NAG C Library Function Document

nag_opt_nlp_solve (e04wdc)

Note: *this function uses* **optional arguments** *to define choices in the problem specification and in the details of the algorithm. If you wish to use* default *settings for all of the optional arguments, you need only read Sections 1 to 9 of this document. Refer to the additional Sections 10, 11 and 12 for a detailed description of the algorithm, the specification of the optional arguments and a description of the monitoring information produced by the function*.

1 Purpose

nag_opt_nlp_solve (e04wdc) is designed to minimize an arbitrary smooth function subject to constraints (which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints) using a sequential quadratic programming (SQP) method. As many first derivatives as possible should be supplied by you; any unspecified derivatives are approximated by finite differences. It is not intended for large sparse problems.

nag_opt_nlp_solve (e04wdc) may also be used for unconstrained, bound-constrained and linearly constrained optimization.

nag_opt_nlp_solve (e04wdc) uses **forward communication** for evaluating the objective function, the nonlinear constraint functions, and any of their derivatives.

The initialization function nag_opt_nlp_init (e04wcc) **must** have been called prior to calling nag_opt_nlp_solve (e04wdc).

2 Specification

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

```
void nag_opt_nlp_solve (Integer n, Integer nclin, Integer ncnln, Integer pda,
     Integer pdcj, Integer pdh, const double a[], const double bl[],
     const double bu[],
     void (*confun)(Integer *mode, Integer ncnln, Integer n, Integer pdcj,
          const Integer needc[], const double x[], double ccon[], double cjac[],
          Integer nstate, Nag_Comm *comm),
     void (*objfun)(Integer *mode, Integer n, const double x[], double *objf,
          double grad[], Integer nstate, Nag_Comm *comm),
     Integer *majits, Integer istate[], double ccon[], double cjac[],
     double clamda[], double *objf, double grad[], double hess[], double x[],
     Nag_E04State *state, Nag_Comm *comm, NagError *fail)
```

Before calling nag_opt_nlp_solve (e04wdc), or any of the option setting functions nag_opt_nlp_option_set_file (e04wec), nag_opt_nlp_option_set_string (e04wfc), nag_opt_nlp_option_set_integer (e04wgc) or nag_opt_nlp_option_set_double (e04whc), nag_opt_nlp_init (e04wcc) **must** be called. The specification for nag_opt_nlp_init (e04wcc) is:

```
void nag_opt_nlp_init (Nag_E04State *state, NagError *fail)
```

The contents of **state must not** be altered between calls of the functions nag_opt_nlp_init (e04wcc), nag_opt_nlp_solve (e04wdc), nag_opt_nlp_option_set_file (e04wec), nag_opt_nlp_option_set_integer (e04wgc) or nag_opt_nlp_option_set_double (e04whc).

3 Description

nag_opt_nlp_solve (e04wdc) is designed to solve nonlinear programming problems – the minimization of a smooth nonlinear function subject to a set of constraints on the variables. nag_opt_nlp_solve (e04wdc) is suitable for small dense problems. The problem is assumed to be stated in the following form:

$$\underset{x \in R^n}{\text{minimize}}\, F(x) \quad \text{subject to} \quad l \leq \begin{pmatrix} x \\ A_L x \\ c(x) \end{pmatrix} \leq u, \tag{1}$$

where $F(x)$ (the *objective function*) is a nonlinear function, A_L is an n_L by n constant matrix, and $c(x)$ is an n_N element vector of nonlinear constraint functions. (The matrix A_L and the vector $c(x)$ may be empty.) The objective function and the constraint functions are assumed to be smooth, i.e., at least twice-continuously differentiable. (The method of nag_opt_nlp_solve (e04wdc) will usually solve (1) if there are only isolated discontinuities away from the solution.)

Note that although the bounds on the variables could be included in the definition of the linear constraints, we prefer to distinguish between them for reasons of computational efficiency. For the same reason, the linear constraints should **not** be included in the definition of the nonlinear constraints. Upper and lower bounds are specified for all the variables and for all the constraints. An *equality* constraint can be specified by setting $l_i = u_i$. 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 option **Infinite Bound Size** in Section 11.2.)

A typical invocation of nag_opt_nlp_solve (e04wdc) would be:

```
e04wcc (&state, ...);
e04wec (ispecs, &state, ...);
e04wdc (n, nclin, ncnln, ...);
```

where nag_opt_nlp_option_set_file (e04wec) reads a file of optional definitions.

Table 1 illustrates the feasible region for the jth pair of constraints $\ell_j \leq r_j(x) \leq u_j$. The quantity of δ is the option **Feasibility Tolerance**, which can be set by you (see Section 11). The constraints $\ell_j \leq r_j \leq u_j$ are considered 'satisfied' if r_j lies in Regions 2, 3 or 4, and 'inactive' if r_j lies in Region 3. The constraint $r_j \geq l_j$ is considered 'active' in Region 2, and 'violated' in Region 1. Similarly, $r_j \leq u_j$ is active in Region 4, and violated in Region 5. For equality constraints ($\ell_j = u_j$), Regions 2 and 4 are the same and Region 3 is empty.

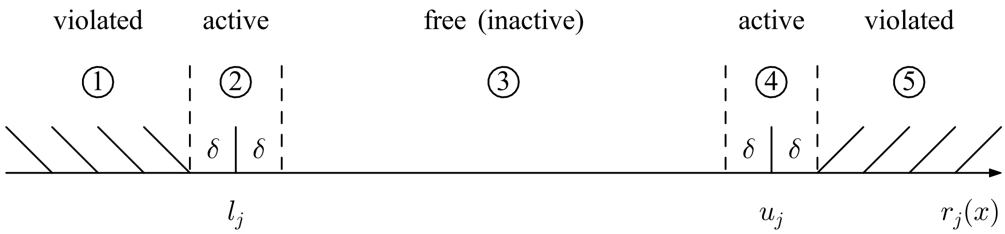

Table 1
Illustration of the constraints $\ell_j \leq r_j(x) \leq u_j$

If there are no nonlinear constraints in (1) and F is linear or quadratic, then it will generally be more efficient to use one of nag_opt_lp (e04mfc), nag_opt_lin_lsq (e04ncc) or nag_opt_qp (e04nfc). If the problem is large and sparse and does have nonlinear constraints, then nag_opt_sparse_nlp_solve (e04vhc) should be used, since nag_opt_nlp_solve (e04wdc) treats all matrices as dense.

You must supply an initial estimate of the solution to (1), together with functions that define $F(x)$, $c(x)$ and as many first partial derivatives as possible; unspecified derivatives are approximated by finite differences.

The objective function is defined by function **objfun**, and the nonlinear constraints are defined by function **confun**. On every call, these functions must return appropriate values of the objective and nonlinear constraints. You should also provide the available partial derivatives. Any unspecified derivatives are approximated by finite differences; see Section 11.2 for a discussion of the option **Derivative Level**. Just before either **objfun** or **confun** is called, each element of the current gradient array **grad** or **cjac** is initialized to a special value. On exit, any element that retains the value is estimated by finite differences. Note that if there *are* any nonlinear constraints then the *first* call to **confun** will precede the *first* call to **objfun**.

For maximum reliability, it is preferable for you to provide all partial derivatives (see Chapter 8 of Gill *et al.* (1981), for a detailed discussion). If all gradients cannot be provided, it is similarly advisable to provide as many as possible. While developing the functions **objfun** and **confun**, the option **Verify Level** (see Section 11.2) should be used to check the calculation of any known gradients.

The method used by nag_opt_nlp_solve (e04wdc) is described in detail in Section 10.

4 References

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

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

Gill P E, Murray W and Saunders M A (1999) Users' guide for SQOPT 5.3: a Fortran package for large-scale linear and quadratic programming *Report SOL 99* Department of Operations Research, Stanford University

Gill P E, Murray W and Saunders M A (2002) *SNOPT: An SQP Algorithm for Large-scale Constrained Optimization* **12** 979–1006 SIAM J. Optim.

Gill P E, Murray W, Saunders M A and Wright M H (1986c) Users' guide for NPSOL (Version 4.0): a Fortran package for nonlinear programming *Report SOL 86-2* Department of Operations Research, Stanford University

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 and Wright M H (1981) *Practical Optimization* Academic Press

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

5 Arguments

Note 1: In the following specification of nag_opt_nlp_solve (e04wdc), we define $r(x)$ as the vector of combined constraint functions

$$r(x) = \begin{pmatrix} x \\ A_L x \\ c(x) \end{pmatrix},$$

and use $nctotl$ to denote a variable that holds its dimension $\mathbf{n} + \mathbf{nclin} + \mathbf{ncnln}$.

1: **n** – Integer *Input*

On entry: n, the number of variables.

Constraint: $\mathbf{n} > 0$.

2: **nclin** – Integer *Input*

On entry: n_L, the number of general linear constraints.

Constraint: $\mathbf{nclin} \geq 0$.

3: **ncnln** – Integer *Input*

On entry: n_N, the number of nonlinear constraints.

Constraint: $\mathbf{ncnln} \geq 0$.

4: **pda** – Integer *Input*

On entry: the stride separating column elements in the array **a**.

Constraints:

if **nclin** > 0, **pda** $\geq$ **n**;
otherwise **pda** ≥ 1.

5: **pdcj** – Integer *Input*

On entry: the stride separating column elements in the array **cjac**.

Constraints:

if **ncnln** > 0, **pdcj** $\geq$ **n**;
otherwise **pdcj** ≥ 1.

6: **pdh** – Integer *Input*

On entry: the stride separating column elements in the array **hess**.

Constraint: **pdh** $\geq$ **n**

7: **a**[dim] – const double *Input*

Note: the dimension, *dim*, of the array **a** must be at least

max(1, **nclin** $\times$ **pda**) when **nclin** > 0.

The (i,j)th element of the matrix A is stored in **a**$[(i-1) \times$ **pda** $+ j - 1]$.

On entry: the ith row of the array **a** must contain the ith row of the matrix A_L of general linear constraints in (1). That is, the ith row contains the coefficients of the ith general linear constraint, for $i = 1, 2, \ldots,$ **nclin**.

If **nclin** $= 0$ then the array **a** is not referenced.

8: **bl**[$nctotl$] – const double *Input*
9: **bu**[$nctotl$] – const double *Input*

On entry: **bl** must contain the lower bounds and **bu** the upper bounds for all the constraints in the following order. The first n elements of each array must contain the bounds on the variables, the next n_L elements the bounds for the general linear constraints (if any) and the next n_N elements the bounds for the general nonlinear constraints (if any). To specify a non-existent lower bound (i.e., $l_j = -\infty$), set **bl**$[j-1] \leq -bigbnd$, and to specify a non-existent upper bound (i.e., $u_j = +\infty$), set **bu**$[j-1] \geq bigbnd$; where $bigbnd$ is the optional argument **Infinite Bound Size** (see Section 11.2). To specify the jth constraint as an *equality*, set **bl**$[j-1] =$ **bu**$[j-1] = \beta$, say, where $|\beta| < bigbnd$.

Constraints:

bl$[j-1] \leq$ **bu**$[j-1]$, for $j = 1, 2, \ldots, nctotl$;
if **bl**$[j-1] =$ **bu**$[j-1] = \beta$, $|\beta| < bigbnd$.

10: **confun** – function, supplied by the user *External Function*

confun must calculate the vector $c(x)$ of nonlinear constraint functions and (optionally) its Jacobian $\left(= \frac{\partial c}{\partial x}\right)$ for a specified n element vector x. If there are no nonlinear constraints (i.e., **ncnln** $= 0$), **confun** will never be called by nag_opt_nlp_solve (e04wdc) and **confun** may be the dummy function e04wdp. (e04wdp is included in the NAG C Library and so need not be supplied by you. If there are nonlinear constraints, the first call to **confun** will occur before the first call to **objfun**.

If all constraint gradients (Jacobian elements) are known (i.e., **Derivative Level** $= 2$ or 3 (see Section 11.2), any *constant* elements may be assigned to **cjac** once only at the start of the optimization. An element of **cjac** that is not subsequently assigned in **confun** will retain its initial value throughout. Constant elements may be loaded in **cjac** during the first call to **confun** (signalled by the value of **nstate** $= 1$). The ability to preload constants is useful when many Jacobian elements are identically zero, in which case **cjac** may be initialized to zero and non-zero elements may be reset by **confun**.

It must be emphasised that, if **Derivative Level** < 2, unassigned elements of **cjac** are *not* treated as constant; they are estimated by finite differences, at non-trivial expense.

Its specification is:

```
void confun (Integer *mode, Integer ncnln, Integer n, Integer pdcj,
     const Integer needc[], const double x[], double ccon[], double cjac[],
     Integer nstate, Nag_Comm *comm)
```

1: **mode** – Integer * *Input/Output*

On entry: is set by nag_opt_nlp_solve (e04wdc) to indicate which values must be assigned during each call of **confun**. Only the following values need be assigned, for each value of i such that **needc**$[i] > 0$:

mode $= 0$

The components of **ccon** corresponding to positive values in **needc** must be set. Other compoments and the array **cjac** are ignored.

mode $= 1$

The known components of the rows of **cjac** corresponding to positive values in **needc** must be set. Other rows of **cjac** and the array **ccon** will be ignored.

mode $= 2$

Only the elements of **ccon** corresponding to positive values of **needc** need to be set (and similarly for the known components of the rows of **cjac**).

On exit: may be used to indicate that you are unable or unwilling to evaluate the constraint functions at the current x.

During the linesearch, the constraint functions are evaluated at points of the form $x = x_k + \alpha p_k$ after they have already been evaluated satisfactorily at x_k. At any such α, if you set **mode** $= -1$, nag_opt_nlp_solve (e04wdc) will evaluate the functions at some point closer to x_k (where they are more likely to be defined).

If for some reason you wish to terminate the current problem, set **mode** < -1.

2: **ncnln** – Integer *Input*

On entry: n_N, the number of nonlinear constraints.

3: **n** – Integer *Input*

On entry: n, the number of variables.

4: **pdcj** – Integer *Input*

On entry: the stride separating column elements in the array **cjac**.

5: **needc**[**ncnln**] – const Integer *Input*

On entry: the indices of the elements of **ccon** and/or **cjac** that must be evaluated by **confun**. If **needc**$[i-1] > 0$ then the ith element of **ccon** and/or the available elements of the ith row of **cjac** (see argument **mode** above) must be evaluated at x.

6: **x**[**n**] – const double *Input*

On entry: x, the vector of variables at which the constraint functions and/or the available elements of the constraint Jacobian are to be evaluated.

7: **ccon**[**ncnln**] – double *Output*

On exit: if **needc**$[i-1] > 0$ and **mode** $= 0$ or 2, **ccon**$[i-1]$ must contain the value of the ith constraint at x. The remaining elements of **ccon**, corresponding to the non-positive elements of **needc**, are ignored.

