}[n+|\mathbf{iobj}|-1] \geq \mathbf{options.inf_bound}, \text{ when } ncnln < \mathbf{iobj} \leq m \text{ or } \mathbf{iobj} = -1.$$

15: **xs[n+m]** – double *Input/Output*

On entry: $\mathbf{xs}[j-1]$, for $j = 1, 2, \dots, n$ must contain the initial values of the variables, x . In addition, if a ‘warm start’ is specified by means of the optional parameter **options.start** (see Section 10.2) the elements $\mathbf{xs}[n+i-1]$, for $i = 1, 2, \dots, m$ must contain the initial values of the slack variables, s .

On exit: the final values of the variables and slacks (x, s) .

16: **ninf** – Integer * *Output*

On exit: the number of constraints that lie outside their bounds by more than the value of the optional parameter **options.minor_feas_tol** (default value = $\sqrt{\epsilon}$).

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, **ninf** 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, **ninf** contains the number of elements of $F(x)$ that lie outside their upper or lower bounds.

17: **sinf** – double * *Output*

On exit: the sum of the infeasibilities of constraints that lie outside their bounds by more than the value of the optional parameter **options.minor_feas_tol** (default value = $\sqrt{\epsilon}$).

If the *linear* constraints are infeasible, **sinf** contains the sum of the infeasibilities of the linear constraints. Otherwise, **sinf** contains the sum of the infeasibilities of the *nonlinear* constraints.

18: **objf** – double * *Output*

On exit: the value of the objective function at the final iterate.

19: **comm** – Nag_Comm *

Structure containing pointers for communication to the user-supplied functions **objfun** and **confun**; see the description of **objfun** and **confun** for details. If the user does not need to make use of this communication feature the null pointer `NAGCOMM_NULL` may be used in the call to `nag_opt_nlp_sparse`; **comm** will then be declared internally for use in calls to user-supplied functions.

20: **options** – Nag_E04_Opt *

A pointer to a structure of type **Nag_E04_Opt** whose members are optional parameters for `nag_opt_nlp_sparse`. These structure members offer the means of adjusting some of the parameter values of the algorithm and on output will supply further details of the results. A description of the members of **options** is given below in Section 10. Some of the results returned in **options** can be used by `nag_opt_nlp_sparse` to perform a ‘warm start’ (see the optional parameter **options.start** in Section 10.2).

If any of these optional parameters are required then the structure **options** should be declared and initialized by a call to `nag_opt_init` (e04xxc) and supplied as an argument to `nag_opt_nlp_sparse`. However, if the optional parameters are not required the NAG defined null pointer, `E04_DEFAULT`, can be used in the function call.

21: **fail** – NagError * *Input/Output*

The NAG error parameter (see the Essential Introduction).

Users are recommended to declare and initialise **fail** and set **fail.print** = **TRUE** for this function.

`nag_opt_nlp_sparse` then returns with **fail.code** = **NE_NO_ERROR** if the iterates have converged to a point x that satisfies the first-order Kuhn-Karesh-Tucker conditions (see Section 9.1) to the accuracy requested by the optional parameters **options.major_feas_tol** (default value = $\sqrt{\epsilon}$) and **options.major_opt_tol** (default value = $\sqrt{\epsilon}$).

4.1 Description of Printed Output

Intermediate and final results are printed out by default. The level of printed output can be controlled by the user with the structure members **options.print_level**, **options.minor_print_level**, and **options.print_80ch** (see Section 10.2 and Section 10.3). The default setting of **options.print_level** = **Nag_Soln_Iter**, **options.print_80ch** = **TRUE**, and **options.minor_print_level** = **Nag_NoPrint** provides a single line of output at each iteration and the final result. This section describes the default printout produced by `nag_opt_nlp_sparse`.

The following line of summary output (≤ 80 characters) is produced at every major 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, <code>Mnr</code> will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 9).
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 (6) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see Section 9.1). As the solution is approached, <code>Merit function</code> will converge to the value of the objective function at the solution. In elastic mode (see Section 9.2), the merit function is a composite function involving the constraint violations weighted by the value of the optional parameter options.elastic_wt (default value = 1.0 or 100.0). If there are no nonlinear constraints present, this entry contains <code>Objective</code> , 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 <code>rowerr</code> , the largest element of the scaled nonlinear constraint residual vector defined in the description of the optional parameter options.major_feas_tol . The solution is regarded as ‘feasible’ if <code>Feasibl</code> is less than (or equal to) the options.major_feas_tol (default value = $\sqrt{\epsilon}$). <code>Feasibl</code> will be approximately zero in the neighbourhood of a solution. If there are no nonlinear constraints present, all iterates are feasible and this entry is not printed.
Optimal	is the value of <code>maxgap</code> , the largest element of the maximum complementarity gap vector defined in the description of the optional parameter options.major_opt_tol . The Lagrange multipliers are regarded as ‘optimal’ if <code>Optimal</code> is less than (or equal to) the optional parameter options.major_opt_tol (default value = $\sqrt{\epsilon}$). <code>Optimal</code> 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 ncnl and nonln are both zero). It is the square of the ratio between the largest and smallest diagonal elements of an upper triangular matrix R . This constitutes a lower bound on the condition number of the matrix $R^T R$ 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 the optional parameters options.major_feas_tol and options.major_opt_tol . Each letter is T if the test is satisfied, and F otherwise. The tests indicate whether the values of <code>Feasibl</code> and <code>Optimal</code> are sufficiently small. For example, TF or TT is printed if there are no nonlinear constraints present (since all iterates are feasible).
M	is printed if an extra evaluation of objfun and confun 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.
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 the optional parameters options.hess_freq (default value = 99999999) and options.hess_update (default value = 20). However, it may also be necessary to reset an ill-conditioned Hessian from time to time.
l	is printed if the change in the variables was limited by the value of the optional parameter options.major_step_lim (default value = 2.0). If this output occurs frequently during later iterations, it may be worthwhile increasing the value of options.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 because the number of iterations specified by the value of the optional parameter options.minor_iter_lim (default value = 500) was reached.
i	is printed if the QP subproblem was infeasible when the routine 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 (8) (see Section 9.2). It is also printed if the minimizer of the elastic subproblem does not satisfy the linearized constraints when the routine 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.

Variable	gives the name of the variable. If the optional parameter options.cnames = NULL, a default name is assigned to the j th variable for $j = 1, 2, \dots, n$. Otherwise, the name supplied in options.cnames [$j - 1$] is assigned to the j th variable.
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 some additional information about the state of a variable. Note that unless the optional parameter options.scale_opt = 0 (default value = 1 or 2) is specified, the tests for assigning a key are applied to the variables of the scaled problem. A <i>Alternative optimum possible</i> . 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 <i>might</i> change, giving a genuine alternative solution. The values of the Lagrange multipliers <i>might</i> also change. D <i>Degenerate</i> . The variable is basic, but it is equal to (or very close to) one of its bounds. I <i>Infeasible</i> . The variable is basic and is currently violating one of its bounds by more than the value of the optional parameter options.minor_feas_tol (default value = $\sqrt{\epsilon}$). N <i>Not precisely optimal</i> . The variable is nonbasic. Its reduced gradient is larger than the value of the optional parameter options.major_feas_tol (default value = $\sqrt{\epsilon}$).
Value	is the value of the variable at the final iterate.
Lower Bound	is the lower bound specified for the variable. None indicates that bl [$j - 1$] \leq -options.inf_bound .
Upper Bound	is the upper bound specified for the variable. None indicates that bu [$j - 1$] \geq options.inf_bound .
Lagr Mult	is the Lagrange multiplier for the associated bound. This will be zero if State is FR. If x is optimal, the multiplier should be non-negative if State is LL, non-positive if State is UL, and zero if State is BS or SBS.
Residual	is the difference between the variable Value and the nearer of its (finite) bounds bl [$j - 1$] and bu [$j - 1$]. A blank entry indicates that the associated variable is not bounded (i.e., bl [$j - 1$] \leq -options.inf_bound and bu [$j - 1$] \geq options.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 , **options.cnames**[$j - 1$] replaced by **options.cnames**[$n + j - 1$], **bl**[$j - 1$] and **bu**[$j - 1$] replaced by **bl**[$n + j - 1$] and **bu**[$n + j - 1$] 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.

5 Error Indicators and Warnings

NE_2_INT_ARG_CONS

On entry, **njnl** = *<value>* while **ncnl** = *<value>*.
These parameters must satisfy **njnl** = 0 when **ncnl** = 0.

NE_2_INT_OPT_ARG_CONS

On entry, **options.obj_check_start** = *<value>* while **options.obj_check_stop** = *<value>*.
These parameters must satisfy **options.obj_check_start** ≤ **options.obj_check_stop**.
(Note that this error may only occur when **options.verify_grad** = **Nag_CheckObj** or **Nag_CheckObjCon**.)

On entry, **options.con_check_start** = *<value>* while **options.con_check_stop** = *<value>*.
These parameters must satisfy **options.con_check_start** ≤ **options.con_check_stop**.
(Note that this error may only occur when **options.verify_grad** = **Nag_CheckCon** or **Nag_CheckObjCon**.)

NE_3_INT_ARG_CONS

On entry, **ncnl** = *<value>*, **iobj** = *<value>* and **m** = *<value>*.
These parameters must satisfy **ncnl** < **iobj** ≤ **m** when **iobj** > 0.

NE_INT_ARRAY_2

Value *<value>* given to **ha**[*<value>*] is not valid.
Correct range for elements of **ha** is 1 to **m**.

NE_ARRAY_CONS

The contents of array **ka** are not valid. Constraint: **ka**[0] = 0.
The contents of array **ka** are not valid. Constraint: **ka**[**n**] = **nnz**.
The contents of array **ka** are not valid. Constraint: $0 \leq \mathbf{ka}[i+1] - \mathbf{ka}[i] \leq \mathbf{m}$, for $0 \leq i < \mathbf{n}$.

NE_INT_ARRAY_1

Value *<value>* given to **ka**[*<value>*] not valid.
Correct range for elements of **ka** is ≥ 0 .

NE_BAD_PARAM

On entry, parameter **options.crash** had an illegal value.
On entry, parameter **options.direction** had an illegal value.
On entry, parameter **options.hess_storage** had an illegal value.
On entry, parameter **options.minor_print_level** had an illegal value.
On entry, parameter **options.print_deriv** had an illegal value.
On entry, parameter **options.print_level** had an illegal value.
On entry, parameter **options.start** had an illegal value.
On entry, parameter **options.verify_grad** had an illegal value.

NE_BOUND

The lower bound for variable *<value>* (array element **bl**[*<value>*]) is greater than the upper bound.

NE_BOUND_EQ

The lower bound and upper bound for variable *<value>* (array elements **bl**[*<value>*] and **bu**[*<value>*]) are equal but they are greater than or equal to **options.inf_bound**.

NE_BOUND_EQ_LCON

The lower bound and upper bound for linear constraint *<value>* (array element **bl**[*<value>*] and **bu**[*<value>*]) are equal but they are greater than or equal to **options.inf_bound**.

NE_BOUND_EQ_NLCON

The lower bound and upper bound for nonlinear constraint $\langle value \rangle$ (array element **bl**[$\langle value \rangle$] and **bu**[$\langle value \rangle$]) are equal but they are greater than or equal to **options.inf_bound**.

NE_BOUND_LCON

The lower bound for linear constraint $\langle value \rangle$ (array element **bl**[$\langle value \rangle$]) is greater than the upper bound.

NE_BOUND_NLCON

The lower bound for non-linear constraint $\langle value \rangle$ (array element **bl**[$\langle value \rangle$]) is greater than the upper bound.

NE_CANNOT_CALCULATE

The objective and/or constraint functions could not be calculated.

NE_CON_DERIV_ERRORS

Subroutine **confun** appears to be giving incorrect gradients.

The user-provided derivatives of the nonlinear constraint functions computed by **confun** appear to be incorrect. Check that **confun** has been coded correctly and that all relevant elements of the nonlinear constraint Jacobian have been assigned their correct values.

NE_DUPLICATE_ELEMENT

Duplicate sparse matrix element found in row $\langle value \rangle$, column $\langle value \rangle$.

NE_INT_ARG_LT

On entry, **n** must not be less than 1: **n** = $\langle value \rangle$.

On entry, **m** must not be less than 1: **m** = $\langle value \rangle$.

On entry, **iobj** must not be less than -1: **iobj** = $\langle value \rangle$.

NE_INT_OPT_ARG_GT

On entry, **options.obj_check_start** = $\langle value \rangle$.

Constraint: **options.obj_check_start** \leq **nonln**.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckObj** or **Nag_CheckObjCon**.)

On entry, **options.obj_check_stop** = $\langle value \rangle$.

Constraint: **options.obj_check_stop** \leq **nonln**.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckObj** or **Nag_CheckObjCon**.)

On entry, **options.con_check_start** = $\langle value \rangle$.

Constraint: **options.con_check_start** \leq **nonln**.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckCon** or **Nag_CheckObjCon**.)

On entry, **options.con_check_stop** = $\langle value \rangle$.

Constraint: **options.con_check_stop** \leq **nonln**.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckCon** or **Nag_CheckObjCon**.)

NE_INT_OPT_ARG_LT

On entry, **options.factor_freq** = $\langle value \rangle$.

Constraint: **options.factor_freq** \geq 0.

On entry, **options.expand_freq** = $\langle value \rangle$.

Constraint: **options.expand_freq** \geq 0.

On entry, **options.obj_check_start** = *<value>*.

Constraint: **options.obj_check_start** \geq 1.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckObj** or **Nag_CheckObjCon**.)

On entry, **options.obj_check_stop** = *<value>*.

Constraint: **options.obj_check_stop** \geq 1.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckObj** or **Nag_CheckObjCon**.)

On entry, **options.con_check_start** = *<value>*.