8: **cjac**[**pdcj** × **n**] – double *Input/Output*

Note: where **CJAC**(i,j) appears in this document, it refers to the array element **cjac**$[(i-1) \times$ **pdcj** $+ j - 1]$.

On entry: the elements of **cjac** are set to special values which enable nag_opt_nlp_solve (e04wdc) to detect whether they are reset by **confun**.

On exit: if **needc**$[i] > 0$ and **mode** $= 1$ or 2, the ith row of **cjac** must contain the available elements of the vector ∇c_i given by

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

where $\frac{\partial c_i}{\partial x_j}$ is the partial derivative of the ith constraint with respect to the jth variable, evaluated at the point x. See also the argument **nstate** below. The remaining rows of **cjac**, corresponding to non-positive elements of **needc**, are ignored.

If all elements of the constraint Jacobian are known (i.e., **Derivative Level** $= 2$ or 3, see Section 11.2), any constant elements may be assigned to **cjac** one time only at the start of the optimization. An element of **cjac** that is not subsequently assigned in **confun** will retain its initial value throughout. Constant elements may be loaded into **cjac** during the first call to **confun** (signalled by the value **nstate** $= 1$). The ability to preload constants is useful when many Jacobian elements are identically zero, in which case **cjac** may be initialized to zero and non-zero elements may be reset by **confun**.

Note that constant non-zero elements do affect the values of the constraints. Thus, if **CJAC**(i,j) is set to a constant value, it need not be reset in subsequent calls to **confun**, but the value **CJAC**$(i,j) \times$ **x**$[j-1]$ must nonetheless be added to **ccon**$[i-1]$. For example, if **CJAC**$(1,1) = 2$ and **CJAC**$(1,2) = -5$ then the term $2 \times$ **x**$[0] - 5 \times$ **x**$[1]$ must be included in the definition of **ccon**$[0]$.

It must be emphasised that, if **Derivative Level** $= 0$ or 1, unassigned elements of **cjac** are not treated as constant; they are estimated by finite differences, at non-trivial expense. If you do not supply a value for **Difference Interval** (see Section 11.2), an interval for each element of x is computed automatically at the start of the optimization. The automatic procedure can usually identify constant elements of **cjac**, which are then computed once only by finite differences.

9: **nstate** – Integer *Input*

On entry: if **nstate** $= 1$ then nag_opt_nlp_solve (e04wdc) is calling **confun** for the first time. This argument setting allows you to save computation time if certain data must be read or calculated only once.

10: **comm** – Nag_Comm * *Communication Structure*

Pointer to structure of type **Nag_Comm**; the following members are relevant to **confun**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_opt_nlp_solve (e04wdc) these pointers may be allocated memory by the user and initialized with various quantities for use by **confun** when called from nag_opt_nlp_solve (e04wdc).

confun should be tested separately before being used in conjunction with nag_opt_nlp_solve (e04wdc). See also the option **Verify Level** in Section 11.2.

11: **objfun** – function, supplied by the user *External Function*

objfun must calculate the objective function $F(x)$ and (optionally) its gradient $g(x) = \left(\frac{\partial F}{\partial x}\right)$ for a specified n element of vector x.

Its specification is:

```
void objfun (Integer *mode, Integer n, const double x[], double *objf,
     double grad[], Integer nstate, Nag_Comm *comm)
```

1: **mode** – Integer * *Input/Output*

On entry: is set by nag_opt_nlp_solve (e04wdc) to indicate which values must be assigned during each call of **objfun**. Only the following values need be assigned:

mode $= 0$

objf.

mode $= 1$

All available elements of **grad**.

mode $= 2$

objf and all available elements of **grad**.

On exit: may be used to indicate that you are unable or unwilling to evaluate the objective function at the current x.

During the linesearch, the function is evaluated at points of the form $x = x_k + \alpha p_k$ after they have already been evaluated satisfactorily at x_k. At any such α, if you set **mode** $= -1$, nag_opt_nlp_solve (e04wdc) will evaluate the functions at some point closer to x_k (where they are more likely to be defined).

If for some reason you wish to terminate solution of the current problem, set **mode** < -1.

2: **n** – Integer *Input*

On entry: n, the number of variables.

3: **x**[**n**] – const double *Input*

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

4: **objf** – double * *Output*

On exit: if **mode** $= 0$ or 2, **objf** must be set to the value of the objective function at x.

5: **grad**[**n**] – double *Input/Output*

On entry: is set to a special value.

On exit: if **mode** $= 1$ or 2, **grad** must return the available elements of the gradient evaluated at x, i.e., **grad**$[i-1]$ contains the partial derivative $\frac{\partial F}{\partial x_i}$.

6: **nstate** – Integer *Input*

On entry: if **nstate** $= 1$ then nag_opt_nlp_solve (e04wdc) is calling **objfun** for the first time. This argument setting allows you to save computation time if certain data must be read or calculated only once.

7: **comm** – Nag_Comm * *Communication Structure*

Pointer to structure of type **Nag_Comm**; the following members are relevant to **objfun**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_opt_nlp_solve (e04wdc) these pointers may be allocated memory by the user and initialized with various quantities for use by **objfun** when called from nag_opt_nlp_solve (e04wdc).

objfun should be tested separately before being used in conjunction with nag_opt_nlp_solve (e04wdc). See also the option **Verify Level** in Section 11.2.

12: **majits** – Integer * *Output*

On exit: the number of major iterations performed.

13: **istate**[$nctotl$] – Integer *Input/Output*

On entry: is an integer array that need not be initialized if nag_opt_nlp_solve (e04wdc) is called with a **Cold Start** (the default option).

For a **Warm Start**, every element of **istate** must be set. If nag_opt_nlp_solve (e04wdc) has just been called on a problem with the same dimensions, **istate** already contains valid values. Otherwise, **istate**[$j-1$] should indicate whether either of the constraints $r_j(x) \geq \ell_j$ or $r_j(x) \leq u_j$ is expected to be active.

The ordering of **istate** is the same as for **bl**, **bu** and $r(x)$, i.e., the first **n** components of **istate** refer to the upper and lower bounds on the variables, the next **nclin** refer to the bounds on $A_L x$, and the last **ncnln** refer to the bounds on $c(x)$. Possible values of **istate**[$i-1$] follow:

0 Neither $r_j(x) \geq \ell_j$ nor $r_j(x) \leq u_j$ is expected to be active.
1 $r_j(x) \approx \ell_j$ is expected to be active.
2 $r_j(x) \approx u_j$ is expected to be active.
3 This may be used if $\ell_j = u_j$. Normally an equality constraint $r_j(x) = \ell_j = u_j$ is active at a solution.

The values 1, 2 or 3 all have the same effect when **bl**[$j-1$] = **bu**[$j-1$]. If necessary, nag_opt_nlp_solve (e04wdc) will override your specification of **istate**, so that a poor choice will not cause the algorithm to fail.

On exit: describes the status of the constraints $\ell \leq r(x) \leq u$. For the jth lower or upper bound, $j = 1, \ldots, nctotl$, the possible values of **istate**[$j-1$] are as follows (see Table 1). δ is the appropriate feasibility tolerance.

-2 (Region 1) The lower bound is violated by more than δ.
-1 (Region 5) The upper bound is violated by more than δ.
0 (Region 3) Both bounds are satisfied by more than δ.
1 (Region 2) The lower bound is active (to within δ).
2 (Region 4) The upper bound is active (to within δ).
3 (Region 2 = Region 4) The bounds are equal and the equality constraint is satisfied (to within δ).

These values of **istate** are labelled in the printed solution according to Table 2.

Region	1	2	3	4	5	$2 \equiv 4$
istate[j]	-2	1	0	2	-1	3

Printed solution	--	LL	FR	UL	++	EQ

Table 2
Labels used in the printed solution for the regions in Table 1

14: **ccon**[*dim*] – double *Output*

Note: the dimension, *dim*, of the array **ccon** must be at least max(1, **ncnln**).

On exit: if **ncnln** > 0, **ccon**$[i-1]$ contains the value of the ith nonlinear constraint function c_i at the final iterate, for $i = 1, 2, \ldots,$ **ncnln**.

If **ncnln** $= 0$ then the array **ccon** is not referenced.

15: **cjac**[*dim*] – double *Input/Output*

Note: the dimension, *dim*, of the array **cjac** must be at least

max(1, **ncnln** × **pdcj**) when **ncnln** > 0.

Where **CJAC**(i,j) appears in this document, it refers to the array element **cjac**$[(i-1) \times$ **pdcj** $+ j - 1]$.

On entry: in general, **cjac** need not be initialized before the call to nag_opt_nlp_solve (e04wdc). However, if **Derivative Level** $= 3$ (the default; see Section 11.2), any constant elements of **cjac** may be initialized. Such elements need not be reassigned on subsequent calls to **confun**.

On exit: if **ncnln** > 0, **cjac** contains the Jacobian matrix of the nonlinear constraint functions at the final iterate, i.e., **CJAC**(i,j) contains the partial derivative of the ith constraint function with respect to the jth variable, for $i = 1, 2, \ldots,$ **ncnln**; $j = 1, 2, \ldots,$ **n**. (See the discussion of argument **cjac** under **confun**.)

If **ncnln** $= 0$ then the array **cjac** is not referenced.

16: **clamda**[*nctotl*] – double *Input/Output*

On entry: need not be set if the (default) **Cold Start** option is used.

If the **Warm Start** option has been chosen (see Section 11.2), **clamda**$[j-1]$ must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by the **istate** array (as above), for $j =$ **n** + **nclin** $+ 1,$ **n** + **nclin** $+ 2, \ldots, nctotl$. The remaining elements need not be set. Note that if the jth constraint is defined as 'inactive' by the initial value of the **istate** array (i.e., **istate**$[j-1] = 0$), **clamda**$[j-1]$ should be zero; if the jth constraint is an inequality active at its lower bound (i.e., **istate**$[j-1] = 1$), **clamda**$[j-1]$ should be non-negative; if the jth constraint is an inequality active at its upper bound (i.e., **istate**$[j-1] = 2$), **clamda**$[j-1]$ should be non-positive. If necessary, the function will modify **clamda** to match these rules.

On exit: the values of the QP multipliers from the last QP subproblem. **clamda**$[j-1]$ should be non-negative if **istate**$[j-1] = 1$ and non-positive if **istate**$[j-1] = 2$.

17: **objf** – double * *Output*

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

18: **grad**[**n**] – double *Output*

On exit: the gradient of the objective function at the final iterate (or its finite difference approximation).

19: **hess**[*dim*] – double *Input/Output*

Note: the dimension, *dim*, of the array **hess** must be at least max(1, **pdh** × **n**).

Where **HESS**(i,j) appears in this document, it refers to the array element **hess**$[(i-1) \times$ **pdh** $+ j - 1]$.

On entry: **hess** need not be initialized if the (default) **Cold Start** option is used and will be set to the identity.

For a **Warm Start**, **hess** provides the initial approximation of the Hessian of the Lagrangian, i.e., $\mathbf{HESS}(i,j) \approx \frac{\partial^2 \mathcal{L}(x,\lambda)}{\partial x_i \partial x_j}$, where $\mathcal{L}(x,\lambda) = F(x) - c(x)^{\mathrm{T}}\lambda$ and λ is an estimate of the Lagrange-multipliers. **hess** must be a positive-definite matrix.

On exit: contains the Hessian of the Lagrangian at the final estimate x.

20: **x**[**n**] – double *Input/Output*

On entry: an initial estimate of the solution.

On exit: the final estimate of the solution.

21: **state** – Nag_E04State * *Communication Structure*

Note: **state** is a NAG defined type (see Section 2.2.1.1 of the Essential Introduction).

state contains internal information required for functions in this suite. It must not be modified in any way.

22: **comm** – Nag_Comm * *Communication Structure*

The NAG communication argument (see Section 2.2.1.1 of the Essential Introduction).

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

The NAG error argument (see Section 2.6 of the Essential Introduction).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Internal error: memory allocation failed when attempting to allocate workspace sizes $\langle value \rangle$ and $\langle value \rangle$.

NE_ALLOC_INSUFFICIENT

Internal memory allocation was insufficient. Please contact NAG.

NE_BAD_PARAM

Basis file dimensions do not match this problem.

NE_BASIS_FAILURE

An error has occurred in the basis package, perhaps indicating incorrect setup of arrays. Set the optional argument **Print File** and examine the output carefully for further information.

NE_DERIV_ERRORS

User-supplied function computes incorrect constraint derivatives.

User-supplied function computes incorrect objective derivatives.

NE_E04WCC_NOT_INIT

Initialization function nag_opt_nlp_init (e04wcc) has not been called.

NE_INT

On entry, $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{n} > 0$.

On entry, $\mathbf{nclin} = \langle value \rangle$.
Constraint: $\mathbf{nclin} \geq 0$.

On entry, **ncnln** $= \langle value \rangle$.
Constraint: **ncnln** ≥ 0.

On entry, **pda** $= \langle value \rangle$.
Constraint: **pda** > 0.

On entry, **pdcj** $= \langle value \rangle$.
Constraint: **pdcj** > 0.

On entry, **pdh** $= \langle value \rangle$.
Constraint: **pdh** > 0.

NE_INT_2

On entry, **nclin** $= \langle value \rangle$, **n** $= \langle value \rangle$.
Constraint: **n** > 0.

On entry, **ncnln** $= \langle value \rangle$, **n** $= \langle value \rangle$.
Constraint: **n** > 0.

On entry, **pda** $<$ **nclin**. **pda** $= \langle value \rangle$, **nclin** $= \langle value \rangle$.

On entry, **pdcj** $<$ **ncnln**. **pdcj** $= \langle value \rangle$, **ncnln** $= \langle value \rangle$.

On entry, **pdh** $<$ **n**. **pdh** $= \langle value \rangle$, **n** $= \langle value \rangle$.

On entry, **pdh** $= \langle value \rangle$, **n** $= \langle value \rangle$.
Constraint: **pdh** $\geq$ **n**.

NE_INT_3

On entry, **pda** $= \langle value \rangle$, **nclin** $= \langle value \rangle$, **n** $= \langle value \rangle$.
Constraint: if **nclin** > 0, **pda** $\geq$ **n**;
otherwise **pda** ≥ 1.

On entry, **pdcj** $= \langle value \rangle$, **ncnln** $= \langle value \rangle$, **n** $= \langle value \rangle$.
Constraint: if **ncnln** > 0, **pdcj** $\geq$ **n**;
otherwise **pdcj** ≥ 1.

NE_INTERNAL_ERROR

An unexpected error has occurred. Set the optional argument **Print File** and examine the output carefully for further information.

NE_NOT_REQUIRED_ACC

The requested accuracy could not be achieved.

NE_NUM_DIFFICULTIES

Numerical difficulties have been encountered and no further progress can be made.

NE_REAL_2

On entry, bounds **bl** and **bu** for $\langle value \rangle$ are equal and infinite. **bl** $=$ **bu** $= \langle value \rangle$, $bigbnd = \langle value \rangle$.

On entry, bounds **bl** and **bu** for $\langle value \rangle \langle value \rangle$ are equal and infinite. **bl** $=$ **bu** $= \langle value \rangle$, $bigbnd = \langle value \rangle$.

On entry, bounds for $\langle value \rangle$ are inconsistent. **bl** $= \langle value \rangle$, **bu** $= \langle value \rangle$.

On entry, bounds for $\langle value \rangle \langle value \rangle$ are inconsistent. **bl** $= \langle value \rangle$, **bu** $= \langle value \rangle$.

NE_UNBOUNDED

The problem appears to be unbounded. The constraint violation limit has been reached.

The problem appears to be unbounded. The objective function is unbounded.

NE_USER_STOP

User-supplied constraint function requested termination.

User-supplied objective function requested termination.

NE_USRFUN_UNDEFINED

Unable to proceed into undefined region of user-supplied function.

User-supplied function is undefined at the first feasible point.

User-supplied function is undefined at the initial point.

NW_NOT_FEASIBLE

The linear constraints appear to be infeasible.

The problem appears to be infeasible. Infeasibilites have been minimized.

The problem appears to be infeasible. Nonlinear infeasibilites have been minimized.

The problem appears to be infeasible. The linear equality constraints could not be satisfied.

NW_TOO_MANY_ITER

Iteration limit reached.

Major iteration limit reached.

The superbasics limit is too small.

7 Accuracy

If **fail.code** = **NE_NOERROR** on exit, then the vector returned in the array **x** is an estimate of the solution to an accuracy of approximately **Major Optimality Tolerance** (see Section 11.2).

8 Further Comments

This section describes the final output produced by nag_opt_nlp_solve (e04wdc). Intermediate and other output are given in Section 12.

8.1 The Final Output

If **Print File** = 0, the final output, including a listing of status of every variable and constraint will be sent to the file descriptors associated with **Print File** (see Section 11.2). The following describes the output for each variable. A full stop (.) is printed for any numerical value that is zero.

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

`State` gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than the **Feasibility Tolerance** (see Section 11.2), `State` will be `++` or `--` respectively. (The latter situation can occur only when there is no feasible point for the bounds and linear constraints.)

A key is sometimes printed before `State` to give some additional information about the state of a variable.

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

`D` *Degenerate*. The variable is free, but it is equal to (or very close to) one of its bounds.

`I` *Infeasible*. The variable is currently violating one of its bounds by more than the **Feasibility Tolerance**.

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

`Lower bound` is the lower bound specified for the variable. `None` indicates that $\mathbf{bl}[j-1] \leq -bigbnd$.

`Upper bound` is the upper bound specified for the variable. `None` indicates that $\mathbf{bu}[j-1] \geq bigbnd$.

`Lagr multiplies` is the Lagrange-multiplier for the associated bound. This will be zero if `State` is `FR` unless $\mathbf{bl}[j-1] \leq -bigbnd$ and $\mathbf{bu}[j-1] \geq bigbnd$, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is `LL` and non-positive if `State` is `UL`.

`Slack` is the difference between the variable `Value` and the nearer of its (finite) bounds $\mathbf{bl}[j-1]$ and $\mathbf{bu}[j-1]$. A blank entry indicates that the associated variable is not bounded (i.e., $\mathbf{bl}[j-1] \leq -bigbnd$ and $\mathbf{bu}[j-1] \geq bigbnd$).

The meaning of the output for linear and nonlinear constraints is the same as that given above for variables, with $\mathbf{bl}[j-1]$ and $\mathbf{bu}[j-1]$ replaced by $\mathbf{bl}[n+j-1]$ and $\mathbf{bu}[n+j-1]$ respectively, and with the following changes in the heading:

`Linear constrnt` gives the name (`lincon`) and index j, for $j = 1, 2, \ldots, n_L$ of the linear constraint.

`Nonlin constrnt` gives the name (`nlncon`) and index $(j - n_L)$, for $j = n_L + 1, n_L + 2, \ldots, n_L + n_N$ of the nonlinear constraint.

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

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

9 Example

This is based on Problem 71 in Hock and Schittkowski (1981) and involves the minimization of the nonlinear function

$$F(x) = x_1 x_4 (x_1 + x_2 + x_3) + x_3$$

subject to the bounds

$$\begin{array}{c} 1 \leq x_1 \leq 5 \\ 1 \leq x_2 \leq 5 \\ 1 \leq x_3 \leq 5 \\ 1 \leq x_4 \leq 5 \end{array}$$

to the general linear constraint

$$x_1 + x_2 + x_3 + x_4 \leq 20,$$

and to the nonlinear constraints

$$\begin{array}{r} x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 40, \\ x_1 x_2 x_3 x_4 \geq 25. \end{array}$$

The initial point, which is infeasible, is

$$x_0 = (1, 5, 5, 1)^{\mathrm{T}},$$

and $F(x_0) = 16$.

The optimal solution (to five figures) is

$$x^{*} = (1.0, 4.7430, 3.8211, 1.3794)^{\mathrm{T}},$$

and $F(x^{*}) = 17.014$. One bound constraint and both nonlinear constraints are active at the solution.

9.1 Program Text

```
/* nag_opt_nlp_solve (e04wdc) Example Program.
 *
 * Copyright 2004 Numerical Algorithms Group.
 *
 * Mark 8, 2004.
 */
#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nage04.h>
#include <nagx04.h>

static void NAG_CALL confun(Integer *mode, Integer ncnln, Integer n,
                            Integer ldcj, const Integer needc[],
                            const double x[], double ccon[], double cjac[],
                            Integer nstate, Nag_Comm *comm);

static void NAG_CALL objfun(Integer *mode, Integer n, const double x[],
                            double *objf, double grad[], Integer nstate,
                            Nag_Comm *comm);

int main(void)
{

  /* Scalars */
  double objf;
  Integer exit_status, i, j, majits, n, nclin, ncnln, nctotal, pda, pdcj, pdh;

  /* Arrays */
  double *a=0, *bl=0, *bu=0, *ccon=0, *cjac=0, *clamda=0, *grad=0, *hess=0;
  double *x=0;
  Integer *istate=0;

  /*Nag Types*/
  Nag_E04State state;
  NagError fail;
  Nag_Comm comm;
  Nag_FileID fileid;

#define A(I,J) a[(I-1)*pda + J - 1]

  exit_status = 0;
  INIT_FAIL(fail);
  Vprintf("nag_opt_nlp_solve (e04wdc) Example Program Results\n");

  /* Skip heading in data file */
  Vscanf("%*[^\n] ");

  Vscanf("%ld%ld%ld%*[^\n] ",   &n,  &nclin,  &ncnln);
  if (n >0 && nclin >= 0 && ncnln >= 0)
    {
      /* Allocate memory */
      nctotal = n + nclin + ncnln;
      if ( !(a = NAG_ALLOC(ncnln*n, double)) ||
           !(bl = NAG_ALLOC(nctotal, double)) ||
           !(bu = NAG_ALLOC(nctotal, double)) ||
           !(ccon = NAG_ALLOC(ncnln, double)) ||
           !(cjac = NAG_ALLOC(ncnln*n, double)) ||
           !(clamda = NAG_ALLOC(nctotal, double)) ||
           !(grad = NAG_ALLOC(n, double)) ||
           !(hess = NAG_ALLOC(n*n, double)) ||
           !(x = NAG_ALLOC(n, double)) ||
```

```
      !(istate = NAG_ALLOC(nctotal, Integer))  )
    {
      Vprintf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
  pda = n;
  pdcj = n;
  pdh = n;

  /* Read a, bl, bu and x from data file */
  if (nclin > 0)
    {

      for (i = 1; i <= nclin; ++i)
        {
          for (j = 1; j <= n; ++j)
            {
              Vscanf("%lf", &A(i,j));
            }
        }
      Vscanf("%*[^\n] ");
    }

  for (i = 1; i <= n+nclin+ncnln; ++i)
    {
      Vscanf("%lf", &bl[i - 1]);
    }
  Vscanf("%*[^\n] ");

  for (i = 1; i <= n+nclin+ncnln; ++i)
    {
      Vscanf("%lf", &bu[i - 1]);
    }
  Vscanf("%*[^\n] ");

  for (i = 1; i <= n; ++i)
    {
      Vscanf("%lf", &x[i - 1]);
    }
  Vscanf("%*[^\n] ");

  /* Call nag_opt_nlp_init (e04wcc) to initialise e04wdc. */
  /* nag_opt_nlp_init (e04wcc).
   * Initialization function for nag_opt_nlp_solve (e04wdc)
   */
  nag_opt_nlp_init(&state, &fail);
  if (fail.code != NE_NOERROR)
    {
      Vprintf("Initialisation of nag_opt_nlp_init (e04wcc) failed.\n");
      exit_status = 1;
      goto END;
    }

  /* By default nag_opt_nlp_solve (e04wdc) does not print monitoring
   * information. Call nag_open_file (x04acc) to set the print file fileid.
   */
  /* nag_open_file (x04acc).
   * Open unit number for reading, writing or appending, and
   * associate unit with named file
   */
  nag_open_file("", 2, &fileid, &fail);
  if (fail.code != NE_NOERROR)
    {
      Vprintf("Fileid could not be obtained.\n");
      exit_status = 1;
      goto END;
    }
  /* nag_opt_nlp_option_set_integer (e04wgc).
   * Set a single option for nag_opt_nlp_solve (e04wdc) from
   * an integer argument
```

```
     */
    nag_opt_nlp_option_set_integer("Print file", fileid, &state, &fail);

    /* Solve the problem. */
    /* nag_opt_nlp_solve (e04wdc).
     * Solves the nonlinear programming (NP) problem
     */
    nag_opt_nlp_solve(n, nclin, ncnln, n, n, n, a, bl, bu,
                      confun, objfun, &majits, istate, ccon, cjac, clamda,
                      &objf, grad, hess, x, &state, &comm, &fail);

    if (fail.code == NE_NOERROR)
      {
        Vprintf("\n\nFinal objective value = %11.3f\n",   objf);

        Vprintf("Optimal X = ");

        for (i = 1; i <= n; ++i)
          {
            Vprintf("%9.2f%s", x[i - 1], i%7 == 0 || i == n ?"\n":" ");
          }

      }
    else
      {
        Vprintf ("Error message from nag_opt_nlp_solve (e04wdc) %s\n",
                 fail.message);
      }
  }
 END:
  if (a) NAG_FREE(a);
  if (bl) NAG_FREE(bl);
  if (bu) NAG_FREE(bu);
  if (ccon) NAG_FREE(ccon);
  if (cjac) NAG_FREE(cjac);
  if (clamda) NAG_FREE(clamda);
  if (grad) NAG_FREE(grad);
  if (hess) NAG_FREE(hess);
  if (x) NAG_FREE(x);
  if (istate) NAG_FREE(istate);


  return exit_status;
}

#undef A


static void objfun(Integer *mode, Integer n, const double x[],
                   double *objf, double grad[], Integer nstate,  Nag_Comm *comm)
{
  /* Routine to evaluate objective function and its 1st derivatives. */

  /* Function Body */
  if (*mode == 0 || *mode == 2)
    {
      *objf = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2];
    }

  if (*mode == 1 || *mode == 2)
    {
      grad[0] = x[3] * (x[0] * 2. + x[1] + x[2]);
      grad[1] = x[0] * x[3];
      grad[2] = x[0] * x[3] + 1.;
      grad[3] = x[0] * (x[0] + x[1] + x[2]);
    }

  return;
} /* objfun */
```

```
static void confun(Integer *mode, Integer ncnln, Integer n,
                   Integer pdcj, const Integer needc[], const double x[],
                   double ccon[], double cjac[], Integer nstate,
                   Nag_Comm *comm)
{
  /* Scalars */
  Integer i, j;

#define CJAC(I,J) cjac[(I-1)*pdcj + J-1]

  /* Routine to evaluate the nonlinear constraints and their 1st */
  /* derivatives. */


  /* Function Body */
  if (nstate == 1)
    {
      /* First call to confun.  Set all Jacobian elements to zero. */
      /* Note that this will only work when 'Derivative Level = 3' */
      /* (the default; see Section 11.2). */
      for (j = 1; j <= n; ++j)
        {
          for (i = 1; i <= ncnln; ++i)
            {
              CJAC(i, j) = 0.;
            }
        }
    }

  if (needc[0] > 0)
    {
      if (*mode == 0 || *mode == 2)
        {
          ccon[0] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] + x[3] * x[3];
        }
      if (*mode == 1 || *mode == 2)
        {
          CJAC(1, 1) = x[0] * 2.;
          CJAC(1, 2) = x[1] * 2.;
          CJAC(1, 3) = x[2] * 2.;
          CJAC(1, 4) = x[3] * 2.;
        }
    }

  if (needc[1] > 0)
    {
      if (*mode == 0 || *mode == 2)
        {
          ccon[1] = x[0] * x[1] * x[2] * x[3];
        }
      if (*mode == 1 || *mode == 2)
        {
          CJAC(2, 1) = x[1] * x[2] * x[3];
          CJAC(2, 2) = x[0] * x[2] * x[3];
          CJAC(2, 3) = x[0] * x[1] * x[3];
          CJAC(2, 4) = x[0] * x[1] * x[2];
        }
    }

  return;
} /* confun */

#undef CJAC
```

9.2 Program Data

```
nag_opt_nlp_solve (e04wdc) Example Program Data
  4   1   2                                                   : N, NCLIN and NCNLN
  1.0   1.0   1.0   1.0                                       : Matrix A
  1.0   1.0   1.0   1.0  -1.0E+25  -1.0E+25  25.0             : Lower bounds BL
  5.0   5.0   5.0   5.0  20.0      40.0      1.0E+25          : Upper bounds BU
  1.0   5.0   5.0   1.0                                       : Initial vector X
```