Constraint: **options.con_check_start** \geq 1.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckCon** or **Nag_CheckObjCon**.)

On entry, **options.con_check_stop** = *<value>*.

Constraint: **options.con_check_stop** \geq 1.

(Note that this error may only occur when **options.verify_grad** = **Nag_CheckCon** or **Nag_CheckObjCon**.)

On entry, **options.fcheck** = *<value>*.

Constraint: **options.fcheck** \geq 0.

NE_INVALID_INT_RANGE_1

Value *<value>* given to **nnz** is not valid.

Correct range is 1 to **n** \times **m**.

Value *<value>* given to **ncnl** is not valid.

Correct range is 0 to **m**.

Value *<value>* given to **nonln** is not valid.

Correct range is 0 to **n**.

Value *<value>* given to **njnl** is not valid.

Correct range is (when **ncnl** $>$ 0) 1 to **n**.

Value *<value>* given to **options.hess_freq** is not valid.

Correct range is **options.hess_freq** $>$ 0.

Value *<value>* given to **options.hess_update** is not valid.

Correct range is **options.hess_update** \geq 0.

Value *<value>* given to **options.iter_lim** is not valid.

Correct range is **options.iter_lim** $>$ 0.

Value *<value>* given to **options.major_iter_lim** is not valid.

Correct range is **options.major_iter_lim** \geq 0.

Value *<value>* given to **options.minor_iter_lim** is not valid.

Correct range is **options.minor_iter_lim** \geq 0.

Value *<value>* given to **options.part_price** is not valid.

Correct range is **options.part_price** $>$ 0.

Value *<value>* given to **options.max_sb** is not valid.

Correct range is **options.max_sb** $>$ 0.

Value *<value>* given to **options.nsb** is not valid.

Correct range is: if **options.start** = **Nag_Warm**, then **options.nsb** \geq 0.

NE_INVALID_INT_RANGE_2

Value *<value>* given to **options.scale_opt** is not valid.

Correct range is $0 \leq$ **options.scale_opt** \leq 2.

NE_INVALID_REAL_RANGE_E

- Value $\langle value \rangle$ given to **options.elastic_wt** is not valid.
Correct range is **options.elastic_wt** > 0.0 .
- Value $\langle value \rangle$ given to **options.inf_bound** is not valid.
Correct range is **options.inf_bound** > 0.0 .
- Value $\langle value \rangle$ given to **options.lu_den_tol** is not valid.
Correct range is **options.lu_den_tol** ≥ 0.0 .
- Value $\langle value \rangle$ given to **options.lu_sing_tol** is not valid.
Correct range is **options.lu_sing_tol** > 0.0 .
- Value $\langle value \rangle$ given to **options.lu_factor_tol** is not valid.
Correct range is **options.lu_factor_tol** ≥ 1.0 .
- Value $\langle value \rangle$ given to **options.lu_update_tol** is not valid.
Correct range is **options.lu_update_tol** ≥ 1.0 .
- Value $\langle value \rangle$ given to **options.major_feas_tol** is not valid.
Correct range is **options.major_feas_tol** $> \epsilon$.
- Value $\langle value \rangle$ given to **options.major_opt_tol** is not valid.
Correct range is **options.major_opt_tol** > 0.0 .
- Value $\langle value \rangle$ given to **options.major_step_lim** is not valid.
Correct range is **options.major_step_lim** > 0.0 .
- Value $\langle value \rangle$ given to **options.minor_feas_tol** is not valid.
Correct range is **options.minor_feas_tol** $> \epsilon$.
- Value $\langle value \rangle$ given to **options.minor_opt_tol** is not valid.
Correct range is **options.minor_opt_tol** > 0.0 .
- Value $\langle value \rangle$ given to **options.nz_coef** is not valid.
Correct range is **options.nz_coef** ≥ 1.0 .
- Value $\langle value \rangle$ given to **options.pivot_tol** is not valid.
Correct range is **options.pivot_tol** > 0.0 .
- Value $\langle value \rangle$ given to **options.unbounded_obj** is not valid.
Correct range is **options.unbounded_obj** > 0.0 .
- Value $\langle value \rangle$ given to **options.inf_step** is not valid.
Correct range is **options.inf_step** > 0.0 .
- Value $\langle value \rangle$ given to **options.violation_limit** is not valid.
Correct range is **options.violation_limit** > 0.0 .

NE_INVALID_REAL_RANGE_EE

- Value $\langle value \rangle$ given to **options.f_diff_int** is not valid.
Correct range is $\epsilon \leq$ **options.f_diff_int** < 1.0 .
- Value $\langle value \rangle$ given to **options.c_diff_int** is not valid.
Correct range is $\epsilon \leq$ **options.c_diff_int** < 1.0 .
- Value $\langle value \rangle$ given to **options.crash_tol** is not valid.
Correct range is $0.0 \leq$ **options.crash_tol** < 1.0 .
- Value $\langle value \rangle$ given to **options.f_prec** is not valid.
Correct range is $\epsilon \leq$ **options.f_prec** < 1.0 .
- Value $\langle value \rangle$ given to **options.linesearch_tol** is not valid.
Correct range is $0.0 \leq$ **options.linesearch_tol** < 1.0 .
- Value $\langle value \rangle$ given to **options.scale_tol** is not valid.
Correct range is $0.0 <$ **options.scale_tol** < 1.0 .

NE_NAME_TOO_LONG

The string pointed to by **options.cnames**[<value>] is too long. It should be no longer than 8 characters.

NE_OBJ_BOUND

Invalid lower bound for objective row. Bound should be \leq **options.inf_bound**.
Invalid upper bound for objective row. Bound should be \geq **options.inf_bound**.

NE_OBJ_DERIV_ERRORS

Subroutine **objfun** appears to be giving incorrect gradients.
The user-provided derivatives of the objective function computed by **objfun** appear to be incorrect. Check that **objfun** has been coded correctly and that all relevant elements of the objective gradient have been assigned their correct values.

NE_OPT_NOT_INIT

Options structure not initialized.

NE_STATE_VAL

options.state[<value>] is out of range. **state**[<value>] = <value>.

NE_SUPERBASICS_LIMIT

Too many superbasic variables (**options.max_sb** = <value>).
The value of the optional parameter **options.max_sb** (default value = $\min(500, \bar{n} + 1, n)$) is too small and should be increased.

NE_BASIS_ILL_COND

Numerical error in trying to satisfy the general constraints. The basis is very ill-conditioned.

NE_BASIS_SINGULAR

The basis is singular after 15 attempts to factorize it (and adding slacks when necessary). Either the problem is badly scaled or the value of the optional parameter **options.lu_factor_tol** (default value = 5.0 or 100.0) is too large.

NE_LIN_NOT_FEASIBLE

No feasible point was found for the linear constraints. Sum of infeasibilities: <value>.
The problem is infeasible. The linear constraints cannot all be satisfied to within the values of the optional parameter **options.minor_feas_tol** (default value = $\sqrt{\epsilon}$).

NE_NONLIN_NOT_FEASIBLE

No feasible point was found for the nonlinear constraints. Sum of infeasibilities: <value>.
The problem is infeasible. The nonlinear constraints cannot all be satisfied to within the values of the optional parameter **options.major_feas_tol** (default value = $\sqrt{\epsilon}$).

NE_MAYBE_UNBOUNDED

Violation limit exceeded. The problem maybe unbounded.
Check the values of the optional parameters **options.unbounded_obj** (default value = 10^{15}) and **options.inf_step** (default value = $\max(\text{bigbnd}, 10^{20})$) 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 the optional parameter **options.violation_limit** (default value = 10.0).

NE_NOT_REQUIRED_ACC

Feasible solution, but required accuracy could not be achieved.

Check that the value of the optional parameter **options.major_opt_tol** (default value = $\sqrt{\epsilon}$) is not too small.

NE_NO_IMPROVE

The current point cannot be improved on.

Check that **objfun** and **confun** have been coded correctly and that they are consistent with the values of the optional parameters **options.obj_deriv** and **options.con_deriv** (default value = **TRUE**).

NE_TOO_MANY_ITER

Iteration limit (**options.iter_lim** = *<value>*) exceeded.

NE_TOO_MANY_MAJOR_ITER

Major iteration limit (**options.major_iter_lim** = *<value>*) exceeded.

NE_UNBOUNDED

Solution appears to be unbounded.

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_opt_nlp_sparse`.

NE_USER_STOP

User requested termination, user flag value = *<value>*.

This exit occurs if the user sets **comm**→**flag** to a negative value in **objfun** or **confun**. If **fail** is supplied the value of **fail.errnum** will be the same as the user's setting of **comm**→**flag**.

NE_ALLOC_FAIL

Memory allocation failed.

NE_NOT_APPEND_FILE

Cannot open file *<string>* for appending.

NE_NOT_CLOSE_FILE

Cannot close file *<string>*.

NE_WRITE_ERROR

Error occurred when writing to file *<string>*.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please consult NAG for assistance.

6 Further Comments

6.1 Termination Criteria

If `nag_opt_nlp_sparse` returns with `fail.code = NE_NO_ERROR`, the iterates have converged to a point x that satisfies the first-order Kuhn-Karesh-Tucker conditions (see Section 9.1) to the accuracy requested by the optional parameters `options.major_feas_tol` (default value = $\sqrt{\epsilon}$) and `options.major_opt_tol` (default value = $\sqrt{\epsilon}$).

6.2 Accuracy

If `options.major_feas_tol` is set to 10^{-d} (default value = $\sqrt{\epsilon}$) and `fail.code = NE_NOERROR` on exit, then the final value of $f(x)$ should have approximately d correct digits.

6.3 References

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

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) 101–128 North Holland

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

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

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

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

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

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) 26–40 SIAM Philadelphia

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

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

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

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

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

7 See Also

`nag_opt_lp` (e04mfc)

`nag_opt_lin_lsq` (e04ncc)

`nag_opt_qp` (e04nfc)

`nag_opt_sparse_convex_qp` (e04nkc)

`nag_opt_nlp` (e04ucc)

`nag_opt_init` (e04xxc)

`nag_opt_read` (e04xyc)

`nag_opt_free` (e04xzc)

8 Example 1

This is a reformulation of Problem 74 from Hock and Schittkowski (1981) and involves minimization of the nonlinear function

$$f(x) = 10^{-6}x_3^3 + \frac{2}{3} \times 10^{-6}x_4^3 + 3x_3 + 2x_4$$

subject to the bounds

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

to the nonlinear constraints

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

and to the linear constraints

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

The initial point, which is infeasible, is

$$x_0 = (0, 0, 0, 0)^T,$$

and $f(x_0) = 0$.

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.

This example shows the simple use of `nag_opt_nlp_sparse` where default values are used for all optional parameters. An example showing the use of optional parameters is given in Section 11. There is one example program file, the main program of which calls both examples. The main program and Example 1 are given below.

The use of the interface to `nag_opt_nlp_sparse` for this particular example is briefly illustrated below. First, note that because of the constraints on the definitions of nonlinear jacobian variables and nonlinear objective variables in the interface to `nag_opt_nlp_sparse`, the first objective variables x_1 and x_2 are considered as nonlinear objective variables. Thus, **nonln** = 4, and there are **njnl** = 2 nonlinear Jacobian variables (x_1 and x_2). (The alternative would have consisted in reordering the problem to have **nonln** = 2 nonlinear objective variables and **njnl** = 4 nonlinear constraint variables, but, as mentioned earlier, it is preferable to keep the size of the nonlinear Jacobian (J) small, having **nonln** > **njnl**.)

The Jacobian matrix A is the **m** = 6 by **n** = 4 matrix below

$$A = \begin{pmatrix} \mathbf{conjac}[0] & \mathbf{conjac}[3] & -1 & 0 \\ \mathbf{conjac}[1] & \mathbf{conjac}[4] & 0 & -1 \\ \mathbf{conjac}[2] & \mathbf{conjac}[5] & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 3 & 2 \end{pmatrix},$$

where zeros are not stored, each column represents a variable, each row a constraint (except the free row), and the **conjac**[i] entries reflect the structure of the Jacobian J corresponding to the nonlinear constraints.

The first 3 rows correspond to the **ncnln** = 3 nonlinear constraints, rows 4 and 5 define the 2 linear constraints and there is finally an **iobj**=6th free row defining the linear part of the objective function, $3x_3 + 2x_4$.

A contains **nnz** = 14 nonzero elements of which 6 entries define the structure of J . In this case all entries in J are defined in the supplied function **confun** and there is no constant value that we want to pass only once via A , so all entries in the corresponding array **a** corresponding to J can just be initialized to dummy values (here $1.0e+25$). Effective Jacobian values will be provided in the parameter **conjac**[$i - 1$], $i = 1, 2, \dots, \mathbf{nnzjac}$, for **nnzjac**=6, in the routine **confun**. Note also that in this simple example, J is indeed full; otherwise, the structure of A should reflect the sparsity of J .

This example includes source code to store the matrix A in the arrays **a**, **ha**, **ka**, based on the simple format from the data file.

Finally, the lower and upper bounds are defined by

$$\mathbf{bl} = (-0.55, -0.55, 0.0, 0.0, -894.6, -894.6, -1294.8, -0.55, -0.55, -1.0e + 25)^T, \text{ and}$$

$$\mathbf{bu} = (0.55, 0.55, 1200.0, 1200.0, -894.6, -894.6, -1294.8, 1.0e + 25, 1.0e + 25, 1.0e + 25)^T.$$

The first **n** = 4 elements of **bl** and **bu** are simple bounds on the variables; the next 3 elements are bounds on the nonlinear constraints; the next 2 elements are bounds on the linear constraints; and finally, the last (unbounded) element corresponds to the free row.