9.3 Program Results

```
nag_opt_nlp_solve (e04wdc) Example Program Results

 Parameters
 ==========

 Files
 -----
 Solution file..........         0       Old basis file ........         0       (Print file)...........         6
 Insert file............         0       New basis file ........         0       (Summary file).........         0
 Punch file.............         0       Backup basis file......         0
 Load file..............         0       Dump file..............         0

 Frequencies
 -----------
 Print frequency........       100       Check frequency........        60       Save new basis map.....       100
 Summary frequency......       100       Factorization frequency        50       Expand frequency.......     10000

 QP subproblems
 --------------
 QPsolver Cholesky......
 Scale tolerance........     0.900       Minor feasibility tol..  1.00E-06       Iteration limit........     10000
 Scale option...........         0       Minor optimality  tol..  1.00E-06       Minor print level......         1
 Crash tolerance........     0.100       Pivot tolerance........  2.05E-11       Partial price..........         1
 Crash option...........         3       Elastic weight.........  1.00E+04       Prtl price section ( A)         4
                                         New superbasics........        99       Prtl price section (-I)         3

 The SQP Method
 --------------
 Minimize...............                 Cold start.............                 Proximal Point method..         1
 Nonlinear objectiv vars         4       Major optimality tol...  2.00E-06       Function precision.....  1.72E-13
 Unbounded step size....  1.00E+20       Superbasics limit......         4       Difference interval....  4.15E-07
 Unbounded objective....  1.00E+15       Reduced Hessian dim....         4       Central difference int.  5.57E-05
 Major step limit.......  2.00E+00       Derivative linesearch..                 Derivative level.......         3
 Major iterations limit.      1000       Linesearch tolerance...   0.90000       Verify level...........         0
 Minor iterations limit.       500       Penalty parameter......  0.00E+00       Major Print Level......         1

 Hessian Approximation
 ---------------------
 Full-Memory Hessian....                 Hessian updates........  99999999       Hessian frequency......  99999999
                                                                                 Hessian flush..........  99999999

 Nonlinear constraints
 ---------------------
 Nonlinear constraints..         2       Major feasibility tol..  1.00E-06       Violation limit........  1.00E+06
 Nonlinear Jacobian vars         4

 Miscellaneous
 -------------
 LU factor tolerance....      1.10       LU singularity tol.....  2.05E-11       Timing level...........         0
 LU update tolerance....      1.10       LU swap tolerance......  1.03E-04       Debug level............         0
 LU partial  pivoting...                 eps (machine precision)  1.11E-16       System information.....        No


 Nonlinear constraints       2     Linear constraints       1
 Nonlinear variables         4     Linear variables         0
 Jacobian  variables         4     Objective variables      4
 Total constraints           3     Total variables          4
```

```
 The user has defined       8   out of       8   constraint gradients.
 The user has defined       4   out of       4   objective  gradients.

 Cheap test of user-supplied problem derivatives...

 The constraint gradients seem to be OK.

 -->  The largest discrepancy was    1.84E-07  in constraint     6


 The objective  gradients seem to be OK.

 Gradient projected in one direction   4.99993000077E+00
 Difference approximation              4.99993303727E+00



   Itns Major Minors    Step   nCon Feasible  Optimal  MeritFunction     L+U BSwap     nS  condHz Penalty
      2     0      2              1  1.7E+00  2.8E+00  1.6000000E+01       7             2 1.0E+00         _  r
      4     1      2 1.0E+00      2  1.3E-01  3.2E-01  1.7726188E+01       8             1 6.2E+00 8.3E-02 _n r
      5     2      1 1.0E+00      3  3.7E-02  1.7E-01  1.7099571E+01       7             1 2.0E+00 8.3E-02 _s
      6     3      1 1.0E+00      4  2.2E-02  1.1E-02  1.7014005E+01       7             1 1.8E+00 8.3E-02 _
      7     4      1 1.0E+00      5  1.5E-04  6.0E-04  1.7014018E+01       7             1 1.8E+00 9.2E-02 _
      8     5      1 1.0E+00      6 (3.3E-07) 2.3E-05  1.7014017E+01       7             1 1.9E+00 3.6E-01 _
      9     6      1 1.0E+00      7 (4.2E-10)(2.4E-08) 1.7014017E+01       7             1 1.9E+00 3.6E-01 _

 E04WDF EXIT   0 -- finished successfully
 E04WDF INFO   1 -- optimality conditions satisfied

 Problem name                     NLP
 No. of iterations                  9   Objective value      1.7014017287E+01
 No. of major iterations            6   Linear objective     0.0000000000E+00
 Penalty parameter          3.599E-01   Nonlinear objective  1.7014017287E+01
 No. of calls to funobj             8   No. of calls to funcon              8
 No. of superbasics                 1   No. of basic nonlinears             2
 No. of degenerate steps            0   Percentage                       0.00
 Max x                      2 4.7E+00   Max pi                      2 5.5E-01
 Max Primal infeas          0 0.0E+00   Max Dual infeas             3 4.8E-08
 Nonlinear constraint violn    2.7E-09


 Variable         State      Value       Lower bound     Upper bound   Lagr multiplier       Slack

 variable     1     LL    1.000000        1.000000        5.000000        1.087871             .
 variable     2     FR    4.743000        1.000000        5.000000            .              0.2570
 variable     3     FR    3.821150        1.000000        5.000000            .               1.179
 variable     4     FR    1.379408        1.000000        5.000000            .              0.3794


 Linear constrnt  State      Value       Lower bound     Upper bound   Lagr multiplier       Slack

 lincon       1     FR    10.94356          None         20.00000            .               9.056


 Nonlin constrnt  State      Value       Lower bound     Upper bound   Lagr multiplier       Slack

 nlncon       1     UL    40.00000          None         40.00000      -0.1614686        -0.2700E-08
 nlncon       2     LL    25.00000        25.00000          None        0.5522937        -0.2215E-08


Final objective value =       17.014
Optimal X =      1.00      4.74      3.82      1.38
```

Note: *the remainder of this document is intended for more advanced users. Section 10 contains a detailed description of the algorithm which may be needed in order to understand Sections 11 and 12. Section 11 describes the optional arguments which may be set by calls to nag_opt_nlp_option_set_string (e04wfc),* *nag_opt_nlp_option_set_integer (e04wgc) and/or nag_opt_nlp_option_set_double (e04whc). Section 12 describes the quantities which can be requested to monitor the course of the computation.*

10 Algorithmic Details

Here we summarize the main features of the SQP algorithm used in nag_opt_nlp_solve (e04wdc) and introduce some terminology used in the description of the function and its arguments. The SQP algorithm is fully described in Gill *et al.* (2002).

10.1 Constraints and Slack Variables

The upper and lower bounds on the m components of $c(x)$ and A_Lx are said to define the *general constraints* of the problem. nag_opt_nlp_solve (e04wdc) converts the general constraints to equalities by introducing a set of *slack variables* $s = (s_1, s_2, \ldots, s_m)^{\mathrm{T}}$. For example, the linear constraint $5 \leq 3x_1 + 3x_2 \leq +\infty$ is replaced by $2x_1 + 3x_2 - s_1 = 0$ together with the bounded slack $5 \leq s_1 \leq +\infty$. The minimization problem (1) can therefore be written in the equivalent form

$$\underset{x,s}{\text{minimize}}\, F(x) \quad \text{subject to} \begin{pmatrix} c(x) \\ A_Lx \end{pmatrix} - s = 0, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u. \tag{2}$$

The general constraints become the equalities $c(x) - s_N = 0$ and $A_Lx - s_L = 0$, where s_L and s_N are known as the *linear* and *nonlinear* slacks.

10.2 Major Iterations

The basic structure of the SQP algorithm 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 that satisfies the first-order conditions for optimality. 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 $A_Lx - s_L = 0$ and the nonlinear constraint linearization

$$c(x_k) + c'(x_k)(x - x_k) - s_N = 0, \tag{3}$$

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

$$\begin{array}{lll} c'(x_k)x & -s_N & = -c(x_k) + c'(x_k)x_k, \\ A_Lx & -s_L & = 0, \end{array} \tag{4}$$

where x and s are bounded above and below by u and l as before. If the m by n matrix A and m-vector b are defined as

$$A = \begin{pmatrix} c'(x_k) \\ A_L \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} -c(x_k) + c'(x_k)x_k \\ 0 \end{pmatrix}, \tag{5}$$

then the QP subproblem can be written as

$$\underset{x,s}{\text{minimize}}\, q(x) \quad \text{subject to } Ax - s = b, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u, \tag{6}$$

where $q(x)$ is a quadratic approximation to a modified Lagrangian function (see Gill *et al.* (2002)).

10.3 Minor Iterations

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

$$Bx_B + Sx_S + Nx_N = b, \tag{7}$$

where the *basic matrix B* is square and nonsingular. 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 subproblem, the basic and superbasic variables will lie somewhere between their bounds, while the nonbasic variables will normally be equal to one of their bounds. At each 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 (or the sum of infeasibilities). 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 for LP problems.

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 is decreased by one.

Associated with each of the m equality constraints $Ax - s = b$ are the *dual variables* π. Similarly, each variable in (x, s) has an associated *reduced gradient* d_j. The reduced gradients for the variables x are the quantities $g - A^{\mathrm{T}}\pi$, where g is the gradient of the QP objective, and the reduced gradients for the slacks are the dual variables π. The QP subproblem 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 $(\hat{x}_k, \hat{s}_k, \hat{\pi}_k)$ is found by relaxing these conditions.

10.4 The Merit Function

After a QP subproblem has been solved, new estimates of the solution are computed using a line search on the augmented Lagrangian merit function

$$\mathcal{M}(x, s, \pi) = F(x) - \pi^{\mathrm{T}}(c(x) - s_N) + \frac{1}{2}(c(x) - s_N)^{\mathrm{T}} D(c(x) - s_N), \tag{8}$$

where D is a diagonal matrix of penalty arguments $(D_{ii} \geq 0)$. If (x_k, s_k, π_k) denotes the current solution estimate and $(\hat{x}_k, \hat{s}_k, \hat{\pi}_k)$ denotes the QP solution, the line search determines a step α_k $(0 < \alpha_k \leq 1)$ such that the new point

$$\begin{pmatrix} x_{k+1} \\ s_{k+1} \\ \pi_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ s_k \\ \pi_k \end{pmatrix} + \alpha_k \begin{pmatrix} \hat{x}_k - x_k \\ \hat{s}_k - s_k \\ \hat{\pi}_k - \pi_k \end{pmatrix} \tag{9}$$

gives a *sufficient decrease* in the merit function (see (8)). When necessary, the penalties in D are increased by the minimum-norm perturbation that ensures descent for $\mathcal{M}$ (see Gill *et al.* (1992)). s_N is adjusted to minimize the merit function as a function of s prior to the solution of the QP subproblem (see Gill *et al.* (1986c) and Eldersveld (1991)).

10.5 Treatment of Constraint Infeasibilities

nag_opt_nlp_solve (e04wdc) makes explicit allowance for infeasible constraints. First, infeasible *linear* constraints are detected by solving the linear program

$$\underset{x,v,w}{\text{minimize}}\, e^{\mathrm{T}}(v + w) \quad \text{subject to } l \leq \begin{pmatrix} x \\ A_L x - v + w \end{pmatrix} \leq u, \quad u \geq 0, \quad w \geq 0, \tag{10}$$

where e is a vector of ones, and the nonlinear constraint bounds are temporarily excluded from l and u. This is equivalent to minimizing the sum of the general linear constraint violations subject to the bounds on x. (The sum is the ℓ_1-norm of the linear constraint violations. In the linear programming literature, the approach is called *elastic programming*.)

The linear constraints are infeasible if the optimal solution of (10) has $v \neq 0$ or $w \neq 0$. nag_opt_nlp_solve (e04wdc) then terminates without computing the nonlinear functions.

Otherwise, all subsequent iterates satisfy the linear constraints. (Such a strategy allows linear constraints to be used to define a region in which the functions can be safely evaluated.) nag_opt_nlp_solve (e04wdc) proceeds to solve nonlinear problems as given, using search directions obtained from the sequence of QP subproblems (see (6)).

If a QP subproblem proves to be infeasible or unbounded (or if the dual variable π for the nonlinear constraints become large), nag_opt_nlp_solve (e04wdc) enters 'elastic' mode and thereafter solves the problem

$$\underset{x,v,w}{\text{minimize}}\, F(x) + \gamma e^{\mathrm{T}}(v+w) \quad \text{subject to } l \leq \begin{pmatrix} x \\ c(x) - v + w \\ A_L x \end{pmatrix} \leq u, \quad v \geq 0, \quad w \geq 0, \tag{11}$$

where γ is a non-negative argument (the *elastic weight*), and $F(x) + \gamma e^{\mathrm{T}}(v+w)$ is called a *composite objective* (the ℓ_1 penalty function for the nonlinear constraints).

The value of γ may increase automatically by multiples of 10 if the optimal u and w continue to be non-zero. If γ is sufficiently large, this is equivalent to minimizing the sum of the nonlinear constraint violations subject to the linear constraints and bounds.

The initial value of γ is controlled by the optional arguments **Elastic Mode** and **Elastic Weight** (see Section 11.2).

11 Optional Arguments

Several optional arguments in nag_opt_nlp_solve (e04wdc) define choices in the problem specification or the algorithm logic. In order to reduce the number of formal arguments of nag_opt_nlp_solve (e04wdc) these optional arguments have associated *default values* that are appropriate for most problems. Therefore, you need only specify those optional arguments whose values are to be different from their default values.

The remainder of this section can be skipped if you wish to use the default values for all optional arguments. A complete list of optional arguments and their default values is given in Section 11.1.

Optional arguments may be specified by calling one, or more, of the functions nag_opt_nlp_option_set_file (e04wec), nag_opt_nlp_option_set_string (e04wfc) and nag_opt_nlp_option_set_integer (e04wgc) prior to a call to nag_opt_nlp_solve (e04wdc).

nag_opt_nlp_option_set_file (e04wec) reads options from an external options file, with `Begin` and `End` as the first and last lines respectively and each intermediate line defining a single optional argument. For example,

```
Begin
   Print Level = 5
End
```

The call

```
e04wec (ioptns, &state, &fail);
```

can then be used to read the file on the descriptor `ioptns` as returned by a call of nag_open_file (x04acc). **fail.code** = **NE_NOERROR** on successful exit. nag_opt_nlp_option_set_file (e04wec) should be consulted for a full description of this method of supplying optional arguments.

nag_opt_nlp_option_set_string (e04wfc), nag_opt_nlp_option_set_integer (e04wgc) or nag_opt_nlp_option_set_double (e04whc) can be called to supply options directly, one call being necessary for each optional argument. nag_opt_nlp_option_set_string (e04wfc), nag_opt_nlp_option_set_integer (e04wgc) or nag_opt_nlp_option_set_double (e04whc) should be consulted for a full description of this method of supplying optional arguments.

All optional arguments not specified by you are set to their default values. Optional arguments specified by you are unaltered by nag_opt_nlp_solve (e04wdc) (unless they define invalid values) and so remain in effect for subsequent calls to nag_opt_nlp_solve (e04wdc), unless altered by you.

11.1 Optional Argument Checklist and Default Values

The following list gives the valid options. For each option, we give the keyword, any essential optional qualifiers and the default value. A definition for each option can be found in Section 11.2. The minimum abbreviation of each keyword is underlined. If no characters of an optional qualifier are underlined, the qualifier may be omitted. The letter a denotes a phrase (character string) that qualifies an option. The letters i and r denote Integer and double values required with certain options. The number ϵ is a generic notation for ***machine precision*** (see nag_machine_precision (X02AJC)), and ϵ_R denotes the relative precision of the objective function (see **Function Precision**).

Optional arguments used to specify files (e.g., **Dump File** and **Print File**) have type **Nag_FileID**. This ID value must either be set to 0 (the default value) in which case there will be no output, or will be as returned by a call of nag_open_file (x04acc).

Optional Arguments	**Default Values**
Backup Basis File	Default $= 0$
Central Difference Interval	Default $= \epsilon^{4/15}$
Check Frequency	Default $= 60$
Cold Start	Default $=$ **Cold Start**
Crash Option	Default $= 3$
Crash Tolerance	Default $= 0.1$
Derivative Level	Default $= 3$
Defaults	
Derivative Linesearch	Default
Difference Interval	Default $= \epsilon^{0.4}$
Dump File	Default $= 0$
Elastic Mode	Default $=$ No
Elastic Weight	Default $= 10^4$
Expand Frequency	Default $= 10000$
Factorization Frequency	Default $= 50$
Feasibility Tolerance	Default $= 1.0\mathrm{e}-6$
Feasible Point	
Function Precision	Default $= \epsilon^{0.8}$
Hessian Full Memory	Default $=$ Full if $n_1 \leq 75$
Hessian Limited Memory	
Hessian Frequency	Default $= 99999999$
Hessian Updates	Default $= 99999999$
Insert File	Default $= 0$
Infinite Bound Size	Default $= 10^{20}$
Linesearch Tolerance	Default $= 0.9$
List	Default $=$ **Nolist**
Load File	Default $= 0$
LU Complete Pivoting	
LU Density Tolerance	Default $= 0.6$
LU Factor Tolerance	Default $= 1.01$
LU Partial Pivoting	Default
LU Rook Pivoting	
LU Singularity Tolerance	Default $= \sqrt{\epsilon}$
LU Update Tolerance	Default $= 1.01$
Major Feasibility Tolerance	Default $= 1.0\mathrm{e}-6$
Major Iterations Limit	Default $= \max\{1000, m\}$
Major Optimality Tolerance	Default $= 2.0\mathrm{e}-6$
Major Print Level	Default $= 00001$
Major Step Limit	Default $= 2.0$
Maximize	
Minimize	Default
Minor Feasibility Tolerance	Default $= 1.0\mathrm{e}-6$
Minor Iterations Limit	Default $= 500$
Minor Print Level	Default $= 1$

New Basis File	Default $= 0$
New Superbasics Limit	Default $= 99$
Nolist	
Nonderivative Linesearch	
Old Basis File	Default $= 0$
Partial Price	Default $= 1$
Pivot Tolerance	Default $= 10 \times \epsilon$
Print File	Default $= 0$
Print Frequency	Default $= 100$
Proximal Point Method	Default $= 1$
Punch File	Default $= 0$
Save Frequency	Default $= 100$
Scale Option	Default $= 0$
Scale Tolerance	Default $= 0.9$
Solution File	Default $= 0$
Start Constraint Check At Variable	Default $= 1$
Start Objective Check At Variable	Default $= 1$
Stop Constraint Check At Variable	Default $= n$
Stop Objective Check At Variable	Default $= n$
Summary File	Default $= 0$
Summary Frequency	Default $= 100$
Superbasics Limit	Default $= \min(500, n_1 + 1)$
Suppress Parameters	
Timing Level	Default $= 0$
Unbounded Objective	Default $= 1.0\mathrm{e}+15$
Unbounded Step Size	Default $= 1.0\mathrm{e}+20$
Verify Level	Default $= 0$
Violation Limit	Default $= 1.0\mathrm{e}+6$
Warm Start	

11.2 Description of the Optional Arguments

Central Difference Interval – double $\quad r \quad$ Default $= \epsilon^{4/15}$

When **Derivative Level** < 3, the central-difference interval r is used near an optimal solution to obtain more accurate (but more expensive) estimates of gradients. Twice as many function evaluations are required compared to forward differencing. The interval used for the jth variable $h_j = r(1 + |x_j|)$. The resulting derivative estimates should be accurate to $O(r^2)$, unless the functions are badly scaled.

Check Frequency – Integer $\quad i \quad$ Default $= 60$

Every ith minor iteration after the most recent basis factorization, a numerical test is made to see if the current solution x satisfies the general linear constraints (the linear constraints and the linearized nonlinear constraints, if any). The constraints are of the form $Ax - s = b$, where s is the set of slack variables. To perform the numerical test, the residual vector $r = b - Ax + s$ is computed. If the largest component of r is judged to be too large, the current basis is refactorized and the basic variables are recomputed to satisfy the general constraints more accurately.

Check Frequency $= 1$ is useful for debugging purposes, but otherwise this option should not be needed.

Cold Start Default $=$ **Cold Start**
Warm Start

This option controls the specification of the initial working set in both the procedure for finding a feasible point for the linear constraints and bounds and in the first QP subproblem thereafter. With a **Cold Start**, the first working set is chosen by nag_opt_nlp_solve (e04wdc) based on the values of the variables and constraints at the initial point. Broadly speaking, the initial working set will include equality constraints and bounds or inequality constraints that violate or 'nearly' satisfy their bounds (to within **Crash Tolerance**; see below).

With a **Warm Start**, you must set the **istate** array and define **clamda** and **hess** as discussed in Section 5. **istate** values associated with bounds and linear constraints determine the initial working set of the procedure to find a feasible point with respect to the bounds and linear constraints. **istate** values associated with nonlinear constraints determine the initial working set of the first QP subproblem after such a feasible point has been found. nag_opt_nlp_solve (e04wdc) will override your specification of **istate** if necessary, so that a poor choice of the working set will not cause a fatal error. For instance, any elements of **istate** which are set to -2, -1 or 4 will be reset to zero, as will any elements which are set to 3 when the corresponding elements of **bl** and **bu** are not equal. A warm start will be advantageous if a good estimate of the initial working set is available – for example, when nag_opt_nlp_solve (e04wdc) is called repeatedly to solve related problems.

Crash Option – Integer i Default $= 3$
Crash Tolerance – double r Default $= 0.1$

If optional **Cold Start**, an internal Crash procedure is used to select an initial basis from certain rows and columns of the constraint matrix $(A \quad -I)$. The **Crash Option** i determines which rows and columns of A are eligible initially, and how many times the Crash procedure is called. Columns of $-I$ are used to pad the basis where necessary.

i	**Meaning**
0	The initial basis contains only slack variables: $B = I$.
1	The Crash procedure is called once, looking for a triangular basis in all rows and columns of the matrix A.
2	The Crash procedure is called twice (if there are 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 (retaining the current basis for linear rows).
3	The Crash procedure is called up to three times (if there are nonlinear constraints). The first two calls treat *linear equalities* and *linear inequalities* separately. As before, the last call treats nonlinear rows before the first major iteration.

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

The **Crash Tolerance** r allows the starting Crash procedure to ignore certain 'small' non-zeros in each column of A. If $a_{\max}$ is the largest element in column j, other non-zeros of a_{ij} in the columns are ignored if $|a_{ij}| \leq a_{\max} \times r$. (To be meaningful, r should be in the range $0 \leq r < 1$.)

When $r > 0.0$, the basis obtained by the Crash procedure may not be strictly triangular, but it is likely to be nonsingular 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 sooner on some problems.

Defaults

This special keyword may be used to reset all optional arguments to their default values.

Derivative Level – Integer i Default $= 3$

Derivative Level specifies which nonlinear function gradients are known analytically and will be supplied to nag_opt_nlp_solve (e04wdc) by your functions **objfun** and **confun**.

i	**Meaning**
3	All objective and constraint gradients are known.
2	All constraint gradients are known, but some or all components of the objective gradient are unknown.
1	The objective gradient is known, but some or all of the constraint gradients are unknown.

0 Some components of the objective gradient are unknown and some of the constraint gradients are unknown.

The value $i = 3$ should be used whenever possible. It is the most reliable and will usually be the most efficient.

If $i = 0$ or 2, nag_opt_nlp_solve (e04wdc) will *estimate* the missing components of the objective gradient, using finite differences. This may simplify the coding of function **objfun**. However, it could increase the total run-time substantially (since a special call to **objfun** is required for each missing element), and there is less assurance that an acceptable solution will be located. If the nonlinear variables are not well scaled, it may be necessary to specify a nonstandard **Difference Interval**.

If $i = 0$ or 1, nag_opt_nlp_solve (e04wdc) will estimate missing elements of the Jacobian. For each column of the Jacobian, one call to **confun** is needed to estimate all missing elements in that column, if any. If the sparsity pattern of the Jacobian happens to be

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

where $*$ indicates known gradients and ? indicates unknown elements, nag_opt_nlp_solve (e04wdc) will use one call to **confun** to estimate the missing element in column 2, and another call to estimate both missing elements in column 3. No calls are needed for columns 1 and 4.

At times, central differences are used rather than forward differences. Twice as many calls to **objfun** and **confun** are needed. (This is not under your control.)

Derivative Linesearch Default
Nonderivative Linesearch

At each major iteration a line search is used to improve the merit function. A **Derivative Linesearch** uses safeguarded cubic interpolation and requires both function and gradient values to compute estimates of the step α_k. If some analytic derivatives are not provided, or a **Nonderivative Linesearch** is specified, nag_opt_nlp_solve (e04wdc) employs a line search based upon safeguarded quadratic interpolation, which does not require gradient evaluations.

A nonderivative line search can be slightly less robust on difficult problems, and it is recommended that the default be used if the functions and derivatives can be computed at approximately the same cost. If the gradients are very expensive relative to the functions, a nonderivative line search may give a significant decrease in computation time.

Difference Interval – double r Default $= \epsilon^{0.4}$

This alters the interval r that is used to estimate gradients by forward differences in the following circumstances:

– in the interval ('cheap') phase of verifying the problem derivatives;

– for verifying the problem derivatives;

– for estimating missing derivatives.

In all cases, a derivative with respect to x_j is estimated by perturbing that component of x to the value $x_j + r(1 + |x_j|)$, and then evaluating $F(x)$ or $c(x)$ at the perturbed point. The resulting gradient estimates should be accurate to $O(r)$ unless the functions are badly scaled. Judicious alteration of r may sometimes lead to greater accuracy.

Dump File – Nag_FileID i_1 Default $= 0$
Load File – Nag_FileID i_2 Default $= 0$

(See Section 11.1 for a description of **Nag_FileID**.)

Dump File and **Load File** are similar to **Punch File** and **Insert File**, but they record solution information in a manner that is more direct and more easily modified. A full description of information recorded in **Dump File** and **Load File** is given in Gill *et al.* (1999).

If **Dump File** > 0, the last solution obtained will be output to the file associated with ID i_1.

If **Load File** > 0, the file associated with ID i_2, containing basis information, will be read. The file will usually have been output previously as a **Dump File**. The file will not be accessed if an **Old Basis File** or an **Insert File** is specified.

Elastic Mode Default $=$ No

Normally nag_opt_nlp_solve (e04wdc) initiates elastic mode only when it seems necessary. Setting **Elastic Mode** $=$ Yes causes elastic mode to be entered from the beginning.

Elastic Weight – double r Default $= 10^4$

This keyword determines the initial weight γ associated with the problem (11) (see Section 10.5).

At major iteration k, if elastic mode has not yet started, a scale factor $\sigma_k = 1 + \|g(x_k)\|_\infty$ is defined from the current objective gradient. Elastic mode is then started if the QP subproblem is infeasible, or the QP dual variables are larger in magnitude than $\sigma_k r$. The QP is resolved in elastic mode with $\gamma = \sigma_k r$.

Thereafter, major iterations continue in elastic mode until they converge to a point that is optimal for (11) (see Section 10.5). If the point is feasible for $(v = w = 0)$, it is declared locally optimal. Otherwise, γ is increased by a factor of 10 and major iterations continue.

Expand Frequency – Integer i Default $= 10000$

This option is part of the anti-cycling procedure 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 bounds on the variables by a small amount. Suppose that the **Minor Feasibility Tolerance** is δ. Over a period of i iterations, the tolerance actually used by nag_opt_nlp_solve (e04wdc) increases from 0.5δ to δ (in steps of $0.5\delta/i$).

For nonlinear models, the same procedure is used for iterations in which there is only one superbasic variable. (Cycling can occur only 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 i helps reduce the number of slightly infeasible nonbasic basic variables (most of which are eliminated during a resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see **Pivot Tolerance**).

Factorization Frequency – Integer i Default $= 50$

At most i basis changes will occur between factorizations of the basis matrix.

With linear programs, the basis factors are usually updated every iteration. The default i is reasonable for typical problems. Higher values up to $i = 100$ (say) may be more efficient on well scaled problems.

When the objective function is nonlinear, fewer basis updates will occur as an optimum is approached. The number of iterations between basis factorizations will therefore increase. During these iterations a test is made regularly (according to the **Check Frequency**) to ensure that the general constraints are satisfied. If necessary the basis will be refactorized before the limit of i updates is reached.

Feasibility Tolerance – Integer i Default $= 1.0\text{e}-6$

See **Minor Feasiblity Tolerance**.

Function Precision – double r Default $= \epsilon^{0.8}$

The *relative function precision* ϵ_r is intended to be a measure of the relative accuracy with which the functions can be computed. For example, if $F(x)$ is computed as 1000.56789 for some relevant x and if the first 6 significant digits are known to be correct, the appropriate value for ϵ_r would be $1.0\mathrm{e}-6$.

(Ideally the functions $F(x)$ or $c_i(x)$ 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 $F(x) = 1.23456789\mathrm{e}-4$ at some point and if the first 6 significant digits are known to be correct, the appropriate value for ϵ_r would be $1.0\mathrm{e}-10$.)

The default value of ϵ_r is appropriate for simple analytic functions.

In some cases the function values will be the result of extensive computation, possibly involving a costly iterative procedure that can provide few digits of precision. Specifying an appropriate **Function Precision** may lead to savings, by allowing the line search procedure to terminate when the difference between function values along the search direction becomes as small as the absolute error in the values.

Hessian Full Memory Default = Full if $n_1 \leq 75$
Hessian Limited Memory

These options select the method for storing and updating the approximate Hessian. (nag_opt_nlp_solve (e04wdc) uses a quasi-Newton approximation to the Hessian of the Lagrangian. A BFGS update is applied after each major iteration.)