8.1 Program Text

```

/* nag_nlp_sparse(e04ugc) Example Program.
 *
 * Copyright 2000 Numerical Algorithms Group.
 *
 * NAG C Library
 *
 * Mark 6, 2000.
 *
 */

#include <nag.h>
#include <stdio.h>
#include <math.h>
#include <nag_stdlib.h>
#include <nage04.h>

static void confun(Integer ncnln, Integer njnln,
                  Integer nnzjac, const double x[], double conf[],
                  double conjac[], Nag_Comm *comm);

static void objfun(Integer nonln,
                  const double x[], double * objf,
                  double objgrad[], Nag_Comm *comm);

static int ex1(void);
static int ex2(void);

int main()
{
    /* Two examples are called: ex1() uses the
     * default settings to solve a problem while
     * ex2() solves the same problem with some
     * of the optional parameters set by the user,
     * perturbs the solution and then uses the
     * warm start facility.
     */
    Vprintf("e04ugc Example Program Results\n");
    if (ex1() != 0) return 1;

```

```

    if (ex2() != 0) return 1;
    return 0;
}

static int ex1(void)
{
    double *a=0, *b1=0, *bu=0, *xs=0;
    Integer *ha=0, *ka=0;

    Integer exit_status=0;

    double obj, sinf;
    Integer m, n, nnz, ncnln, njnln, nonln;
    Integer ninf;
    Integer i, icol, j, jcol;
    Integer iobj;
    static NagError fail;

    Vprintf("\nExample 1: default options used.\n");
    Vscanf("%*[^\\n]"); /* Skip heading in data file. */
    Vscanf("%*[^\\n]");

    fail.print = TRUE;

    /* Read the problem dimensions */
    Vscanf("%*[^\\n]");
    Vscanf("%ld%ld", &n, &m);

    /* Read NCNLN, NONLN and NJNLN from data file. */
    Vscanf("%*[^\\n]");
    Vscanf("%ld%ld%ld", &ncnln, &nonln, &njnln);

    /* Read NNZ, IOBJ */
    Vscanf("%*[^\\n]");
    Vscanf("%ld%ld", &nnz, &iobj);

    if ( ! (a = NAG_ALLOC(nnz, double)) ||
        ! (b1 = NAG_ALLOC(n+m, double)) ||
        ! (bu = NAG_ALLOC(n+m, double)) ||
        ! (xs = NAG_ALLOC(n+m, double)) ||
        ! (ha = NAG_ALLOC(nnz, Integer)) ||
        ! (ka = NAG_ALLOC(n+1, Integer)) )
    {
        Vprintf("Allocation failure\n");
        exit_status = 1;
        goto END;
    }
    /* read the matrix and set up ka. */
    jcol = 1;
    ka[jcol - 1] = 0;
    Vscanf("%*[^\\n]");
    for (i = 0; i < nnz; ++i)
    {
        /* a[i] stores (ha[i], icol) element of matrix */
        Vscanf("%lf%ld%ld", &a[i], &ha[i], &icol);
        if (icol < jcol)
            {

```

```

        /* Elements not ordered by increasing column index. */
        Vprintf("%s%5ld%s%5ld%s\n", "Element in column", icol,
                " found after element in column", jcol, ". Problem aban-
done.");
        exit_status = 1;
        goto END;
    }
    else if (icol == jcol + 1)
    {
        /* Index in a of the start of the icol-th column equals i. */
        ka[icol - 1] = i;
        jcol = icol;
    }
    else if (icol > jcol + 1)
    {
        /* 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 ka to i.
        */
        for (j = jcol + 1; j <= icol - 1; ++j)
            ka[j - 1] = i;

        ka[icol - 1] = i;
        jcol = icol;
    }
}
ka[n] = nnz;
if (n > icol)
{
    /* Columns N,N-1,...,ICOL+1 are empty. Set the
    * corresponding elements of ka accordingly. */
    for (j = icol; j <= n - 1; ++j)
        ka[j] = nnz;
}

/* Read the bounds */
Vscanf("%*[\n]");
for (i = 0; i < n + m; ++i)
    Vscanf("%lf", &bl[i]);
Vscanf("%*[\n]");
for (i = 0; i < n + m; ++i)
{
    Vscanf("%lf", &bu[i]);
}

/* Read the initial estimate of x */
Vscanf("%*[\n]");
for (i = 0; i < n; ++i)
{
    Vscanf("%lf", &xs[i]);
}
Vscanf("%*[\n]");

/* Solve the problem. */
e04ugc (confun, objfun, n, m,
        ncnln, nonln, njnln, iobj, nnz,
        a, ha, ka, bl, bu, xs,
        &ninf, &sinf, &obj, NAGCOMM_NULL,

```

```

        E04_DEFAULT, &fail);

END:
    if ( a) NAG_FREE(a);
    if (b1) NAG_FREE(b1);
    if (bu) NAG_FREE(bu);
    if (xs) NAG_FREE(xs);
    if (ha) NAG_FREE(ha);
    if (ka) NAG_FREE(ka);
    return exit_status;
}

static void confun(Integer ncnln, Integer njnln,
                  Integer nnzjac, const double x[], double conf[],
                  double conjac[], Nag_Comm *comm)
{
#define CONJAC(I) conjac[(I)-1]
#define CONF(I) conf[(I)-1]
#define X(I) x[(I)-1]
    /* Compute the nonlinear constraint functions and their Jacobian. */
    if (comm->flag == 0 || comm->flag == 2)
    {
        CONF(1) = sin (-X(1) - .25) * 1e3 + sin (-X(2) - .25) * 1e3;
        CONF(2) = sin (X(1) - .25) * 1e3 + sin (X(1) - X(2) - .25) * 1e3;
        CONF(3) = sin (X(2) - X(1) - .25) * 1e3 + sin (X(2) - .25) * 1e3;
    }
    if (comm->flag == 1 || comm->flag == 2)
    {
        /* Nonlinear Jacobian elements for column 1.0 */
        CONJAC(1) = cos (-X(1) - .25) * -1e3;
        CONJAC(2) = cos (X(1) - .25) * 1e3 + cos (X(1) - X(2) - .25) * 1e3;
        CONJAC(3) = cos (X(2) - X(1) - .25) * -1e3;
        /* Nonlinear Jacobian elements for column 2.0 */
        CONJAC(4) = cos (-X(2) - .25) * -1e3;
        CONJAC(5) = cos (X(1) - X(2) - .25) * -1e3;
        CONJAC(6) = cos (X(2) - X(1) - .25) * 1e3 + cos (X(2) - .25) * 1e3;
    }
}

static void objfun(Integer nonln,
                  const double x[], double *objf,
                  double objgrad[], Nag_Comm *comm)
{
#define OBJGRAD(I) objgrad[(I)-1]
#define X(I) x[(I)-1]
    /* Compute the nonlinear part of the objective function and its grad */
    if (comm->flag == 0 || comm->flag == 2)
    {
        *objf = X(3) * X(3) * X(3) * 1e-6 + X(4) * X(4) * X(4) * 2e-6 / 3.0;
    }
    if (comm->flag == 1 || comm->flag == 2)
    {
        OBJGRAD(1) = 0.0;
        OBJGRAD(2) = 0.0;
        OBJGRAD(3) = X(3) * X(3) * 3e-6;
        OBJGRAD(4) = X(4) * X(4) * 2e-6;
    }
}

```


8.2 Program Data

e04ugc Example Program Data

Data for example 1.

Values of n and m

4 6

Values of ncnln, nonln and njnln

3 4 2

Values of nnz and iobj

14 6

Matrix nonzeros: value, row index, column index

```

1.0E+25  1  1
1.0E+25  2  1
1.0E+25  3  1
   -1.0  4  1
    1.0  5  1
1.0E+25  1  2
1.0E+25  2  2
1.0E+25  3  2
    1.0  4  2
   -1.0  5  2
    3.0  6  3
   -1.0  1  3
   -1.0  2  4
    2.0  6  4

```

Lower bounds

```

-0.55    -0.55    0.0      0.0      -894.8   -894.8   -1294.8  -0.55
-0.55    -1.0E+25

```

Upper bounds

```

0.55     0.55     1200.0   1200.0  -894.8   -894.8   -1294.8   1.0E+25
1.0E+25  1.0E+25

```

Initial estimate of X

```

0.0  0.0  0.0  0.0

```

8.3 Program Results

e04ugc Example Program Results

Example 1: default options used.

Parameters to e04ugc

Frequencies.

```

fcheck..... 60      expand_freq..... 10000
factor_freq..... 50

```

QP subproblems.

```

scale_tol..... 9.00e-01  minor_feas_tol..... 1.05e-08
scale_opt..... 1      minor_opt_tol..... 1.05e-08
part_price..... 1      crash_tol..... 1.00e-01

```

```

pivot_tol..... 2.04e-11      minor_print_level.... Nag_NoPrint
crash..... Nag_NoCrash      elastic_wt..... 1.00e+02

Derivatives.
obj_deriv..... TRUE          con_deriv..... TRUE
verify_grad..... Nag_SimpleCheck  print_deriv..... Nag_D_Print
Start obj check at col.. 1      Stop obj check at col.. 4
Start con check at col.. 1      Stop con check at col.. 2

The SQP method.
direction..... Nag_Minimize
Nonlinear objective vars 4      major_opt_tol..... 1.05e-08
f_prec..... 1.72e-13          inf_step..... 1.00e+20
max_sb..... 4                f_diff_int..... 4.15e-07
unbounded_obj..... 1.00e+15    c_diff_int..... 5.56e-05
major_step_lim..... 2.00e+00    deriv_linesearch..... FALSE
print_level..... Nag_Soln_Iter  major_iter_lim..... 1000
linesearch_tol..... 9.00e-01    minor_iter_lim..... 500
inf_bound..... 1.00e+20        iter_lim..... 10000

Hessian approximation.
hess_storage..... Nag_HessianFull  hess_update..... 20
hess_freq..... 99999999

Nonlinear constraints.
Nonlinear constraints... 3      major_feas_tol..... 1.05e-08
Nonlinear Jacobian vars. 2      violation_limit..... 1.00e+01

Miscellaneous.
Variables..... 4              Linear constraints.... 3
Nonlinear variables.... 4      Linear variables..... 0
lu_factor_tol..... 5.00e+00    lu_sing_tol..... 2.04e-11
lu_update_tol..... 5.00e+00    lu_den_tol..... 6.00e-01
eps (machine precision). 1.11e-16
start..... Nag_Cold          feas_exit..... FALSE
Names..... not supplied      print_80ch..... TRUE
outfile..... stdout

Memory allocation.
nz_coef..... 5.00e+00          Initial sizes of work arrays.
state..... Nag                Integers..... 1628
lambda..... Nag                Reals..... 1258

XXX Scale option reduced from 1 to 0.
XXX Feasible linear rows.
XXX Norm(x-x0) minimized. Sum of infeasibilities = 0.00e+00.

confun sets 6 out of 6 constraint gradients.
objfun sets 4 out of 4 objective gradients.

-----
Verification of constraint gradients returned by subroutine confun
-----

Cheap test on confun...

The Jacobian seems to be OK.

The largest discrepancy was 4.41e-08 in constraint 2.

```

```
-----
Verification of objective gradients returned by subroutine objfun
-----
```

Cheap test on objfun...

The objective gradients seem to be OK.

Gradient projected in two directions 0.00000000000e+00 0.00000000000e+00

Difference approximations 1.74111992322e-19 4.48742248252e-21

XXX All-slack basis B = I selected.

XXX 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 NP problem after 8 major iterations,
21 minor iterations.

Variable	State	Value	Lower Bound	Upper Bound	Lagr Mult	Residual
X1	BS	1.188764e-01	-5.50000e-01	5.50000e-01	-1.2529e-07	4.3112e-01
X2	BS	-3.962336e-01	-5.50000e-01	5.50000e-01	1.9245e-08	1.5377e-01
X3	BS	6.799453e+02	0.00000e+00	1.20000e+03	1.7001e-10	5.2005e+02
X4	SBS	1.026067e+03	0.00000e+00	1.20000e+03	-2.1918e-10	1.7393e+02

Constrnt	State	Value	Lower Bound	Upper Bound	Lagr Mult	Residual
N1	EQ	-8.948000e+02	-8.94800e+02	-8.94800e+02	-4.3870e+00	3.3644e-09
N2	EQ	-8.948000e+02	-8.94800e+02	-8.94800e+02	-4.1056e+00	6.0038e-10
N3	EQ	-1.294800e+03	-1.29480e+03	-1.29480e+03	-5.4633e+00	3.3549e-09
L1	BS	-5.151099e-01	-5.50000e-0	None	25 0.0000e+00	3.4890e-02
L2	BS	5.151099e-01	-5.50000e-0	None	25 0.0000e+00	1.0651e+00
Free Row	BS	4.091970e+03	None	None	25 -1.0000e+00	4.0920e+03

Exit e04ugc - Optimal solution found.

Final objective value = 5126.498

9 Further Description

nag_opt_nlp_sparse implements a sequential quadratic programming (SQP) method that obtains search directions from a sequence of quadratic programming (QP) subproblems. This section gives a detailed description of the algorithms used by nag_opt_nlp_sparse. This, and possibly the next section, Section 10, may be omitted if the more sophisticated features of the algorithm and software are not currently of interest.

9.1 Overview

Here we briefly summarize the main features of the method and introduce some terminology. Where possible, explicit reference is made to the names of variables that are parameters of the routine or appear in the printed output. Further details can be found in Gill *et al.* (1997).

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 (1970)). The vector $r'_j(x)$ comprises the j th 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 (1974)) 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

$$g(x) = r'(x)^T \lambda = (I \quad J(x)^T \quad G^T) \lambda, \quad (4)$$

where g is the gradient of f evaluated at x ;

- (c) the Lagrange multiplier λ_j associated with the j th 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

$$Z^T g(x) = 0,$$

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

$$N = \{z \mid r'_j(x)z = 0 \text{ for all } j \in \mathcal{G}\}.$$

The columns of Z may then be taken as any basis for the vector space N . The vector $Z^T g$ 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 (1974)).

The basic structure of `nag_opt_nlp_sparse` 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

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

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

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

where $q(x)$ is a quadratic approximation to a modified Lagrangian function (see Gill *et al.* (1997)).

The linear constraint matrix A is stored in the arrays **a**, **ha** and **ka** (see Section 4). 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 $d_j \geq 0$ for all nonbasic variables at their lower bounds, $d_j \leq 0$ 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 the optional parameter **options.minor_opt_tol**).

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

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

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 $0 < \alpha_k \leq 1$) such that the new point

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

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.* (1992)). As in `nag_opt_nlp` (e04ucc), 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 (1991) and Gill *et al.* (1986).

9.2 Treatment of Constraint Infeasibilities

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

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

where $e = (1, 1, \dots, 1)^T$. 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 routine 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 routine 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 routine 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

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

where

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

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_1 QP algorithm of Fletcher (1984). See also Conn (1973).

10 Optional Parameters

A number of optional input and output parameters to nag_opt_nlp_sparse are available through the structure argument **options**, type **Nag_E04_Opt**. A parameter may be selected by assigning an appropriate value to the relevant structure member; those parameters not selected will be assigned default values. If no use is to be made of any of the optional parameters the user should use the NAG defined null pointer, E04_DEFAULT, in place of **options** when calling nag_opt_nlp_sparse; the default settings will then be used for all parameters.

Before assigning values to **options** directly the structure **must** be initialized by a call to the function nag_opt_init (e04xxc). Values may then be assigned to the structure members in the normal C manner.

Option settings may also be read from a text file using the function nag_opt_read (e04xyc) in which case initialization of the **options** structure will be performed automatically if not already done. Any subsequent direct assignment to the **options** structure must **not** be preceded by initialization.

If assignment of memory to pointers in the **options** structure is required, then this must be done directly in the calling program; they cannot be assigned using nag_opt_read (e04xyc).

10.1 Optional Parameter Checklist and Default Values

For easy reference, the following list shows the members of **options** which are valid for nag_opt_nlp_sparse together with their default values where relevant. The number ϵ is a generic notation for *machine precision* (see nag_machine_precision (X02AJC)).

Nag_Start start

Nag_Cold

Boolean list	TRUE
Boolean print_80ch	TRUE
Nag_PrintType print_level	Nag_Soln_Iter
Nag_PrintType minor_print_level	Nag_NoPrint
Nag_DPrintType print_deriv	Nag_D_Print
char outfile[80]	stdout
char **crnames	NULL
Boolean obj_deriv	TRUE
Boolean con_deriv	TRUE
Nag_GradChk verify_grad	Nag_SimpleCheck
Integer obj_check_start	1
Integer obj_check_stop	nonln
Integer con_check_start	1
Integer con_check_stop	njln
double f_diff_int	$\sqrt{f_prec}$
double c_diff_int	f_prec ^{0.33}
Nag_CrashType crash	Nag_NoCrash or Nag_CrashThreeTimes (see below)
Integer expand_freq	10000
Integer factor_freq	50 or 100 (see below)
Integer fcheck	60
Integer hess_freq	99999999
Integer hess_update	20
Integer iter_lim	10000
Integer major_iter_lim	1000
Integer minor_iter_lim	500
Integer part_price	1 or 10 (see below)
Integer scale_opt	1 or 2 (see below)
Integer max_sb	$\min(500, n, \bar{n}+1)$
double crash_tol	0.1
double elastic_wt	1.0 or 100.0 (see below)
double f_prec	$\epsilon^{0.8}$
double inf_bound	10^{20}
double linesearch_tol	0.9
double lu_den_tol	0.6
double lu_sing_tol	$\epsilon^{0.67}$
double lu_factor_tol	5.0 or 100.0 (see below)
double lu_update_tol	5.0 or 10.0 (see below)
double major_feas_tol	$\sqrt{\epsilon}$
double major_opt_tol	$\sqrt{\epsilon}$
double major_step_lim	2.0
double minor_feas_tol	$\sqrt{\epsilon}$
double minor_opt_tol	$\sqrt{\epsilon}$
double nz_coef	5.0
double pivot_tol	$\epsilon^{0.67}$
double scale_tol	0.9
double unbounded_obj	10^{15}
double inf_step	$\max(\text{inf_bound}, 10^{20})$
double violation_limit	10.0
Boolean deriv_linesearch	TRUE
Boolean feas_exit	FALSE
Nag_HessianType hess_storage	Nag_HessianFull or Nag_HessianLimited (see below)
Nag_DirectionType direction	Nag_Minimize
Integer *state	size n+m
double *lambda	size n+m
Integer iter	
Integer major_iter	
Integer nsb	
Integer nf	

10.2 Description of Optional Parameters

start – Nag_Start Input Default = Nag_Cold

On entry: indicates how a starting basis is to be obtained as follows. If **start** = **Nag_Cold**, then an initial Crash procedure will be used to choose an initial basis. If **start** = **Nag_Warm**, then the user must provide a valid definition of the optional parameters **options.state** and **options.nsb**. (Those may be the output of a previous call.)

A warm start will be advantageous if a good estimate of the initial working set is available – for example, when `nag_opt_nlp_sparse` is called repeatedly to solve related problems.

Constraint: **options.start** = **Nag_Cold** or **Nag_Warm**.

list – Boolean Input Default = TRUE

On entry: if **options.list** = **TRUE** the parameter settings in the call to `nag_opt_nlp_sparse` will be printed.

print_80ch – Boolean Input Default = TRUE

On entry: controls the maximum length of each line of output produced by major and minor iterations and by the printing of the solution.

If **print_80ch** = **TRUE** (the default), then a maximum of 80 characters per line is printed;

if **print_80ch** = **FALSE**, then a maximum of 120 characters per line is printed.

(See also **options.print_level** and **options.minor_print_level** below.)

print_level – Integer Input Default = Nag_Soln_Iter

On entry: the level of results printout produced by `nag_opt_nlp_sparse` at each major iteration, as indicated below. A detailed description of the printed output is given in Section 4.1 and Section 10.3. (See also **options.minor_print_level**, below.)

Nag_NoPrint No output.

Nag_Soln The final solution only.

Nag_Iter One line of output for each major iteration (no printout of the final solution).

Nag_Soln_Iter The final solution and one line of output for each iteration.

Nag_Soln_Iter_Full The final solution, one line of output for each major iteration, matrix statistics (initial status of rows and columns, number of elements, density, biggest and smallest elements, etc.), details of the scale factors resulting from the scaling procedure (if **options.scale_opt** = 1 or 2; see below), basis factorization statistics and details of the initial basis resulting from the Crash procedure (if **options.start** = **Nag_Cold** and **options.crash** ≠ **Nag_NoCrash**).

Note that the output for each line of major iteration and for the solution printout contains a maximum of 80 characters if **options.print_80ch** = **TRUE**, and a maximum of 120 characters otherwise. However, if **print_level** = **Nag_Soln_Iter_Full**, some printout may exceed 80 characters even when **options.print_80ch** = **TRUE**.

Constraint: **options.print_level** = **Nag_NoPrint**, **Nag_Soln**, **Nag_Iter**, **Nag_Soln_Iter**, or **Nag_Soln_Iter_Full**.

minor_print_level – Nag_PrintType Input Default = Nag_NoPrint

On entry: controls the amount of printout produced by the minor iterations of `nag_opt_nlp_sparse` (i.e., the iterations of the quadratic programming algorithm), as indicated below. A detailed description of the printed output is given in Section 6.1 (default output at each iteration) and in Section 10.3. (See also **options.print_level** above.)

If **minor_print_level** = **Nag_NoPrint**, no output is produced;

if **minor_print_level** = **Nag_Iter**, the following output is produced for each minor iteration:

if **print_80ch** = **TRUE**, one line of summary output (≤ 80 characters);

if **print_80ch** = **FALSE**, one long line of output (≤ 120 characters).

Constraint: **options.minor_print_level** = **Nag_NoPrint** or **Nag_Iter**.

print_deriv – Nag_DPrintType Input Default = **Nag_D_Print**

On entry: controls whether the results of any derivative checking are printed out (see also the optional parameter **options.verify_grad**).

If a component derivative check has been carried out, then full details will be printed if **print_deriv** = **Nag_D_Print**. If only a simple derivative check is requested, **Nag_D_Print** will produce a statement indicating failure or success. To prevent any printout from a derivative check, set **print_deriv** = **Nag_D_NoPrint**.

Constraint: **options.print_deriv** = **Nag_D_NoPrint** or **Nag_D_Print**.

outfile – char[80] Input Default = `stdout`

On entry: the name of the file to which results should be printed. If **outfile**[0] = `'\0'` then the `stdout` stream is used.

crnames – char ** Input Default = `NULL`

On entry: if **crnames** is not `NULL` then it must point to an array of **n+m** character strings with maximum string length 8, containing the names of the columns and rows (i.e., variables and constraints) of the problem. Thus, **crnames**[*j* – 1] contains the name of the *j*th column (variable), for *j* = 1, 2, ..., **n**, and **crnames**[**n+i** – 1] contains the names of the *i*th row (constraint), for *i* = 1, 2, ..., **m**. If supplied, the names are used in the solution output (see Section 6.1 and Section 10.3).

Constraint: **options.crnames** = `NULL` or $\text{strlen}(\text{options.crnames}[i - 1]) \leq 8$, for *i* = 1, 2, ..., **n+m**.

obj_deriv – Boolean Input Default = **TRUE**

On entry: this argument indicates whether all elements of the objective gradient are provided by the user in function **objfun**. If none or only some of the elements are being supplied by **objfun** then **obj_deriv** should be set to **FALSE**.

Whenever possible all elements should be supplied, since `nag_opt_nlp_sparse` is more reliable and will usually be more efficient when all derivatives are exact.

If **obj_deriv** = **FALSE**, `nag_opt_nlp_sparse` will approximate unspecified elements of the objective gradient using finite differences. The computation of finite-difference approximations usually increases the total run-time, since a call to **objfun** is required for each unspecified element. Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* (1997), for a discussion of limiting accuracy).

At times, central differences are used rather than forward differences, in which case twice as many calls to **objfun** are needed. (The switch to central differences is not under the user's control.)

con_deriv – Boolean Input Default = **TRUE**

On entry: this argument indicates whether all elements of the constraint Jacobian are provided by the user in function **confun** (or possibly directly in **a** for constant elements). If none or only some of the derivatives are being supplied then **con_deriv** should be set to **FALSE**.

Whenever possible all elements should be supplied, since `nag_opt_nlp_sparse` is more reliable and will usually be more efficient when all derivatives are exact.

If **con_deriv** = **FALSE**, `nag_opt_nlp_sparse` will approximate unspecified elements of the constraint Jacobian. One call to **confun** will be needed for each variable for which partial derivatives are not available. For example, if the sparsity of the constraint Jacobian has the form

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

where ‘*’ indicates an element provided by the user and ‘?’ indicates an unspecified element, `nag_opt_nlp_sparse` will call **confun** twice: once to estimate the missing element in column 2, and again once to estimate the two missing elements in column 3. (Since columns 1 and 4 are known, they require no calls to **confun**.)

At times, central differences are used rather than forward differences, in which case twice as many calls to **confun** are needed. (The switch to central differences is not under the user’s control.)

verify_grad – Nag_GradChk *Input* Default = **Nag_SimpleCheck**

On entry: specifies the level of derivative checking to be performed by `nag_opt_nlp_sparse` on the gradient elements computed by the user-supplied functions **objfun** and **confun**. Gradients are verified at the first point that satisfies the linear constraints and the upper and lower bounds. Unspecified gradient elements are not checked, and hence they result in no overhead.

The following values are available:

- | | |
|------------------------|---|
| Nag_NoCheck | No derivative checking is performed. |
| Nag_SimpleCheck | Only a cheap test is performed, requiring three calls to objfun and two calls to confun . Note that no checks are carried out if every column of the constraint gradients (Jacobian) contains a missing element. |
| Nag_CheckObj | Individual objective gradient elements will be checked using a reliable (but more expensive) test. If options.print_deriv = Nag_D_Print (see above), a key of the form OK or BAD? indicates whether or not each element appears to be correct. |
| Nag_CheckCon | Individual columns or the constraint gradient (Jacobian) will be checked using a reliable (but more expensive) test. If options.print_deriv = Nag_D_Print (see above), a key of the form OK or BAD? indicates whether or not each element appears to be correct. |
| Nag_CheckObjCon | Check both constraint and objective gradients (in that order) as described for Nag_CheckCon and Nag_CheckObj (respectively). |

This component check will be made in the range specified by the optional parameters **options.obj_check_start** and **options.obj_check_stop** for the objective gradient, with default values 1 and **nonln**, respectively. For the constraint gradient the range is specified by **options.con_check_start** and **options.con_check_stop**, with default values 1 and **njln**.

Constraint: **options.verify_grad = Nag_NoCheck, Nag_SimpleCheck, Nag_CheckObj, Nag_CheckCon, or Nag_CheckObjCon.**

obj_check_start – Integer	<i>Input</i>	Default = 1
obj_check_stop – Integer	<i>Input</i>	Default = nonln

These options take effect only when **options.verify_grad** is equal to **Nag_CheckObj** or **Nag_CheckObjCon**.

On entry: these parameters may be used to control the verification of gradient elements computed by the function **objfun**. For example, if the first 30 elements of the objective gradient appear to be correct in an earlier run, so that only element 31 remains questionable, it is reasonable to specify **obj_check_start = 31**.

Constraint: $1 \leq \text{options.obj_check_start} \leq \text{options.obj_check_stop} \leq \text{nonln}$.

con_check_start – Integer *Input* Default = 1
con_check_stop – Integer *Input* Default = **njnl**

These options take effect only when **options.verify_grad** is equal to **Nag_CheckCon** or **Nag_CheckObjCon**.