If **Hessian Full Memory** is specified, the approximate Hessian is treated as a dense matrix and the BFGS updates are applied explicitly. This option is most efficient when the number of nonlinear variables n_1 is not too large (say, less than 75). In this case, the storage requirement is fixed and one can expect 1-step Q-superlinear convergence to the solution.

Hessian Limited Memory should be used on problems where n_1 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.)

Hessian Frequency – Integer i Default $= 99999999$

If **Hessian Full Memory** is selected and i BFGS updates have already been carried out, the Hessian approximation is reset to the identity matrix. (For certain problems, occasional resets may improve convergence, but in general they should not be necessary.)

Hessian Full Memory and **Hessian Frequency** $= 20$ have a similar effect to **Hessian Limited Memory** and **Hessian Updates** $= 20$ (except that the latter retains the current diagonal during resets).

Hessian Updates – Integer i Default $= 99999999$

If **Hessian Limited Memory** is selected and i BFGS updates have already been carried out, all but the diagonal elements of the accumulated updates are discarded and the updating process starts again.

Broadly speaking, the more updates stored, the better the quality of the approximate Hessian. However, the more vectors stored, the greater the cost of each QP iteration. The default value is likely to give a robust algorithm without significant expense, but faster convergence can sometimes be obtained with significantly fewer updates (e.g., $i = 5$).

Infinite Bound Size – double r Default $= 10^{20}$

If $r > 0$, r defines the 'infinite' bound $bigbnd$ in the definition of the problem constraints. Any upper bound greater than or equal to $bigbnd$ will be regarded as plus infinity (and similarly any lower bound less than or equal to $-bigbnd$ will be regarded as minus infinity). If $r \leq 0$, the default value is used.

Linesearch Tolerance – double r Default $= 0.9$

This tolerance, r, controls the accuracy with which a steplength will be located along the direction of each search iteration. At the start of each line search a target directional derivative for the merit function is identified. This argument determines the accuracy to which this target value is approximated.

r must be a double value in the range $0.0 \leq r \leq 1.0$.

The default value $r = 0.9$ requests just moderate accuracy in the line search.

If the nonlinear functions are cheap to evaluate, a more accurate search may be appropriate; try $r = 0.1$, 0.01 or 0.001.

If the nonlinear functions are expensive to evaluate, a less accurate search may be appropriate. *If all gradients are known*, try $r = 0.99$. (The number of major iterations might increase, but the total number of function evaluations may decrease enough to compensate.)

If not all gradients are known, a moderately accurate search remains appropriate. Each search will require only 1–5 function values (typically), but many function calls will then be needed to estimate missing gradients for the next iteration.

List Default = **Nolist**
Nolist

For nag_opt_nlp_solve (e04wdc), normally each optional argument specification is printed as it is supplied. **Nolist** may be used to suppress the printing and **List** may be used to turn on printing.

LU Factor Tolerance – double r_1 Default $= 1.01$
LU Update Tolerance – double r_2 Default $= 1.01$

The values of r_1 and r_2 affect the stability of the basis factorization $B = LU$, during refactorization and updates respectively. The lower triangular matrix L is a product of matrices of the form

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

where the multipliers μ will satisfy $|\mu| \leq r_i$. The default values of r_1 and r_2 usually strike a good compromise between stability and sparsity. They must satisfy r_1, $r_2 \geq 1.0$.

For large and relatively dense problems, $r_1 = 10.0$ or 5.0 (say) may give a useful improvement in stability without impairing sparsity to a serious degree.

LU Partial Pivoting Default
LU Rook Pivoting
LU Complete Pivoting

The LU factorization implements a Markowitz-type search for a pivot that locally minimizes the fill-in subject to a threshhold pivoting stability criterion. The default option is to use threshhold partial pivoting. The options **LU Rook Pivoting** and **LU Complete Pivoting** are more expensive than partial pivoting but are more stable and better at revealing rank.

LU Density Tolerance – double r_1 Default $= 0.6$
LU Singularity Tolerance – double r_2 Default $= \sqrt{\epsilon}$

The density tolerance, r_1, is used during 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 r_1, the Markowitz strategy for choosing pivots is terminated. The remaining matrix is factored by a dense LU procedure. Raising the density tolerance towards 1.0 may give slightly sparser LU factors, with a slight increase in factorization time.

The singularity tolerance, r_2, helps guard against ill-conditioned basis matrices. When the basis is refactorized, the diagonal elements of U are tested as follows: if $|U_{jj}| \leq r_2$ or $|U_{jj}| < r_2 \max_i |U_{ij}|$, the jth column of the basis is replaced by the corresponding slack variable. (This is most likely to occur after a restart, or at the start of a major iteration.)

In some cases, the Jacobian matrix may converge to values that make the basis exactly singular. (For example, a whole row of the Jacobian 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 a larger tolerance $r_2 = 1.0\mathrm{e}-5$, say, may help cause a judicious change of basis.

Major Feasibility Tolerance – double r Default $= 1.0\mathrm{e}-6$

This tolerance, r, specifies how accurately the nonlinear constraints should be satisfied. The default value is appropriate when the linear and nonlinear constraints contain data to about that accuracy.

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

$$rowerr = \max_i viol_i/\|x\| \leq r, \tag{12}$$

where $viol_i$ is the violation of the ith nonlinear constraint ($i = 1 : \mathbf{nclin}$).

In the major iteration log (see Section , *rowerr* appears as the quantity labelled 'Feasible'. If some of the problem functions are known to be of low accuracy, a larger **Major Feasibility Tolerance** may be appropriate.

Major Optimality Tolerance – double r Default $= 2.0\mathrm{e}-6$

This tolerance, r, specifies the final accuracy of the dual variables. On successful termination, nag_opt_nlp_solve (e04wdc) will have computed a solution (x, s, π) such that

$$maxComp = \max_j Comp_j/\|\pi\| \leq r, \tag{13}$$

where $Comp_j$ is an estimate of the complementarity slackness for variable j ($j = 1, 2, \ldots, n + m$). The values $Comp_j$ are computed from the final QP solution using the reduced gradients $d_j = g_j - \pi^{\mathrm{T}} a_j$ (where g_j is the jth component of the objective gradient, a_j is the associated column of the constraint matrix $(A \quad -I)$, and π is the set of QP dual variables):

$$Comp_j = \left\{ \begin{array}{ll} d_j \min\left\{x_j - l_j, 1\right\} & \text{if } d_j \geq 0; \\ -d_j \min\left\{u_j - x_j, 1\right\} & \text{if } d_j < 0. \end{array} \right) \tag{14}$$

In the **Print File**, *maxComp* appears as the quantity labelled 'Optimal'.

Major Iterations Limit – Integer i Default $= \max\{1000, m\}$

This is the maximum number of major iterations allowed. It is intended to guard against an excessive number of linearizations of the constraints.

Major Print Level – Integer i Default $= 00001$

This controls the amount of output to the **Print File** and **Summary File** at each major iteration. **Major Print Level** $= 0$ suppresses most output, except for error messages. **Major Print Level** $= 1$ gives normal output for linear and nonlinear problems, and **Major Print Level** $= 11$ gives additional details of the Jacobian factorization that commences each major iteration.

In general, the value being specified may be thought of as a binary number of the form

Major Print Level *JFDXbs*

where each letter stands for a digit that is either 0 or 1 as follows:

s a single line that gives a summary of each major iteration. (This entry in *JFDXbs* is not strictly binary since the summary line is printed whenever *JFDXbs* ≥ 1);

b basis statistics, i.e., information relating to the basis matrix whenever it is refactorized. (This output is always provided if *JFDXbs* ≥ 10);

X x_k, the nonlinear variables involved in the objective function or the constraints;

D π_k, the dual variables for the nonlinear constraints;

F $F(x_k)$, the values of the nonlinear constraint functions;

J $J(x_k)$, the Jacobian matrix.

To obtain output of any items *JFDXbs*, set the corresponding digit to 1, otherwise to 0.

If $J = 1$, the Jacobian matrix will be output column-wise at the start of each major iteration. Column j will be preceded by the value of the corresponding variable x_j and a key to indicate whether the variable is basic, superbasic or nonbasic. (Hence if $J = 1$, there is no reason to specify $X = 1$ unless the objective contains more nonlinear variables than the Jacobian.) A typical line of output is

```
3 1.250000e+01 BS 1 1.00000e+00 4 2.00000e+00
```

which would mean that x_3 is basic at value 12.5, and the third column of the Jacobian has elements of 1.0 and 2.0 in rows 1 and 4.

Major Step Limit – double r Default $= 2.0$

This argument limits the change in x during a line search. It applies to all nonlinear problems, once a 'feasible solution' or 'feasible subproblem' has been found.

1. A line search determines a step α over the range $0 < \alpha \leq \beta$, where β is 1 if there are nonlinear constraints, or the step to the nearest upper or lower bound on x if all the constraints are linear. Normally, the first steplength tried is $\alpha_1 = \min(1, \beta)$.

2. In some cases, such as $f(x) = ae^{bx}$ or $f(x) = ax^b$, even a moderate change in the components of x can lead to floating-point overflow. The argument r is therefore used to define a limit $\bar{\beta} = r(1 + \|x\|)/\|p\|$ (where p is the search direction), and the first evaluation of $f(x)$ is at the potentially smaller steplength $\alpha_1 = \min\left(1, \bar{\beta}, \beta\right)$.

3. Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at meaningless points. The **Major Step Limit** provides an additional safeguard. The default value $r = 2.0$ should not affect progress on well behaved problems, but setting $r = 0.1$ or 0.01 may be helpful when rapidly varying functions are present. A 'good' starting point may be required. An important application is to the class of nonlinear least-squares problems.

4. In cases where several local optima exist, specifying a small value for r may help locate an optimum near the starting point.

Minimize Default
Maximize
Feasible Point

The keywords **Minimize** and **Maximize** specify the required direction of optimization. It applies to both linear and nonlinear terms in the objective.

The keyword **Feasible Point** means 'Ignore the objective function' while finding a feasible point for the linear and nonlinear constraints. It can be used to check that the nonlinear constraints are feasible without altering the call to nag_opt_nlp_solve (e04wdc).

Minor Feasibility Tolerance – double r Default $= 1.0\text{e}-6$

nag_opt_nlp_solve (e04wdc) tries to ensure that all variables eventually satisfy their upper and lower bounds to within this tolerance, r. This includes slack variables. Hence, general linear constraints should also be satisfied to within r.

Feasibility with respect to nonlinear constraints is judged by the **Major Feasibility Tolerance** (not by r).

If the bounds and linear constraints cannot be satisfied to within r, the problem is declared *infeasible*. Let `sInf` be the corresponding sum of infeasibilities. If `sInf` is quite small, it may be appropriate to raise r by a factor of 10 or 100. Otherwise, some error in the data should be suspected.

Nonlinear functions will be evaluated only at points that satisfy the bounds and linear constraints. If there are regions where a function is undefined, every attempt should be made to eliminate these regions from the problem.

For example, if $f(x) = \sqrt{x_1} + \log(x_2)$, it is essential to place lower bounds on both variables. If $r = 1.0\text{e}-6$, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious. In general, keep x as far away from singularities as possible.)

If **Scale Option** ≥ 1, feasibility is defined in terms of the *scaled* problem (since it is then more likely to be meaningful).

In reality, nag_opt_nlp_solve (e04wdc) uses r 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. nag_opt_nlp_solve (e04wdc) is then in *elastic mode* thereafter (with only the linearized nonlinear constraints defined to be elastic). See the **Elastic Mode** option.

Minor Iterations Limit – Integer i Default $= 500$

If the number of minor iterations for the optimality phase of the QP subproblem exceeds i, then all nonbasic QP variables that have not yet moved are frozen at their current values and the reduced QP is solved to optimality.

Note that more than i minor iterations may be necessary to solve the reduced QP to optimality. These extra iterations are necessary to ensure that the terminated point gives a suitable direction for the line search.

In the major iteration log (see Section) a 't' at the end of a line indicates that the corresponding QP was artificially terminated using the limit i.

Note that **Minor Iterations Limit** defines an independent *absolute* limit on the *total* number of minor iterations (summed over all QP subproblems).

Minor Print Level – Integer i Default $= 1$

This controls the amount of output to the **Print File** and **Summary File** during solution of the QP subproblems. The value of i has the following effect:

i	Output
0	No minor iteration output except error messages.
≥ 1	A single line of output at each minor iteration (controlled by **Print Frequency** and **Summary Frequency**.
≥ 10	Basis factorization statistics generated during the periodic refactorization of the basis (see **Factorization Frequency**). Statistics for the *first factorization* each major iteration are controlled by the **Major Print Level**.

New Basis File – Nag_FileID i_1 Default $= 0$
Backup Basis File – Nag_FileID i_2 Default $= 0$
Save Frequency – Integer i_3 Default $= 100$

(See Section 11.1 for a description of **Nag_FileID**.)

New Basis File and **Backup Basis File** are sometimes referred to as basis maps. They contain the most compact representation of the state of each variable. They are intended for restarting the solution of a problem at a point that was reached by an earlier run. For non-trivial problems, it is advisable to save basis maps at the end of a run, in order to restart the run if necessary.

If **New Basis File** > 0, a basis map will be saved on the **New Basis File** every i_3th iteration. The first record of the file will contain the word PROCEEDING if the run is still in progress. A basis map will also be saved at the end of a run, with some other word indicating the final solution status.

If **Backup Basis File** > 0, it is intended as a safeguard against losing the results of a long run. Suppose that a **New Basis File** is being saved every 100 (**Save Frequency**) iterations, and that nag_opt_nlp_solve (e04wdc) is about to save such a basis at iteration 2000. It is conceivable that the run may be interrupted during the next few milliseconds (in the middle of the save). In this case the basis file will be corrupted and the run will have been essentially wasted.

To eliminate this risk, both a **New Basis File** and a **Backup Basis File** may be specified using calls of nag_open_file (x04acc).

The current basis will then be saved every 100 iterations, first on the **New Basis File** and then immediately on the **Backup Basis File**. If the run is interrupted at iteration 2000 during the save on the **New Basis File**, there will still be a usable basis on the **Backup Basis File** (corresponding to iteration 1900).

Note that a new basis will be saved in **New Basis File** at the end of a run if it terminates normally, but it will not be saved in **Backup Basis File**. In the above example, if an optimum solution is found at iteration

2050 (or if the iteration limit is 2050), the final basis in the file **New Basis File** will correspond to iteration 2050, but the last basis saved in the file **Backup Basis File** will be the one for iteration 2000.