On entry: these parameters may be used to control the verification of the Jacobian elements computed by the function **confun**. For example, if the first 30 columns of the constraint Jacobian appeared to be correct in an earlier run, so that only column 31 remains questionable, it is reasonable to specify **con_check_start** = 31.

Constraint: $1 \leq \text{options.con_check_start} \leq \text{options.con_check_stop} \leq \text{njnl}$.

f_diff_int – double *Input* Default $\sqrt{\text{f_prec}}$

This option does not apply when both **obj_deriv** and **con_deriv** are true.

On entry: **f_diff_int** defines an interval used to estimate derivatives by finite differences in the following circumstances:

- (a) For verifying the objective and/or constraint gradients (see the description of the optional parameter **options.verify_grad**).
- (b) For estimating unspecified elements of the objective and/or the constraint Jacobian matrix.

Using the notation $r = \text{f_diff_int}$, a derivative with respect to x_j is estimated by perturbing that element of x to the value $x_j + r(1 + |x_j|)$, and then evaluating $f(x)$ and/or $F(x)$ (as appropriate) at the perturbed point. If the functions are well scaled, the resulting derivative approximation should be accurate to $O(r)$. Judicious alteration of **f_diff_int** may sometimes lead to greater accuracy. See Gill *et al.* (1981) for a discussion of the accuracy in finite difference approximations.

Constraint: $\epsilon \leq \text{options.f_diff_int} < 1.0$.

c_diff_int – double *Input* Default = **f_prec**^{0.33}

This option does not apply when both **options.obj_deriv** and **options.con_deriv** are true.

On entry: **c_diff_int** is 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 difference (see the optional parameter **options.f_diff_int**). Using the notation $r = \text{c_diff_int}$, the interval used for the j th variable is $h_j = r(1 + |x_j|)$. If the functions are well scaled, the resultant gradient estimates should be accurate to $O(r^2)$. The switch to central differences (not under user-control) is indicated by c at the end of each line of intermediate printout produced by the major iterations (see Section 4.1). The use of finite differences is discussed under the option **options.f_diff_int**.

Constraint: $\epsilon \leq \text{options.c_diff_int} < 1.0$.

crash – Nag_Crash_Type *Input* Default = **Nag_NoCrash** or **Nag_CrashThreeTimes**

This option does not apply when **options.start** = **Nag_Warm**.

On entry: the default value of **crash** is **Nag_NoCrash** if there are any nonlinear constraints, and **Nag_Crash_Three_Times** otherwise.

If **start** = **Nag_Cold**, an internal Crash procedure is used to select an initial basis from the various rows and columns of the constraint matrix ($A - I$). The value of **crash** 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** are the following.

crash	Meaning
Nag_NoCrash	The initial basis contains only slack variables: $B = I$.
Nag_CrashOnce	The Crash procedure is called once (looking for a triangular basis in all rows and columns of A).

Nag_CrashTwice 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).

Nag_CrashThreeTimes 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 **crash** \neq **Nag_NoCrash**, certain slacks on inequality rows are selected for the basis first. (If **crash** = **Nag_CrashTwice** or **Nag_CrashThreeTimes**, 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.

Constraint: **options.crash** = **Nag_NoCrash**, **Nag_CrashOnce**, **Nag_CrashTwice**, or **Nag_CrashThreeTimes**.

expand_freq – Integer Input Default = 10000

On entry: this option is part of the EXPAND anti-cycling procedure due to Gill *et al.* (1989), 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 **options.minor_feas_tol** (see below) is δ . Over a period of **expand_freq** iterations, the feasibility tolerance actually used by **nag_opt_nlp_sparse** (i.e., the *working* feasibility tolerance) increases from 0.5 to δ (in steps of $0.5\delta/\mathbf{expand_freq}$).

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 **options.pivot_tol** below).

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

Constraint: **options.expand_freq** \geq 0.

factor_freq – Integer Input Default = 50 or 100

On entry: **factor_freq** specifies the maximum number of basis changes that will occur between factorizations of the basis matrix. The default value of **factor_freq** is 50 if there are any nonlinear constraints, and 100 otherwise.

For linear problems, the basis factors are usually updated at every iteration. The default value **factor_freq** = 100 is reasonable for typical problems, particularly those that are extremely sparse and 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 the optional parameter **options.fcheck** (see below) to ensure that the general constraints are satisfied. If necessary, the basis will be refactorized before the limit of **factor_freq** updates is reached.

Constraint: **options.factor_freq** \geq 0.

fcheck – Integer *Input* Default = 60

On entry: every **fcheck**-th iteration after the most recent basis iteration, a numerical test is made to see if the current solution (x, s) satisfies the general linear constraints (including any linearized nonlinear constraints). 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 **fcheck** = 0, the value **fcheck** = 99999999 is used and effectively no checks are made.

Constraint: **options.fcheck** ≥ 0 .

hess_freq – Integer *Input* Default = 99999999

This option only takes effect when **options.hess_storage** = **Nag_HessianFull**.

On entry: this option forces the approximate Hessian formed from **hess_freq** BFGS updates to be reset to the identity matrix upon completion of a major iteration.

Constraint: **options.hess_freq** > 0 .

hess_update – Integer *Input* Default = 20

This option only takes effect when **options.hess_storage** = **Nag_HessianLimited**.

On entry: if **options.hess_storage** = **Nag_HessianLimited** (see below), this option 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_update** 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 of **hess_update** is likely to give a robust algorithm without significant expense, but faster convergence may often be obtained with far fewer updates (e.g., **hess_update** = 5).

Constraint: **options.hess_update** ≥ 0 .

iter_lim – Integer *Input* Default = 10000

On entry: 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 also **options.major_iter_lim** and **options.minor_iter_lim** below.)

Constraint: **options.iter_lim** > 0 .

major_iter_lim – Integer *Input* Default = 1000

On entry: 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. Setting **options.major_iter_lim** = 0 and **options.print_deriv** = **Nag_D_Print** (see above) means that the objective and constraint gradients will be checked if **options.verify_grad** \neq **Nag_NoCheck** (see above), but no iterations will be performed.

Constraint: **options.major_iter_lim** ≥ 0 .

minor_iter_lim – Integer *Input* Default = 500

On entry: specifies the maximum number of iterations allowed between successive linearizations of the nonlinear constraints. A value in the range $10 \leq i \leq 50$ prevents excessive effort being expended on early major iterations, but allows 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 **options.crash** above.)

In general, it is unsafe to specify values as small as 1 or 2 (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).

Constraint: **options.minor_iter_lim** ≥ 0 .

part_price – Integer *Input* Default = 1 or 10

On entry: this option is recommended for large problems that have significantly more variables than constraints (i.e., $n \gg m$). The default value of **part_price** is 1 if there are any nonlinear constraints, and 10 otherwise. 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.

part_price	Meaning
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 $j = 1, 2, \dots, p$ (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.

Constraint: **options.part_price** > 0 .

scale_opt – Integer *Input* Default = 1 or 2

On entry: the default value of **scale_opt** is 1 if there are any nonlinear constraints, and 2 otherwise. This option enables you to scale the variables and constraints using an iterative procedure due to Fourer (1982), 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.

scale_opt	Meaning
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 routine 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.

Constraint: **options.scale_opt** = 0, 1 or 2.

max_sb – Integer *Input* Default = $\min(500, n, \bar{n} + 1)$

This option does not apply to linear problems.

On entry: **max_sb** places a limit on the storage allocated for superbasic variables. Ideally, the value of **max_sb** should be set slightly larger than the ‘number of degrees of freedom’ expected at the solution.

For nonlinear problems, the number of degrees of freedom is often called the ‘number of independent variables’. Normally, the value of **max_sb** need not be greater than $\bar{n} + 1$ (where $\bar{n} = \max(\text{nonln}, \text{njnl})$), but for many problems it may be considerably smaller. (This will save storage if \bar{n} is very large.)

Constraint: **options.max_sb** > 0 .

crash_tol – double *Input* Default = 0.1

On entry: this option 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 j th column, other non-zeros a_{ij} in the column are ignored if $|a_{ij}| \leq a_{max} \times \text{crash_tol}$.

When **crash_tol** > 0, the basis obtained by the Crash procedure may not be strictly triangular, 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.

Constraint: $0.0 \leq \text{options.crash_tol} < 1.0$.

elastic_wt – double *Input* Default = 1.0 or 100.0

On entry: this option defines the initial weight γ associated with problem (8). The default value of **elastic_wt** is 100.0 if there are any nonlinear constraints, and 1.0 otherwise.

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 $\gamma = \text{elastic_wt} \times (1 + \|g(x_k)\|_2)$.

Constraint: **options.elastic_wt** ≥ 0.0 .

f_prec – double *Input* Default = $\epsilon^{0.8}$

On entry: this option 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.* (1981) 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 **f_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.

Constraint: $\epsilon \leq \text{options.f_prec} < 1.0$.

inf_bound – double *Input* Default = 10^{20}

On entry: **inf_bound** defines the ‘infinite’ bound 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 $-\text{inf_bound}$ will be regarded as $-\infty$).

Constraint: **options.inf_bound** > 0.0.

linesearch_tol – double *Input* Default = 0.9

On entry: this option 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 **linesearch_tol** = 0.9 requests an inaccurate search, and is appropriate for most problems, particularly those with any nonlinear constraints.

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

If the nonlinear functions are expensive to evaluate, a less accurate search may be appropriate. If all derivatives are available (**options.con_deriv** and **options.obj_deriv** are both true), try **linesearch_tol** = 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 some derivatives are not available (at least one of **options.con_deriv** or **options.obj_deriv** is false), a moderately accurate search may be appropriate; try **linesearch_tol** = 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.

Constraint: $0.0 \leq \text{options.linesearch_tol} < 1.0$.

lu_den_tol – double *Input* Default = 0.6

On entry: this option defines the density tolerance 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.

Constraint: **options.lu_den_tol** ≥ 0.0 .

lu_sing_tol – double *Input* Default = $\epsilon^{0.67}$

On entry: this option defines the singularity tolerance used to guard against ill-conditioned basis matrices. Whenever the basis is refactorized, the diagonal elements of *U* are tested as follows. If $|u_{jj}| \leq \text{lu_sing_tol}$ or $|u_{jj}| < \text{lu_sing_tol} \times \max_i |u_{ij}|$, the *j*th column of the basis is replaced by the corresponding slack variable. This is most likely to occur when **options.start** = **Nag_Warm**, 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** = 0.00001 (say) may therefore help cause a judicious change of basis in such situations.

Constraint: **options.lu_sing_tol** > 0.0 .

lu_factor_tol – double *Input* Default = 5.0 or 100.0
lu_update_tol – double *Input* Default = 5.0 or 10.0

On entry: **lu_factor_tol** and **lu_update_tol** affect the stability and sparsity of the basis factorization $B = LU$, during refactorization and updates respectively. The default values are **lu_factor_tol** = **lu_update_tol** = 5.0 if there are any nonlinear constraints, and **lu_factor_tol** = 100.0 and **lu_update_tol** = 10.0 otherwise.

The lower triangular matrix *L* can be seen as a product of matrices of the form

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

where the multipliers μ satisfy $|\mu| < \text{lu_factor_tol}$ during refactorization or $|\mu| < \text{lu_update_tol}$ during update. The default values of **lu_factor_tol** and **lu_update_tol** usually strike a good compromise between stability and sparsity. Smaller values of **lu_factor_tol** and **lu_update_tol** favour stability, while larger values favour sparsity. For large and relatively dense problems, setting **lu_factor_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_factor_tol** and/or **lu_update_tol** in order to achieve stability.

Constraints:

- options.lu_factor_tol** ≥ 1.0 ,
- options.lu_update_tol** ≥ 1.0 .

major_feas_tol – double Input Default = $\sqrt{\epsilon}$

On entry: this option 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

$$\text{rowerr} = \max_i \frac{\text{viol}_i}{\| (x, s) \|} \leq \mathbf{major_feas_tol},$$

where viol_i is the violation of the i th nonlinear constraint.

Constraint: **options.major_feas_tol** > ϵ .

major_opt_tol – double Input Default = $\sqrt{\epsilon}$

On entry: this option specifies the final accuracy of the dual variables. If `nag_opt_nlp_sparse` terminates with `fail.code = NE_NOERROR`, a primal and dual solution (x, s, π) will have been computed such that

$$\text{maxgap} = \max_j \frac{\text{gap}_j}{\| \pi \|} \leq \mathbf{major_opt_tol},$$

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

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

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 j th element of the objective gradient and a_j is the associated column of the constraint matrix $(A - I)$:

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

Constraint: **options.major_opt_tol** > 0.0.

major_step_lim – double Input Default = 2.0

On entry: this option 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 optional parameter **major_step_lim** is therefore used to define a step limit $\bar{\beta}$ given by

$$\bar{\beta} = \frac{\mathbf{major_step_lim}(1 + \|x\|_2)}{\|p\|_2},$$

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. **major_step_limit** provides an additional safeguard. The default value **major_step_lim** = 2.0 should not affect progress on well-behaved functions, but values such as **major_step_lim** = 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.

Constraint: **options.major_step_lim** > 0.0.

minor_feas_tol – double *Input* Default = $\sqrt{\epsilon}$

On entry: this option attempts to ensure that all variables eventually satisfy their upper and lower bounds to within the tolerance **minor_feas_tol**. 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 **options.major_feas_tol** (see above) 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. 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 **options.scale_opt** ≥ 1 (see above), feasibility is defined in terms of the *scaled* problem (since it is more likely to be meaningful).

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 value **minor_feas_tol** = 10^{-6} is used, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious; in general, you should attempt to keep x as far away from singularities as possible.)

In reality, **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 routine is then in *elastic mode* thereafter (with only the linearized nonlinear constraints defined to be elastic). (See also **options.elastic_wt** above.)

Constraint: **options.minor_feas_tol** $> \epsilon$.

minor_opt_tol – double *Input* Default = $\sqrt{\epsilon}$

On entry: this option 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 j th 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

$$\frac{d_j}{\|\pi\|} \geq -\mathbf{minor_opt_tol} \quad \text{or} \quad \frac{d_j}{\|\pi\|} \leq \mathbf{minor_opt_tol}$$

respectively, and if $\frac{|d_j|}{\|\pi\|} \leq \mathbf{minor_opt_tol}$ 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 the optional parameter **options.major_opt_tol** above.)

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

Constraint: **options.minor_opt_tol** > 0.0 .

nz_coef – double *Input* Default = 5.0

This option is ignored if **options.hess_storage** = **Nag_HessianFull**.

On entry: **nz_coef** defines how much memory is initially allocated for the basis factors: by default, `nag_opt_nlp_sparse` allocates approximately **nnz** \times **options.nz_coef** reals and $2 \times \mathbf{nnz} \times \mathbf{options.nz_coef}$ integers in order to compute and store the basis factors. If at some point this appears not to be enough, an internal warm restart with more memory is automatically attempted, so that `nag_opt_nlp_sparse` should complete anyway. Thus this option generally does not need to be modified.

However, if a lot of memory is available, it is possible to increase the value of **nz_coef** such as to limit the number of compressions of the work space and possibly avoid internal restarts. On the other hand, for large problems where memory might be critical, decreasing the value of **nz_coef** can sometimes save some memory.

Constraint: **options.nz_coef** ≥ 1.0 .

pivot_tol – double *Input* Default = $\epsilon^{0.67}$

On entry: this option is used during the solution of QP subproblems 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 are ignored (and therefore cannot be pivot elements) if they are smaller than **pivot_tol**.

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

Constraint: **options.pivot_tol** > 0.0 .

scale_tol – double *Input* Default = 0.9

On entry: this option 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 p th scaling pass through A . The scaling procedure is terminated if $a_p \geq a_{p-1} \times \text{scale_tol}$ for some $p \geq 3$. Thus, increasing the value of **scale_tol** from 0.9 to 0.99 (say) will probably increase the number of passes through A .

Constraint: $0.0 < \text{options.scale_tol} < 1.0$.

unbounded_obj – double *Input* Default = 10^{15}
inf_step – double *Input* Default = $\max(\text{inf_bound}, 10^{20})$

On entry: these options are intended 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 **inf_step**, the iterations are terminated and the routine returns with **fail.code** = **NE_MAYBE_UNBOUNDED**.

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.

Constraints:

options.unbounded_obj > 0.0 ,

options.inf_step > 0.0 .

violation_limit – double *Input* Default = 10.0

On entry: this option defines 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

$$v_i(x_{k+1}) \leq \text{violation_limit} \times \max(1, v_i(x_0)),$$

where x_0 is the point at which the nonlinear constraints are first evaluated and $v_i(x)$ is the i th 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_limit**. 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_limit** may be any large positive value.

Constraint: **options.violation_limit** > 0.0 .

deriv_linesearch – Boolean *Input* Default = **TRUE**

On entry: at each major iteration, a linesearch is used to improve the value of the Lagrangian merit function (6). The default linesearch uses safeguarded cubic interpolation and requires both function and gradient values in order to compute estimates of the step α_k . If some analytic derivatives are not provided or **deriv_linesearch** = **FALSE** is specified, a linesearch based upon safeguarded quadratic interpolation (which does not require the evaluation or approximation of any gradients) is used instead.

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.

If **deriv_linesearch** = **FALSE** is selected, nag_opt_nlp_sparse signals the evaluation of the linesearch by calling **objfun** and **confun** with **comm**→**flag** = 0. Once the linesearch is complete, the nonlinear functions are re-evaluated with **comm**→**flag** = 2. If the potential savings offered by a nonderivative linesearch are to be fully realized, it is essential that **objfun** and **confun** be coded so that no derivatives are computed when **comm**→**flag** = 0.

Constraint: **options.deriv_linesearch** = **TRUE** or **FALSE**.

feas_exit – Boolean *Input* Default = **FALSE**

This option is ignored if the value of **options.major_iter_lim** (see above) is exceeded, or the linear constraints are infeasible.

On entry: if termination is about to occur at a point that does not satisfy the nonlinear constraints and **feas_exit** = **TRUE** is selected, this option requests that additional iterations be 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.

Otherwise, this option requests no additional iterations be performed.

Constraint: **options.feas_exit** = **TRUE** or **FALSE**.

hess_storage – Nag_Hessian_Type *Input* Default = **Nag_HessianFull** or **Nag_HessianLimited**

On entry: this option specifies the method for storing and updating the quasi-Newton approximation to the Hessian of the Lagrangian function. The default is **Nag_HessianFull** if the number of nonlinear variables \bar{n} ($= \max(\text{nonln}, \text{njln})$) < 75 , and **Nag_HessianLimited** otherwise.

If **hess_storage** = **Nag_HessianFull**, 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, $\bar{n} < 75$). In this case, the storage requirement is fixed and you can expect one-step Q-superlinear convergence to the solution.

hess_storage = **Nag_HessianLimited** should only be specified when \bar{n} is very large. In this case 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.)

Note that if **options.hess_freq** = 20 (see above) is used in conjunction with **hess_storage** = **Nag_HessianFull**, the effect will be similar to using **hess_storage** = **Nag_HessianLimited** in conjunction with **options.hess_update** = 20 (see above), except that the latter will retain the current diagonal during resets.

Constraint: **options.hess_storage** = **Nag_HessianLimited** or **Nag_HessianFull**.

direction – Nag_Direction *Input* Default = **Nag_Minimize**

On entry: if **direction** = **Nag_FeasiblePoint**, nag_opt_nlp_sparse attempts to find a feasible point (if any) for the nonlinear constraints by omitting the objective function. It can also be used to check whether the nonlinear constraints are feasible.

Otherwise, **direction** specifies the direction of optimization. It applies to both linear and nonlinear terms (if any) in the objective function. 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.

Constraint: **options.direction** = **Nag_FeasiblePoint**, **Nag_Minimize**, or **Nag_Maximize**.

state – Integer * *Input/Output* Default memory = **n+m**

On entry: **state** need not be set if the default option of **options.start** = **Nag_Cold** is used as **n+m** values of memory will be automatically allocated by `nag_opt_nlp_sparse`.

If the optional parameter **options.start** = **Nag_Warm** has been chosen, **state** must point to a minimum of **n+m** elements of memory. This memory will already be available if the **options** structure has been used in a previous call to `nag_opt_nlp_sparse` from the calling program, with **options.start** = **Nag_Cold** and the same values of **n** and **m**. If a previous call has not been made sufficient memory must be allocated by the user.

If the user does supply a **state** vector and **options.start** = **Nag_Cold**, then the first **n** elements of **state** must specify the initial states of the problem variables. (The slacks s need not be initialized.) 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 **state**[$j - 1$] ($j = 1, 2, \dots, n$) are:

state [$j - 1$]	State of $\mathbf{xs}[j - 1]$ 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, the user may set **state**[$j - 1$] = 0 (and **xs**[$j - 1$] = 0.0), for $j = 1, 2, \dots, n$. All variables will then be eligible for the initial basis. Less trivially, to say that the j th variable will probably be equal to one of its bounds, the user should set **state**[$j - 1$] = 4 and **xs**[$j - 1$] = **bl**[$j - 1$] or **state**[$j - 1$] = 5 and **xs**[$j - 1$] = **bu**[$j - 1$] as appropriate.

Following the Crash procedure, variables for which **state**[$j - 1$] = 2 are made superbasic. Other variables not selected for the basis are then made nonbasic at the value **xs**[$j - 1$] if **bl**[$j - 1$] \leq **xs**[$j - 1$] \leq **bu**[$j - 1$], or at the value **bl**[$j - 1$] or **bu**[$j - 1$] closest to **xs**[$j - 1$].

When **options.start** = **Nag_Warm**, **state** and **xs** must specify the initial states and values, respectively, of the variables and slacks (x, s). If `nag_opt_nlp_sparse` has been called previously with the same values of **n** and **m**, **state** already contains satisfactory information.

Constraints:

If **options.start** = **Nag_Cold**, $0 \leq \mathbf{options.state}[j - 1] \leq 5$, for $j = 1, 2, \dots, n$.

If **options.start** = **Nag_Warm**, $0 \leq \mathbf{options.state}[j - 1] \leq 3$, for $j = 1, 2, \dots, n+m$.

On exit: the final states of the variables and slacks (x, s). The significance of each possible value of **state** is as follows:

state [$j - 1$]	State of variable j	Normal value of $\mathbf{xs}[j - 1]$
0	Nonbasic	bl [$j - 1$]
1	Nonbasic	bu [$j - 1$]
2	Superbasic	Between bl [$j - 1$] and bu [$j - 1$]
3	Basic	Between bl [$j - 1$] and bu [$j - 1$]

If the problem is feasible (i.e., **ninf** = 0), basic and superbasic variables may be outside their bounds by as much as the optional parameter **options.minor_feas_tol**. Note that unless the optional parameter **options.scale_opt** = 0, **options.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 **options.minor_feas_tol**, and there may be some nonbasic variables for which $\mathbf{xs}[j-1]$ lies strictly between its bounds.

If the problem is infeasible (i.e., **ninf** > 0), some basic and superbasic variables may be outside their bounds by an arbitrary amount (bounded by **sinf** if scaling was not used (**options.scale_opt** = 0)).

lambda – double * *Input/Output* Default memory = **n+m**

On entry: if **options.start** = **Nag_Cold**, the user does not need to provide memory for **lambda**, as **n+m** values of memory will be automatically allocated by **nag_opt_nlp_sparse**. This is the recommended method of use of **lambda**. However the user may supply memory from the calling program.

If the option **options.start** = **Nag_Warm** has been chosen, **lambda** must point to a minimum of **n+m** elements of memory. This memory will already be available if the **options** structure has been used in a previous call to **nag_opt_nlp_sparse** from the calling program, with **options.start** = **Nag_Cold** and the same values of **n** and **m**. If a previous call has not been made, sufficient memory must be allocated by the user.

When a ‘warm start’ is chosen **lambda**[$j-1$] must contain a multiplier estimate for each nonlinear constraint for $j = \mathbf{n}+1, \mathbf{n}+2, \dots, \mathbf{n}+\mathbf{ncnlin}$. The remaining elements need not be set. If nothing is known about the problem, or there is no wish to provide special information, the user may set **lambda**[$j-1$] = 0 for $j = \mathbf{n}+1, \mathbf{n}+2, \dots, \mathbf{n}+\mathbf{ncnlin}$.

On exit: a set of Lagrange multipliers for the bound constraints on the variables (*reduced costs*) and the general constraints (*shadow costs*). More precisely, the first **n** elements contain the multipliers for the bound constraints on the variables, the next **ncnln** elements contain the multipliers for the nonlinear constraints $F(x)$ (if any) and the next **m-ncnln** elements contain the multipliers for the linear constraints Gx and the free row (if any).

iter – Integer *Output*

On exit: the total number of minor iterations (summed over all major iterations).

major_iter – Integer *Output*

On exit: the number of major iterations that have been performed in **nag_opt_nlp_sparse**.

nsb – Integer *Input/Output*

On entry: the number of superbasic variables. It need not be specified if **options.start** = **Nag_Cold** but must retain its value from a previous call when **options.start** = **Nag_Warm**.

Constraint: **options.nsb** ≥ 0 when **options.start** = **nag_warm**.

On exit: the final number of superbasic variables.

nf – Integer *Output*

On exit: the number of calls to **objfun**.

ncon – Integer *Output*

On exit: the number of calls to **confun**.

10.3 Description of Printed Output

This section describes the intermediate printout and final printout produced by **nag_opt_nlp_sparse**. The level of printed output can be controlled by the user with the structure members **options.list**, **options.print_deriv**, **options.print_level**, **options.minor_print_level**, **options.print_80ch**, and **options.outfile** (see Section 10.2). If **options.list** = **TRUE** then the parameter values to **nag_opt_nlp_sparse** are listed, followed by the result of any derivative check when **options.print_deriv** = **Nag_D_Print**. The printout of results is then governed by the values of **options.print_80ch**, **options.print_level** and **options.minor_print_level**. The default of **options.print_level** = **Nag_Soln_Iter**, **options.minor_print_level** = **Nag_NoPrint**, and

options.print_80ch = TRUE produces a single line of output at each major iteration and the final result (see Section 4.1). This section describes all of the possible other levels of results printout available from `nag_opt_nlp_sparse`.

If a simple derivative check, **options.verify_grad = Nag_SimpleCheck**, is requested then a statement indicating success or failure is given. The largest error found in the objective and the constraint Jacobian are also output.

When a component derivative check (see the optional parameter **options.verify_grad** in Section 10.2) is selected, the element with the largest relative error is identified for the objective and the constraint Jacobian.

If **options.print_deriv = Nag_D_Print** then the following results are printed for each component:

<code>x[j-1]</code>	the element of x .
<code>dx[j-1]</code>	the finite difference interval.
Jacobian value	the nonlinear Jacobian element.
<code>g[j-1]</code>	the objective gradient element.
Difference approx.	the finite difference approximation.

The indicator, OK or BAD?, states whether the derivative provided and the finite difference approximation are in agreement. If the derivatives are believed to be in error `nag_opt_nlp_sparse` will exit with **fail.code** set to either **NE_CON_DERIV_ERRORS**, or **NE_OBJ_DERIV_ERROR**, depending on whether the error was detected in the constraint Jacobian or in the objective gradient.

When **options.print_level = Nag_Iter, Nag_Soln_Iter, or Nag_Soln_Iter_Full**, and **options.print_80ch = FALSE**, the following line of intermediate printout (≤ 120 characters) is sent at every major iteration to **options.outfile**. Unless stated otherwise, the values of the quantities printed are those in effect *on completion* of the given iteration.