A full description of information recorded in **New Basis File** and **Backup Basis File** is given in Gill *et al.* (1999).

New Superbasics Limit – Integer i Default $= 99$

This option causes early termination of the QP subproblems if the number of free variables has increased significantly since the first feasible point. If the number of new superbasics is greater than i the nonbasic variables that have not yet moved are frozen and the resulting smaller QP is solved to optimality.

In the major iteration log (see Section), a 'T' at the end of a line indicates that the QP was terminated early in this way.

Old Basis File – Nag_FileID i Default $= 0$

(See Section 11.1 for a description of **Nag_FileID**.)

If **Old Basis File** > 0, the basis maps information will be obtained from the file associated with ID i. A full description of information recorded in **New Basis File** and **Backup Basis File** is given in Gill *et al.* (1999). The file will usually have been output previously as a **New Basis File** or **Backup Basis File**.

The file will not be acceptable if the number of rows or columns in the problem has been altered.

Partial Price – Integer i Default $= 1$

This argument is recommended for large problems that have significantly more variables than constraints. It reduces the work required for each 'pricing' operation (when a nonbasic variable is selected to become superbasic).

When $i = 1$, all columns of the constraint matrix $(A \quad -I)$ are searched.

Otherwise, A and I are partitioned to give i roughly equal segments A_j, I_j, for $j = 1, \ldots, i$. 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}. (All subscripts here are modulo i.)

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 become superbasic. If nothing is found, the search continues on the next segments A_{j+2} I_{j+2}, and so on.

Partial Price r (or $r/2$ or $r/3$) may be appropriate for time-stage models having r time periods.

Pivot Tolerance – double r Default $= 10 \times \epsilon$

During the solution of QP subproblems, the pivot tolerance is used 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 search direction p, a 'ratio test' is used to determine which component 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 the pivot tolerance r.

It is common for two or more variables to reach a bound at essentially the same time. In such cases, the **Minor Feasibility Tolerance** (say, t) provides some freedom to maximize the pivot element and thereby improve numerical stability. Excessively small values of t should therefore not be specified.

To a lesser extent, the **Expand Frequency** (say, f) also provides some freedom to maximize the pivot element. Excessively *large* values of f should therefore not be specified.

Print File – Nag_FileID i Default $= 0$

(See Section 11.1 for a description of **Nag_FileID**.)

If **Print File** > 0, the following information is output to a file associated with ID i during the solution of each problem:

– a listing of the optional arguments;

– some statistics about the problem;

– the amount of storage available for the LU factorization of the basis matrix;

– notes about the initial basis resulting from a Crash procedure or a basis file;

– the iteration log;

– basis factorization statistics;

– the exit **fail** condition and some statistics about the solution obtained;

– the printed solution, if requested.

These items are described in Sections 8 and 12. Further brief output may be directed to the **Summary File**.

Print Frequency – Integer i Default $= 100$

If $i > 0$, one line of the iteration log will be printed every ith iteration. A value such as $i = 10$ is suggested for those interested only in the final solution.

Proximal Point Method – Integer i Default $= 1$

$i = 1$ or 2 specifies minimization of $\|x - x_0\|_1$ or $\frac{1}{2}\|x - x_0\|_2{}^2$ when the starting point x_0 is changed to satisfy the linear constraints (where x_0 refers to nonlinear variables).

Punch File – Nag_FileID i_1 Default $= 0$
Insert File – Nag_FileID i_2 Default $= 0$

(See Section 11.1 for a description of **Nag_FileID**.)

The **Punch File** from a previous run may be used as an **Insert File** for a later run on the same problem. A full description of information recorded in **Insert File** and **Punch File** is given in Gill *et al.* (1999).

If **Punch File** > 0, the final solution obtained will be output to the file associated with ID i_1. For linear programs, this format is compatible with various commercial systems.

If **Insert File** > 0, the file associated with ID i_2, containing basis information, will be read. The file will usually have been output previously as a **Punch File**. The file will not be accessed if **Old Basis File** is specified.

Scale Option – Integer i Default $= 0$
Scale Tolerance – double r Default $= 0.9$

Three scale options are available as follows:

i	Meaning
0	No scaling. This is recommended if it is known that x and the constraint matrix never have very large elements (say, larger than 1000).
1	The constraints and variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0 (see Fourer (1982)). This will sometimes improve the performance of the solution procedures.
2	The constraints and variables are scaled by the iterative procedure. Also, a certain additional scaling is performed that may be helpful if the right-hand side b or the solution x is large. This takes into account columns of $\begin{pmatrix} A & -I \end{pmatrix}$ that are fixed or have positive lower bounds or negative upper bounds.

Scale Tolerance affects how many passes might be needed through the constraint matrix. On each pass, the scaling procedure computes the ratio of the largest and smallest non-zero coefficients in each column:

$$\rho_j = \max_i |a_{ij}| / \min_i |a_{ij}| \quad (a_{ij} \neq 0).$$

If $\max \rho_j$ is less than r times its previous value, another scaling pass is performed to adjust the row and column scales. Raising r from 0.9 to 0.99 (say) usually increases the number of scaling passes through A. At most 10 passes are made.

Solution File – Nag_FileID i Default $= 0$

(See Section 11.1 for a description of **Nag_FileID**.)

If **Solution File** > 0, the final solution will be output to the file associated with ID i.

To see more significant digits in the printed solution, it will sometimes be useful to specify that the **Solution File** refers to the **Print File**.

Start Objective Check At Variable – Integer i_1 Default $= 1$
Stop Objective Check At Variable – Integer i_2 Default $= n$
Start Constraint Check At Variable – Integer i_3 Default $= 1$
Stop Constraint Check At Variable – Integer Default $= n$

These keywords take effect only if **Verify Level** > 0. They may be used to control the verification of gradient elements computed by function **objfun** and/or Jacobian elements computed by function **confun**. For example, if the first 30 elements of the objective gradient appeared to be correct in an earlier run, so that only element 31 remains questionable, it is reasonable to specify **Start Objective Check At Variable** $= 31$. If the first 30 variables appear linearly in the objective, so that the corresponding gradient elements are constant, the above choice would also be appropriate.

Summary File – Nag_FileID i_1 Default $= 0$
Summary Frequency – Integer i_2 Default $= 100$

(See Section 11.1 for a description of **Nag_FileID**.)

If **Summary File** > 0, a brief log will be output to the file associated with i_1, including one line of information every i_2th iteration. In an interactive environment, it is useful to direct this output to the terminal, to allow a run to be monitored on-line. (If something looks wrong, the run can be manually terminated.) Further details are given in Section 12.6.

Superbasics Limit – Integer i Default $= \min(500, n_1 + 1)$

This option places a limit on the storage allocated for superbasic variables. Ideally, i should be set slightly larger than the 'number of degrees of freedom' expected at an optimal solution.

For linear programs, an optimum is normally a basic solution with no degrees of freedom. (The number of variables lying strictly between their bounds is no more than m, the number of general constraints.) The default value of i is therefore 1.

For nonlinear problems, the number of degrees of freedom is often called the 'number of independent variables'.

Normally, i need not be greater than $n_1 + 1$, where n_1 is the number of nonlinear variables.

For many problems, i may be considerably smaller than n_1. This will save storage if n_1 is very large.

Suppress Parameters

Normally nag_opt_nlp_solve (e04wdc) prints the options file as it is being read, and then prints a complete list of the available keywords and their final values. The **Suppress Parameters** option tells nag_opt_nlp_solve (e04wdc) not to print the full list.

Timing Level – Integer i Default $= 0$

If $i > 0$, some timing information will be output to the **Print File**.

Unbounded Objective – double r_1 Default $= 1.0\text{e}+15$
Unbounded Step Size – double r_2 Default $= 1.0\text{e}+20$

These arguments are intended to detect unboundedness in nonlinear problems. During a line search, F is evaluated at points of the form $x + \alpha p$, where x and p are fixed and α varies. If $|F|$ exceeds r_1 or α exceeds r_2, iterations are terminated with the exit message

```
Problem is unbounded (or badly scaled)
```

If singularities are present, unboundedness in $F(x)$ may be manifested by a floating-point overflow (during the evaluation of $F(x + \alpha p)$), before the test against r_1 can be made.

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

Verify Level – Integer i Default $= 0$

This option refers to finite-difference checks on the derivatives computed by the user-supplied functions. Derivatives are checked at the first point that satisfies all bounds and linear constraints.

i	**Meaning**
0	Only a 'cheap' test will be performed, requiring two calls to **confun**.
1	Individual gradients will be checked (with a more reliable test). A key of the form `OK` or `Bad?` indicates whether or not each component appears to be correct.
2	Individual columns of the problem Jacobian will be checked.
3	Options 2 and 1 will both occur (in that order).
-1	Derivative checking is disabled.

Verify Level $= 3$ should be specified whenever a new user function is being developed. The **Start** and **Stop** keywords may be used to limit the number of nonlinear variables checked. Missing derivatives are not checked, so they result in no overhead.

Violation Limit – double r Default $= 1.0\text{e}+6$

This keyword defines an absolute limit on the magnitude of the maximum constraint violation, r, after the line search. On completion of the line search, the new iterate x_{k+1} satisfies the condition

$$v_i(x_{k+1}) \leq r \ \max\{1, v_i(x_0)\},$$

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

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

12 Description of Monitoring Information

nag_opt_nlp_solve (e04wdc) produces monitoring information, statistical information and information about the solution. Section 8.1 contains the final output information sent to **Print File**. This section contains other output information.

12.1 Major Iteration Log

This section describes the output to **Print File** if **Major Print Level** > 0. One line of information is output every kth major iteration, where k is **Print Frequency** (see Section 11.2).

Label	**Description**
`Itns`	is the cumulative number of minor iterations.
`Major`	is the current major iteration number.
`Minors`	is the number of iterations required by both the feasibility and optimality phases of the QP subproblem. Generally, `Minors` will be 1 in the later iterations, since

theoretical analysis predicts that the correct active set will be identified near the solution (see Section 10).

`Step` is the step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.

`nCon` the number of times function **confun** has been called to evaluate the nonlinear problem functions. Evaluations needed for the estimation of the derivatives by finite differences are not included. `nCon` is printed as a guide to the amount of work required for the line search.

`Feasible` is the value of *rowerr* (see (12)), the maximum component of the scaled nonlinear constraint residual (see **Major Feasibility Tolerance** in Section 11.2). The solution is regarded as acceptably feasible if `Feasible` is less than the **Major Feasibility Tolerance**. In this case, the entry is contained in parentheses.

If the constraints are linear, all iterates are feasible and this entry is not printed.

`Optimal` is the value of *maxComp* (see (13)), the maximum complementary gap (see **Major Optimalility Tolerance** in Section 11.2). It is an estimate of the degree of nonoptimality of the reduced costs. Both `Feasible` and `Optimal` are small in the neighbourhood of a solution.

`MeritFunction` is the value of the augmented Lagrangian merit function (see (7)). This function will decrease at each iteration unless it was necessary to increase the penalty arguments (see Section 10.4). As the solution is approached, `MeritFunction` will converge to the value of the objective at the solution.

In elastic mode, the merit function is a composite function involving the constraint violations weighted by the elastic weight.

If the constraints are linear, this item is labelled `Objective`, the value of the objective function. It will decrease monotonically to its optimal value.

`L+U` is the number of non-zeros representing the basis factors L and U on completion of the QP subproblem.

If nonlinear constraints are present, the basis factorization $B = LU$ is computed at the start of the first minor iteration. At this stage, $\mathtt{LU} = \mathtt{lenL} + \mathtt{lenU}$, where `LenL` (see Section 12.3) is the number of subdiagonal elements in the columns of a lower triangular matrix and `lenU` (see Section 12.3) is the number of diagonal and superdiagonal elements in the rows of an upper-triangular matrix.

As columns of B are replaced during the minor iterations, `LU` may fluctuate up or down but, in general, will tend to increase. As the solution is approached and the minor iterations decrease towards zero, `LU` will reflect the number of non-zeros in the LU factors at the start of the QP subproblem.

If the constraints are linear, refactorization is subject only to the **Factorization Frequency**, and `LU` will tend to increase between factorizations.

`BSwap` is the number of columns of the basis matrix B that were swapped with columns of S to improve the condition of B. The swaps are determined by an LU factorization of the rectangular matrix $B_S = \begin{pmatrix} B & S \end{pmatrix}^{\mathrm{T}}$ with stability being favoured more than sparsity.

`nS` is the current number of superbasic variables.

`CondHz` is an estimate of the condition number of $R^{\mathrm{T}}R$, an estimate of $Z^{\mathrm{T}}HZ$, the reduced Hessian of the Lagrangian. It is the square of the ratio of the largest and smallest diagonals of the upper triangular matrix R (which is a lower bound on the condition number of $R^{\mathrm{T}}R$). `CondHz` gives a rough indication of whether or not the optimization procedure is having difficulty. If ϵ is the relative ***machine precision*** being used, the SQP algorithm will make slow progress if `CondHz` becomes as large as $\epsilon^{-1/2} \approx 10^8$, and will probably fail to find a better solution if `CondHz` reaches $\epsilon^{-3/4} \approx 10^{12}$.

To guard against high values of `CondHz`, attention should be given to the scaling of the variables and the constraints. In some cases it may be necessary to add upper or lower bounds to certain variables to keep them a reasonable distance from singularities in the nonlinear functions or their derivatives.

`Penalty` is the Euclidean norm of the vector of penalty arguments used in the augmented Lagrangian merit function (not printed if there are no nonlinear constraints).

The summary line may include additional code characters that indicate what happened during the course of the major iteration.

Label	**Description**
`c`	central differences have been used to compute the unknown components of the objective and constraint gradients. A switch to central differences is made if either the line search 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.
`d`	during the line search it was necessary to decrease the step in order to obtain a maximum constraint violation conforming to the value of **Violation Limit**.
`l`	the norm wise change in the variables was limited by the value of the **Major Step Limit**. If this output occurs repeatedly during later iterations, it may be worthwhile increasing the value of **Major Step Limit**.
`i`	if nag_opt_nlp_solve (e04wdc) is not in elastic mode, an `i` signifies that the QP subproblem is infeasible. 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 (11) (see Section 10.5). If nag_opt_nlp_solve (e04wdc) is already in elastic mode, an `i` indicates that the minimizer of the elastic subproblem does not satisfy the linearized constraints. (In this case, a feasible point for the usual QP subproblem may or may not exist.)
`M`	an extra evaluation of the problem functions was needed to define an acceptable positive-definite quasi-Newton update to the Lagrangian Hessian. This modification is only done when there are nonlinear constraints.
`m`	this is the same as `M` except that it was also necessary to modify the update to include an augmented Lagrangian term.
`n`	no positive-definite BFGS update could be found. The approximate Hessian is unchanged from the previous iteration.
`R`	the approximate Hessian has been reset by discarding all but the diagonal elements. This reset will be forced periodically by the **Hessian Frequency** and **Hessian Updates** keywords (see Section 11.2. However, it may also be necessary to reset an ill-conditioned Hessian from time to time.
`r`	the approximate Hessian was reset after ten consecutive major iterations in which no BFGS update could be made. The diagonals of the approximate Hessian are retained if at least one update has been done since the last reset. Otherwise, the approximate Hessian is reset to the identity matrix.
`s`	a self-scaled BFGS update was performed. This update is always used when the Hessian approximation is diagonal, and hence always follows a Hessian reset.
`t`	the minor iterations were terminated because of the **Minor Iterations Limit** (see Section 11.2).
`T`	the minor iterations were terminated because of the **New Superbasics Limit** (see Section 11.2).