Major	is the major iteration count.
Minor	is the number of minor iterations required by the feasibility and optimality phases of the QP subproblem. Generally, <code>Minor</code> will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 9).
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.
nObj	is the number of times objfun 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. <code>nObj</code> is printed as a guide to the amount of work required for the linesearch.
nCon	is the number of times confun has been called to evaluate the nonlinear constraint functions (not printed if ncnln is zero).
Merit	is the value of the augmented Lagrangian merit function (6) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see Section 9.1). As the solution is approached, <code>Merit</code> will converge to the value of the objective function at the solution.

In elastic mode (see Section 9.2), the merit function is a composite function involving the constraint violations weighted by the value of the optional parameter **options.elastic_wt** (default value = 1.0 or 100.0).

If there are no nonlinear constraints present, 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 <i>rowerr</i> , the largest element of the scaled nonlinear constraint residual vector defined in the description of the optional parameter options.major_feas_tol . The solution is regarded as ‘feasible’ if <i>Feasibl</i> is less than (or equal to) options.major_feas_tol (default value = $\sqrt{\epsilon}$). <i>Feasibl</i> will be approximately zero in the neighbourhood of a solution. If there are no nonlinear constraints present, all iterates are feasible and this entry is not printed.
Optimal	is the value of <i>maxgap</i> , the largest element of the maximum complementarity gap vector defined in the description of the optional parameter options.major_opt_tol . The Lagrange multipliers are regarded as ‘optimal’ if <i>Optimal</i> is less than (or equal to) options.major_opt_tol (default value = $\sqrt{\epsilon}$). <i>Optimal</i> will be approximately zero in the neighbourhood of a solution.
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 ncnl is zero).
LU	is the number of non-zeros representing the basis factors <i>L</i> and <i>U</i> 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, $LU = \text{lenL} + \text{lenU}$, where <i>lenL</i> is the number of subdiagonal elements in the columns of a lower triangular matrix and <i>lenU</i> is the number of diagonal and superdiagonal elements in the rows of an upper triangular matrix. As columns of <i>B</i> are replaced during the minor iterations, the value of <i>LU</i> 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, <i>LU</i> will reflect the number of non-zeros in the <i>LU</i> factors at the start of each QP subproblem. If there are no nonlinear constraints present, refactorization is subject only to the value of the optional parameter options.factor_freq (default value = 50 or 100) and hence <i>LU</i> will tend to increase between factorizations.
Swp	is the number of columns of the basis matrix <i>B</i> that were swapped with columns of <i>S</i> in order to improve the condition number of <i>B</i> (not printed if ncnl is zero). The swaps are determined by an <i>LU</i> factorization of the rectangular matrix $B_S = (B \ S)^T$, with stability being favoured more than sparsity.
Cond Hz	is an estimate of the condition number of the reduced Hessian of the Lagrangian (not printed if ncnl and nonln are both zero). It is the square of the ratio between the largest and smallest diagonal elements of the upper triangular matrix <i>R</i> . This constitutes a lower bound on the condition number of the matrix $R^T R$ 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 the optional parameters options.major_feas_tol and options.major_opt_tol . Each letter is T if the test is satisfied, and F otherwise. The tests indicate whether the values of <i>Feasibl</i> and <i>Optimal</i> are sufficiently small. For example, TF or TT is printed if there are no nonlinear constraints present (since all iterates are feasible).
M	is printed if an extra evaluation of objfun and confun 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.
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 the optional parameters options.hess_freq (default value = 99999999) and options.hess_update (default value = 20). However, it may also be necessary to reset an ill-conditioned Hessian from time to time.
l	is printed if the change in the variables was limited by the value of the optional parameter options.major_step_lim (default value = 2.0). If this output occurs frequently during later iterations, it may be worthwhile increasing the value of options.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 because the number of iterations specified by the value of the optional parameter options.minor_iter_lim (default value = 500) was reached.
i	is printed if the QP subproblem was infeasible when the routine 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 (8) (see Section 9.2). It is also printed if the minimizer of the elastic subproblem does not satisfy the linearized constraints when the routine 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.

When **options.minor_print_level = Nag_Iter** and **options.print_80ch = TRUE**, the following line of intermediate printout (≤ 80 characters) is sent at every minor iteration to **options.outfile**. Unless stated otherwise, the values of the quantities printed are those in effect *on completion* of the given iteration.

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 (9) 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 **options.minor_print_level = Nag_Iter** and **options.print_80ch = FALSE**, the following line of intermediate printout (≤ 120 characters) is sent at every minor iteration to **options.outfile**. Unless stated otherwise, the values of the quantities printed are those in effect *on completion* of the given iteration.

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 is the iteration count.

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 is the value of the step length α taken along the current search direction p . The variables x have just been changed to $x + \alpha p$. If a variable is made superbasic during the current iteration (i.e., **+SBS** is positive), **Step** will be the step to the nearest bound. During the optimality phase, the step can be greater than unity only if the reduced Hessian is not positive-definite.

Pivot is the r th element of a vector y satisfying $By = a_q$ whenever a_q (the q th column of the constraint matrix $(A - I)$) replaces the r th 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 the optional parameter **options.pivot_tol** (default value = $\epsilon^{0.67}$) to exclude very small elements of y from consideration during the computation of **Step**.

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/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 (9), 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 $\text{len}L$. 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 $\text{len}U$. 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. If it does not, `nag_opt_nlp_sparse` will attempt to expand the internal workspace allocated for the basis factors.

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` is the Euclidean norm of the reduced gradient at the start of the current iteration. During the optimality phase, this norm will be approximately zero after a unit step.

`nS` is the current number of superbasic variables.

`Cond Hz` is an estimate of the condition number of the reduced Hessian of the Lagrangian (not printed if `ncnl` and `nonln` are both zero). It is the square of the ratio between the largest and smallest diagonal elements of an upper triangular matrix R . This constitutes a lower bound on the condition number of the matrix $R^T R$ that approximates the reduced Hessian. The larger this number, the more difficult the problem.

When `options.print_level = Nag_Soln_Iter_Full`, the following lines of intermediate printout (≤ 120 characters) are sent to `options.outfile` whenever the matrix B or $B_S = (B \ S)^T$ is factorized. 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 optional parameter `options.lu_factor_tol` (default value = 5.0 or 100.0).

Note that B_S may be factorized at the beginning of just some of the major iterations. It is immediately followed by a factorization of B itself. Note also that factorizations can occur during the solution of a QP problem.

`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 the optional parameter <code>options.factor_freq</code> (default value = 50 or 100).
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 (see the description for the optional parameter <code>options.fcheck</code>).
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).

`Elms` 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, $Density = 100 \times Elms / (Nonlinear + Linear + Slacks)^2$.

`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, $\text{Increase} = 100 \times (\text{lenL} + \text{lenU} - \text{Elems})/\text{Elems}$.
m	is the number of rows in the problem. Note that $m = U_t + L_t + \text{bp}$.
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 the optional parameter options.lu_factor_tol (default value = 5.0 or 100.0).
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 U_{\max} . If the basis is strictly triangular, <i>none</i> of the elements of B will contribute, and U_{\max} will be zero.) Ideally, U_{\max} should not be significantly larger than B_{\max} . If it is several orders of magnitude larger, it may be advisable to reset the options.lu_factor_tol to some value nearer unity. U_{\max} 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 U_{\max}/B_{\max} , which should not be too large. Providing L_{\max} is not large (say < 10.0), the ratio $\max(B_{\max}, U_{\max})/U_{\min}$ 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 U_{\min} 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 **options.print_level** = **Nag_Soln_Iter_Full**, and **options.crash** \neq **Nag_NoCrash** (default value = **Nag_NoCrash** or **Nag_CrashThreeTimes**), the following lines of intermediate printout are sent to **options.outfile** whenever **options.start** = **Nag_Cold**. 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., <code>options.state[j - 1] = 3</code> for some $j \leq n$). It will be a subset of the columns for which <code>options.state[j - 1] = 3</code> was specified.
Unit	is the number of unit columns in the basis.
Double	is the number of columns in the basis containing two non-zeros.
Triangle	is the number of triangular columns in the basis with three (or more) non-zeros.
Pad	is the number of slacks used to pad the basis (to make it a non-singular triangle).

When `options.print_level = Nag_Soln` or `Nag_Soln_Iter`, and `options.print_80ch = FALSE`, the following lines of final printout (≤ 120 characters) are sent to `options.outfile`.

Let x_j denote the j th ‘column variable’, for $j = 1, 2, \dots, n$. We assume that a typical variable x_j has bounds $\alpha \leq x_j \leq \beta$.

The following describes the printout for each column (or variable).

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. Usually $\alpha < x_j < \beta$.

A key is sometimes printed before `State` to give some additional information about the state of x_j . Note that unless the optional parameter `options.scale_opt = 0` (default value = 1 or 2) is specified, 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 the optional parameter `options.minor_feas_tol` (default value = $\sqrt{\epsilon}$).

N *Not precisely optimal.* x_j is nonbasic. Its reduced gradient is larger than the value of the optional parameter `options.major_feas_tol` (default value = $\sqrt{\epsilon}$).

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 Limit	is α , the lower bound specified for x_j . None indicates that <code>bl[j - 1] ≤ -options.inf_bound</code> .

Upper Limit is β , the upper bound specified for x_j . None indicates that $\mathbf{bu}[j-1] \geq \mathbf{options.inf_bound}$.

Reduced Gradient is the value of d_j at the final iterate.

$m + j$ is the value of $m + j$.

General linear constraints take the form $l \leq Ax \leq u$. Let a_i^T denote the i th row of A , for $i = 1, 2, \dots, n$. The i th 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 i th '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 row (or constraint).

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 i th row.

State gives the state of the i th 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 some additional information about the state of s_i . Note that unless the optional parameter **options.scale_opt** = 0 (default value = 1 or 2) is specified, 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 dual variables (or 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 the optional parameter **options.minor_feas_tol** (default value = $\sqrt{\epsilon}$).

N *Not precisely optimal*. s_i is nonbasic. Its reduced gradient is larger than the value of the optional parameter **options.major_feas_tol** (default value = $\sqrt{\epsilon}$).

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 Limit is α , the lower bound specified for the i th row. None indicates that $\mathbf{bl}[n + i - 1] \leq -\mathbf{options.inf_bound}$.

Upper Limit is β , the upper bound specified for the i th row. None indicates that $\mathbf{bu}[n + i - 1] \geq \mathbf{options.inf_bound}$.

Dual Activity is the value of the dual variable π_i .

i gives the index i of the i th row.

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

11 Example 2

This example solves the same problem as Example 1 but illustrates the use of the **options** structure. In addition to the data as read in Example 1, the data for this example also includes a set of user-defined column and row names, and some option settings. The **options** structure is initialized by `nag_opt_init` (e04xxc) and the **crnames** member is assigned to the array of character strings into which the column and row names were read. Two options are read from the data file by use of `nag_opt_read` (e04xyc). Note that, unlike for some other optimization routines, optional parameters to `nag_opt_nlp_sparse` are not checked inside `nag_opt_read` (e04xyc); they are checked inside the main call to `nag_opt_nlp_sparse`.

On return from `nag_opt_nlp_sparse`, the solution is perturbed slightly and some further options set, selecting a warm start and a reduced level of printout. `nag_opt_nlp_sparse` is then called for a second time. Finally, the memory freeing function `nag_opt_free` (e04xzc) is used to free the memory assigned by `nag_opt_nlp_sparse` to the pointers in the options structure. Users should **not** use the standard C function `free()` for this purpose.

11.1 Program Text

```
#define MAXNAMES 300
static int ex2(void)
{
    char names    [MAXNAMES][9];
    char *crnames[MAXNAMES];

    double  *a=0 , *b1=0, *bu=0, *xs=0;
    Integer *ha=0, *ka=0;

    Integer exit_status=0;

    double obj, sinf;
    Integer m, n, nnz, ncnln, njnln, nonln;
    Integer ninf;
    Integer i, icol, j, jcol;
    Integer iobj;
    static NagError fail;
    Nag_E04_Opt options;

    Vprintf("\nExample 2: Use of the option structure.\n");
    Vscanf("%*[\n]");

    fail.print = TRUE;

    /* Read the problem dimensions */
    Vscanf("%*[\n]");
    Vscanf("%ld%ld", &n, &m);

    /* Read NCNLN, NONLN and NJNLN from data file. */
    Vscanf("%*[\n]");
    Vscanf("%ld%ld%ld", &ncnln, &nonln, &njnln);

    /* Read NNZ, IOBJ */
    Vscanf("%*[\n]");
    Vscanf("%ld%ld", &nnz, &iobj);

    if ( ! (a = NAG_ALLOC(nnz, double)) ||
          ! (b1 = NAG_ALLOC(n+m, double)) ||
          ! (bu = NAG_ALLOC(n+m, double)) ||
          ! (xs = NAG_ALLOC(n+m, double)) ||
```

```

    ! (ha = NAG_ALLOC(nnz, Integer)) ||
    ! (ka = NAG_ALLOC(n+1, Integer)) )
{
    Vprintf("Allocation failure\n");
    exit_status = 1;
    goto END;
}

/* Read the column and row names */
Vscanf(" %*[^\\n]");
Vscanf(" %*[']");
for (i = 0; i < n+m; ++i)
{
    Vscanf(" '%8c'", names[i]);
    names[i][8] = '\\0';
    crnames[i] = names[i];
}

/* read the matrix and set up ka. */
jcol = 1;
ka[jcol - 1] = 0;
Vscanf(" %*[^\\n]");
for (i = 0; i < nnz; ++i)
{
    /* a[i] stores (ha[i], icol) element of matrix */
    Vscanf("%lf%ld%ld", &a[i], &ha[i], &icol);
    if (icol < jcol)
    {
        /* Elements not ordered by increasing column index. */
        Vprintf("%s%5ld%s%5ld%s\\n", "Element in column", icol,
            " found after element in column", jcol,
". Problem abandoned.");
        exit_status=1;
        goto END;
    }
    else if (icol == jcol + 1)
    {
        /* Index in a of the start of the icol-th column equals i. */
        ka[icol - 1] = i;
        jcol = icol;
    }
    else if (icol > jcol + 1)
    {
        /* 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 ka to i.
        */
        for (j = jcol + 1; j <= icol - 1; ++j)
            ka[j - 1] = i;

        ka[icol - 1] = i;
        jcol = icol;
    }
}
ka[n] = nnz;
if (n > icol)
{
    /* Columns N,N-1,...,ICOL+1 are empty. Set the
    * corresponding elements of ka accordingly. */

```



```

    for (j = icol; j <= n - 1; ++j)
        ka[j] = nnz;
}

/* Read the bounds */
Vscanf("%*[\n]");
for (i = 0; i < n + m; ++i)
    Vscanf("%lf", &bl[i]);
Vscanf("%*[\n]");
for (i = 0; i < n + m; ++i)
    {
        Vscanf("%lf", &bu[i]);
    }

/* Read the initial estimate of x */
Vscanf("%*[\n]");
for (i = 0; i < n; ++i)
    {
        Vscanf("%lf", &xs[i]);
    }

/* Initialize the options structure */
e04xxc(&options);

/* Read some option values from standard input */
e04xyc("e04ugc", "stdin", &options, (Boolean)TRUE,
      "stdout", NAGERR_DEFAULT);

/* Set some other options directly */
options.major_iter_lim = 100;
options.crnames = crnames;

/* Solve the problem. */
e04ugc (confun, objfun, n, m,
        ncnln, nonln, njnln, iobj, nnz,
        a, ha, ka, bl, bu, xs,
        &ninf, &sinf, &obj, NAGCOMM_NULL,
        &options, &fail);

if (fail.code == NE_NOERROR)
    {
        /* We perturb the solution and solve the
         * same problem again using a warm start.
         */
        Vprintf("\n\n\nA run of the same example with a warm start:\n");
        Vprintf("-----\n");
        options.start = Nag_Warm;

        /* Modify some printing options */
        options.print_deriv = Nag_D_NoPrint;
        options.print_level = Nag_Iter;

        /* Perturb xs */
        for (i=0; i<n+m;i++)
            {
                xs[i]+=0.2;
            }
    }

```

```

/* Reset multiplier estimates to 0.0 */
if (ncnln > 0)
{
    for (i=0; i<ncnln; i++)
        options.lambda[n+i]=0.0;
}
/* Solve the problem again. */
e04ugc (confun, objfun, n, m,
        ncnln, nonln, njnln, iobj, nnz,
        a, ha, ka, bl, bu, xs,
        &ninf, &sinf, &obj, NAGCOMM_NULL,
        &options, &fail);
}
exit_status = fail.code;

/* Free memory allocated by e04ugc to pointers in options */
e04xzc(&options, "all", NAGERR_DEFAULT);

END:
if ( a) NAG_FREE(a);
if (bl) NAG_FREE(bl);
if (bu) NAG_FREE(bu);
if (xs) NAG_FREE(xs);
if (ha) NAG_FREE(ha);
if (ka) NAG_FREE(ka);
return exit_status;
}

```

11.2 Program Data

Data for example 2

Values of n and m

4 6

Values of ncnln, nonln and njnln

3 4 2

Values of nnz and iobj

14 6

Columns and rows names

'Varble 1' 'Varble 2' 'Varble 3' 'Varble 4' 'NlnCon 1'
 'NlnCon 2' 'NlnCon 3' 'LinCon 1' 'LinCon 2' 'Free Row'

Matrix nonzeros: value, row index, column index

```

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

Lower bounds
-0.55   -0.55   0.0   0.0   -894.8   -894.8   -1294.8   -0.55
-0.55   -1.0E+25

Upper bounds
0.55   0.55   1200.0   1200.0   -894.8   -894.8   -1294.8   1.0E+25
1.0E+25  1.0E+25

Initial estimate of X
0.0  0.0  0.0  0.0

Begin e04ugc
minor_iter_lim = 20
iter_lim = 30
End

```

11.3 Program Results

Example 2: Use of the option structure.

Optional parameter setting for e04ugc.

```

-----
Option file: stdin

minor_iter_lim = 20
iter_lim = 30

Parameters to e04ugc
-----

Frequencies.
fcheck..... 60      expand_freq..... 10000
factor_freq..... 50

QP subproblems.
scale_tol..... 9.00e-01  minor_feas_tol..... 1.05e-08
scale_opt..... 1      minor_opt_tol..... 1.05e-08
part_price..... 1      crash_tol..... 1.00e-01
pivot_tol..... 2.04e-11  minor_print_level.... Nag_NoPrint
crash..... Nag_NoCrash  elastic_wt..... 1.00e+02

Derivatives.
obj_deriv..... TRUE      con_deriv..... TRUE
verify_grad..... Nag_SimpleCheck  print_deriv..... Nag_D_Print
Start obj check at col.. 1      Stop obj check at col.. 4
Start con check at col.. 1      Stop con check at col.. 2

The SQP method.
direction..... Nag_Minimize
Nonlinear objective vars 4      major_opt_tol..... 1.05e-08
f_prec..... 1.72e-13  inf_step..... 1.00e+20
max_sb..... 4      f_diff_int..... 4.15e-07
unbounded_obj..... 1.00e+15  c_diff_int..... 5.56e-05
major_step_lim..... 2.00e+00  deriv_linesearch..... FALSE
print_level..... Nag_Soln_Iter  major_iter_lim..... 100
linesearch_tol..... 9.00e-01  minor_iter_lim..... 20

```

```

inf_bound..... 1.00e+20      iter_lim..... 30

Hessian approximation.
hess_storage..... Nag_HessianFull      hess_update..... 20
hess_freq..... 99999999

Nonlinear constraints.
Nonlinear constraints... 3      major_feas_tol..... 1.05e-08
Nonlinear Jacobian vars. 2      violation_limit..... 1.00e+01

Miscellaneous.
Variables..... 4      Linear constraints.... 3
Nonlinear variables.... 4      Linear variables..... 0
lu_factor_tol..... 5.00e+00      lu_sing_tol..... 2.04e-11
lu_update_tol..... 5.00e+00      lu_den_tol..... 6.00e-01
eps (machine precision). 1.11e-16
start..... Nag_Cold      feas_exit..... FALSE
Names..... supplied      print_80ch..... TRUE
outfile..... stdout

Memory allocation.
nz_coef..... 5.00e+00      Initial sizes of work arrays.
state..... Nag      Integers..... 1628
lambda..... Nag      Reals..... 1258

```

```

XXX Scale option reduced from          1 to          0.
XXX Feasible linear rows.
XXX Norm(x-x0) minimized. Sum of infeasibilities = 0.00e+00.

```

```

confun sets      6 out of      6 constraint gradients.
objfun sets      4 out of      4 objective gradients.

```

Verification of constraint gradients returned by subroutine confun

Cheap test on confun...

The Jacobian seems to be OK.

The largest discrepancy was 4.41e-08 in constraint 2.

Verification of objective gradients returned by subroutine objfun

Cheap test on objfun...

The objective gradients seem to be OK.

```

Gradient projected in two directions 0.0000000000e+00 0.0000000000e+00
Difference approximations           1.74111992322e-19 4.48742248252e-21

```

XXX All-slack basis B = I selected.

XXX Large multipliers.

Elastic mode started with weight = 2.0e+02.

Maj	Mnr	Step	Merit Function	Feasibl	Optimal	Cond	Hz	PD		
0	8	0.0e+00	3.199952e+05	1.7e+00	8.0e-01	1.0e+00	FF		R	i
1	4	1.0e+00	2.419951e+05	7.7e-01	3.7e+01	1.0e+00	FF		s	

2	1	1.0e+00	5.172188e+03	2.9e-02	1.9e+00	1.0e+00	FF
3	1	3.8e-01	5.038588e+03	1.8e-02	4.6e+00	1.2e+02	FF
4	1	3.6e-01	5.120830e+03	1.2e-02	2.0e+00	1.6e+02	FF
5	1	1.0e+00	5.126491e+03	1.0e-04	3.6e-02	1.7e+02	FF
6	1	3.1e-01	5.126498e+03	7.1e-05	3.1e-02	1.1e+02	FF
7	1	1.0e+00	5.126498e+03	4.3e-09	6.5e-04	1.1e+02	TF
8	1	1.0e+00	5.126498e+03	4.5e-13	4.8e-05	1.1e+02	TF
9	0	1.0e+00	5.126498e+03	1.2e-15	3.5e-13	1.1e+02	TT

Exit from NP problem after 9 major iterations,
19 minor iterations.

Variable	State	Value	Lower Bound	Upper Bound	Lagr Mult	Residual
Varble 1	BS	1.188764e-01	-5.50000e-01	5.50000e-01	-4.5475e-13	4.3112e-01
Varble 2	BS	-3.962336e-01	-5.50000e-01	5.50000e-01	1.6371e-11	1.5377e-01
Varble 3	BS	6.799453e+02	0.00000e+00	1.20000e+03	-3.0420e-14	5.2005e+02
Varble 4	SBS	1.026067e+03	0.00000e+00	1.20000e+03	-6.9278e-14	1.7393e+02

Constrnt	State	Value	Lower Bound	Upper Bound	Lagr Mult	Residual
NlnCon 1	EQ	-8.948000e+02	-8.94800e+02	-8.94800e+02	-4.3870e+00	3.4106e-12
NlnCon 2	EQ	-8.948000e+02	-8.94800e+02	-8.94800e+02	-4.1056e+00	6.8212e-13
NlnCon 3	EQ	-1.294800e+03	-1.29480e+03	-1.29480e+03	-5.4633e+00	3.1832e-12
LinCon 1	BS	2.773572e-01	-5.50000e-0	None 25	0.0000e+00	8.2736e-01
LinCon 2	BS	-2.773572e-01	-5.50000e-0	None 25	0.0000e+00	2.7264e-01
Free Row	BS	4.091970e+03	None	None 25	-1.0000e+00	4.0920e+03

Exit e04ugc - Optimal solution found.

Final objective value = 5126.498

A run of the same example with a warm start:

Parameters to e04ugc

Frequencies.

fcheck..... 60 expand_freq..... 10000
factor_freq..... 50

QP subproblems.

scale_tol..... 9.00e-01 minor_feas_tol..... 1.05e-08
scale_opt..... 1 minor_opt_tol..... 1.05e-08
part_price..... 1 crash_tol..... 1.00e-01
pivot_tol..... 2.04e-11 minor_print_level.... Nag_NoPrint
crash..... Nag_NoCrash elastic_wt..... 1.00e+02

Derivatives.

obj_deriv..... TRUE con_deriv..... TRUE
verify_grad..... Nag_SimpleCheck print_deriv..... Nag_D_NoPrint
Start obj check at col.. 1 Stop obj check at col.. 4
Start con check at col.. 1 Stop con check at col.. 2

The SQP method.

direction..... Nag_Minimize
Nonlinear objective vars 4 major_opt_tol..... 1.05e-08

f_prec.....	1.72e-13	inf_step.....	1.00e+20
max_sb.....	4	f_diff_int.....	4.15e-07
unbounded_obj.....	1.00e+15	c_diff_int.....	5.56e-05
major_step_lim.....	2.00e+00	deriv_linesearch.....	FALSE
print_level.....	Nag_Iter	major_iter_lim.....	100
linesearch_tol.....	9.00e-01	minor_iter_lim.....	20
inf_bound.....	1.00e+20	iter_lim.....	30
Hessian approximation.			
hess_storage.....	Nag_HessianFull	hess_update.....	20
hess_freq.....	99999999		
Nonlinear constraints.			
Nonlinear constraints...	3	major_feas_tol.....	1.05e-08
Nonlinear Jacobian vars.	2	violation_limit.....	1.00e+01
Miscellaneous.			
Variables.....	4	Linear constraints.....	3
Nonlinear variables.....	4	Linear variables.....	0
lu_factor_tol.....	5.00e+00	lu_sing_tol.....	2.04e-11
lu_update_tol.....	5.00e+00	lu_den_tol.....	6.00e-01
eps (machine precision).	1.11e-16	feas_exit.....	FALSE
start.....	Nag_Warm	print_80ch.....	TRUE
Names.....	supplied		
outfile.....	stdout		
Memory allocation.			
nz_coef.....	5.00e+00	Initial sizes of work arrays.	
state.....	Nag	Integers.....	1628
lambda.....	Nag	Reals.....	1258

XXX Scale option reduced from 1 to 0.
 XXX Feasible linear rows.
 XXX Norm(x-x0) minimized. Sum of infeasibilities = 0.00e+00.
 XXX All-slack basis B = I selected.

Maj	Mnr	Step	Merit Function	Feasibl	Optimal	Cond	Hz	PD	
0	1	0.0e+00	5.128197e+03	1.3e-01	1.1e+00	1.7e+00	FF	R	
1	1	1.0e+00	4.883655e+03	3.5e-03	5.7e-01	2.0e+02	FF	S	
2	1	1.8e-01	5.126320e+03	2.8e-03	3.7e+00	1.9e+02	FF		
3	1	2.8e-01	5.126417e+03	2.0e-03	1.1e+00	1.9e+02	FF		
4	1	1.0e+00	5.126499e+03	3.2e-06	1.9e-01	2.0e+02	FF		
5	1	1.0e+00	5.126498e+03	1.1e-08	4.2e-02	1.1e+02	FF		
6	1	1.0e+00	5.126498e+03	2.2e-09	1.2e-06	1.1e+02	TF		
7	1	1.0e+00	5.126498e+03	8.1e-17	2.7e-10	1.1e+02	TT		

Exit from NP problem after 7 major iterations,
 8 minor iterations.

Exit e04ugc - Optimal solution found.

Final objective value = 5126.498