`u`	the QP subproblem was unbounded.
`w`	a weak solution of the QP subproblem was found.
`z`	the **Superbasics Limit** was reached (see Section 11.2).

12.2 Minor Iteration Log

If **Minor Print Level** > 0, one line of information is output to the **Print File** every kth minor iteration, where k is the specified **Print Frequency** (see Section 11.2). A heading is printed before the first such line following a basis factorization. The heading contains the items described below. In this description, 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). The selected variable is denoted by `jq`. Variable `jq` often becomes basic immediately. Otherwise it remains superbasic, unless it reaches its opposite bound and returns to the nonbasic set.

If **Partial Price** is in effect (see Section 11.2), variable `jq` is selected from A_{pp} or I_{pp}, the ppth segments of the constraint matrix $(A \quad -I)$.

Label **Description**

`Itn` the current iteration number.

`RedCost` or `QPmult` is the reduced cost (or reduced gradient) of the variable `jq` selected by the pricing procedure at the start of the present itearation. Algebraically, `dg` is $d_j = g_j - \pi^{\mathrm{T}} a_j$ for $j =$ `jq`, where g_j is the gradient of the current objective function, π is the vector of dual variables for the QP subproblem, and a_j is the jth column of $(A \quad -I)$.

Note that `dj` is the 1-norm of the reduced-gradient vector at the start of the iteration, just after the pricing procedure.

`LPstep` or `QPstep` is 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 (`+SBS` > 0), `Step` will be the step to the nearest bound. During Phase 2, the step can be greater than one only if the reduced Hessian is not positive-definite.

`nInf` is the number of infeasibilities *after* the present iteration. This number will not increase unless the iterations are in elastic mode.

`SumInf` If `nInf` > 0, this is `sInf`, the sum of infeasibilities after the present iteration. It usually decreases at each non-zero `Step`, but if `nInf` decreases by 2 or more, `SumInf` may occasionally increase.

In elastic mode, the heading is changed to `Composite Obj`, and the value printed decreases monotonically.

`rgNorm` is the norm of the reduced-gradient vector at the start of the iteration. (It is the norm of the vector with elements d_j for variables j in the superbasic set.) During Phase 2 this norm will be approximately zero after a unit step.

(The heading is not printed if the problem is linear.)

`LPobjective` or `QPobjective`
the QP objective function after the present iteration. In elastic mode, the heading is changed to `Elastic QPobj`. In either case, the value printed decreases monotonically.

`+SBS` is the variable `jq` selected by the pricing operation to be added to the superbasic set.

`-SBS` is the variable chosen to leave the set of superbasics.

`-BS` is the variable removed from the basis (if any) to become nonbasic.

`Pivot` if column a_q replaces the rth column of the basis B, `Pivot` is the rth element of a vector y satisfying $By = a_q$. Wherever possible, `Step` is chosen to avoid extremely small values of `Pivot` (since they cause the basis to be nearly singular). In rare

	cases, it may be necessary to increase the **Pivot Tolerance** to exclude very small elements of y from consideration during the computation of `Step`.
`L+U`	is the number of non-zeros representing the basis factors L and U. Immediately after a basis factorization $B = LU$, this is `lenL+lenU`, the number of subdiagonal elements in the columns of a lower triangular matrix and the number of diagonal and superdiagonal elements in the rows of an upper-triangular matrix. Further non-zeros are added to `L` when various columns of B are later replaced. As columns of B are replaced, the matrix U is maintained explicitly (in sparse form). The value of `L` will steadily increase, whereas the value of `U` may fluctuate up or down. Thus the value of `L+U` may fluctuate up or down (in general, it will tend to increase).
`ncp`	is the number of compressions required to recover storage in the data structure for U. This includes the number of compressions needed during the previous basis factorization.
`nS`	is the current number of superbasic variables. (The heading is not printed if the problem is linear.)
`CondHz`	see the major iteration log. (The heading is not printed if the problem is linear.)

12.3 Basis Factorization Statistics

If **Major Print Level** ≥ 10, the following items are output to the **Print File** whenever the basis B or the rectangular matrix $B_S = \begin{pmatrix} B & S \end{pmatrix}^{\mathrm{T}}$ is factorized before solution of the next QP subproblem. See Section 11.2 for a full description of an optional argument.

Note that B_S may be factorized at the start of just some of the major iterations. It is immediately followed by a factorization of B itself.

Gaussian elimination is used to compute a sparse LU factorization of B or B_S, where PLP^{T} and PUQ are lower and upper triangular matrices for some permutation matrices P and Q. Stability is ensured as described under **LU Factor Tolerance**.

If **Minor Print Level** ≥ 10, the same items are printed during the QP solution whenever the current B is factorized.

Label	**Description**
`Factorize`	the number of factorizations since the start of the run.
`Demand`	a code giving the reason for the present factorization.

Code	**Meaning**
0	First LU factorization.
1	The number of updates reached the **Factorization Frequency**.
2	The non-zeros in the updated factors have increased significantly.
7	Not enough storage to update factors.
10	Row residuals too large (see the description of **Check Frequency**).
11	Ill-conditioning has caused inconsistent results.

`Itn`	is the current minor iteration number.
`Nonlin`	is the number of nonlinear variables in the current basis B.
`Linear`	is the number of linear variables in B.
`Slacks`	is the number of slack variables in B.

`B BR BS or BT factorize`

is the type of LU factorization.

`B`	periodic factorization of the basis B.
`BR`	more careful rank-revealing factorization of B using threshold rook pivoting. This occurs mainly at the start, if the first basis factors seem singular or ill-conditioned. Followed by a normal `B` factorize.

BS	B_S is factorized to choose a well-conditioned B from the current $(B\ S)$. Followed by a normal B factorize.
BT	same as BS except the current B is tried first and accepted if it appears to be not much more ill-conditioned than after the previous BS factorize.

m — is the number of rows in B or B_S.

n — is the number of columns in B or B_S. Preceded by '=' or '>' respectively.

Elems — is the number of non-zero elements in B or B_S.

Amax — is the largest non-zero in B or B_S.

Density — is the percentage non-zero density of B or B_S.

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 n 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.

Cmpressns — is the number of times the data structure holding the partially factored matrix needed to be compressed to recover unused storage. Ideally this number should be zero. If it is more than 3 or 4, the amount of workspace available to nag_opt_nlp_solve (e04wdc) should be increased for efficiency.

Incres — is the percentage increase in the number of non-zeros in L and U relative to the number of non-zeros in B or B_S.

Utri — is the number of triangular rows of B or B_S at the top of U.

lenU — the number of non-zeros in U.

Ltol — is the maximum subdiagonal element allowed in L. This is the specified **LU Factor Tolerance** or a smaller value that is currently being used for greater stability.

Umax — the maximum non-zero element in U.

Ugrwth — is the ratio Umax/Amax, which ideally should not be substantially larger than 10.0 or 100.0. If it is orders of magnitude larger, it may be advisable to reduce the **LU Factor Tolerance** to 5.0, 4.0, 3.0 or 2.0, say (but bigger than 1.0).

As long as Lmax is not large (say 10.0 or less), max{Amax, Umax}/DUmin gives an estimate of the condition number B. If this is extremely large, the basis is nearly singular. Slacks are used to replace suspect columns of B and the modified basis is refactored.

Ltri — is the number of triangular columns of B or B_S at the left of L.

dense1 — is the number of columns remaining when the density of the basis matrix being factorized reached 0.3.

Lmax — is the actual maximum subdiagonal element in L (bounded by Ltol).

Akmax — is the largest non-zero generated at any stage of the LU factorization. (Values much larger than Amax indicate instability.)

growth — is the ratio Akmax/Amax. Values much larger than 100 (say) indicate instability.

bump — is the size of the 'bump' or block to be factorized nontrivially after the triangular rows and columns of B or B_S have been removed.

dense2 — is the number of columns remaining when the density of the basis matrix being factorized reached 0.6. (The Markowitz pivot strategy searches fewer columns at that stage.)

DUmax — is the largest diagonal of PUQ.

`DUmin` is the smallest diagonal of PUQ.

`condU` the ratio `DUmax/DUmin`, which estimates the condition number of U (and of B if `Ltol` is less than 100, say).

12.4 Crash Statistics

If **Major Print Level** ≥ 10, the following items are output to the **Print File** when **Cold Start** and no basis file is loaded (see Section 11.2). They refer to the number of columns that the Crash procedure selects during selected passes through A while searching for a triangular basis matrix.

Label	**Description**
`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., $\texttt{hs}(j) = 3$ for some $j \leq n$). It will be a subset of the columns for which $\texttt{hs}(j) = 3$ was specified.
`Unit`	is the number of unit columns in the basis.
`Double`	is the number of columns in the basis containing 2 non-zeros.
`Triangle`	is the number of triangular columns in the basis with 3 or more non-zeros.
`Pad`	is the number of slacks used to pad the basis (to make it a nonsingular triangle).

12.5 The Solution File

At the end of a run, the final solution may be output as a solution file, according to **Solution File** (see Section 11.2). Some header information appears first to identify the problem and the final state of the optimization procedure. A ROWS section and a COLUMNS section then follow, giving one line of information for each row and column. The format used is similar to certain commercial systems, though there is no industry standard.

The maximum record length is 111 characters.

To reduce clutter, a full stop (.) is printed for any numerical value that is exactly zero. The values ± 1 are also printed specially as 1.0 and -1.0. Infinite bounds ($\pm 10^{20}$ or larger) are printed as `None`.

A solution file is intended to be read from disk by a self-contained program that extracts and saves certain values as required for possible further computation. Typically, the first 14 records would be ignored. Each subsequent record may be read adapted to suit the occasion. The end of the ROWS section is marked by a record that starts with a 1 and is otherwise blank. If this and the next 4 records are skipped, the COLUMNS section can then be read under the same format.

The ROWS section

The general constraints take the form $l \leq Ax \leq u$. The ith constraint is therefore of the form

$$\alpha \leq \nu_i^{\mathrm{T}} x \leq \beta,$$

where ν_i is the ith row of A.

Internally, the constraints take the form $Ax - s = 0$, where s is the set of slack variables (which happen to satisfy the bounds $l \leq s \leq u$). For the ith constraint it is the slack variable s_i that is directly available, and it is sometimes convenient to refer to its state. A '.' is printed for any numerical value that is exactly zero.

Label	**Description**
`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 v_i.
`State`	the state of vi (the state of s_i relative to the bounds α and β. The various states possible are as follows:

`LL`	s_i is nonbasic at its lower limit, α.
`UL`	s_i is nonbasic at its upper limit, β.
`EQ`	s_i is nonbasic and fixed at the value $\alpha = \beta$.
`FR`	s_i is nonbasic and currently zero, even though it is free to take any value between its bounds α and β.
`BS`	s_i is basic.
`SBS`	s_i is superbasic.

A key is sometimes printed before `State` to give some additional information about the state of a variable. Note that unless the optional argument **Scale Option** $= 0$ (see Section 11.2) is specified, the tests for assigning a key are applied to the variables of the scaled problem.

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

`D` *Degenerate*. The variable is basic or superbasic, but it is equal to (or very close to) one of its bounds.

`I` *Infeasible*. The variable is basic or superbasic and is currently violating one of its bounds by more than the value of the optional argument **Feasibility Tolerance** (see Section 11.2).

`N` *Not precisely optimal*. The variable is nonbasic or superbasic. If the value of the reduced gradient for the variable exceeds the value of the optional argument **Major Optimality Tolerance** (see Section 11.2), the solution would not be declared optimal because the reduced gradient for the variable would not be considered negligible.

`Activity` is the value of v_i at the final iterate (the ith element of $A^{\mathrm{T}}x$).

`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 variable s_i. `None` indicates that $\mathbf{bl}[j] \leq -bigbnd$.

`Upper Bound` is β, the upper bound specified for the variable s_i. `None` indicates that $\mathbf{bu}[j] \geq bigbnd$.

`Dual Activity` is the value of the dual variable π_i (the Lagrange-multiplier for the ith constraints). The full vector π always satisfies $B^{\mathrm{T}}\pi = g_B$, where B is the current basis matrix and g_B contains the associated gradients for the current objective function. For FP problems, π_i is set to zero.

`i` gives the index i of the ith row.

The COLUMNS section

Let the jth component of x be the variable x_j and assume that it satisfies the bounds $\alpha \leq x_j \leq \beta$. A '.' is printed for any numerical value that is exactly zero.

Label	**Description**
`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` the state of x_j relative to the bounds α and β. The various states possible 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 and currently zero, even though it is free to take any value between its bounds α and β.
- `BS` x_j is basic.
- `SBS` x_j is superbasic.

A key is sometimes printed before `State` to give some additional information about the state of a variable. Note that unless the optional argument **Scale Option** $= 0$ (see Section 11.2) is specified, the tests for assigning a key are applied to the variables of the scaled problem.

- `A` *Alternative optimum possible*. The variable is nonbasic, but its reduced gradient is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change in the value of the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them could encounter a bound immediately. In either case, the values of the Lagrange-multipliers *might* also change.
- `D` *Degenerate*. The variable is basic or superbasic, but it is equal to (or very close to) one of its bounds.
- `I` *Infeasible*. The variable is basic or superbasic and is currently violating one of its bounds by more than the value of the optional argument **Feasibility Tolerance** (see Section 11.2).
- `N` *Not precisely optimal*. The variable is nonbasic or superbasic. If the value of the reduced gradient for the variable exceeds the value of the optional argument **Major Optimality Tolerance** (see Section 11.2), the solution would not be declared optimal because the reduced gradient for the variable would not be considered negligible.

`Activity` is the value of x_j at the final iterate.

`Obj Gradient` is the value of g_j at the final iterate. For FP problems, g_j is set to zero.

`Lower Bound` is the lower bound specified for the variable. `None` indicates that **bl**$[j] \leq -bigbnd$.

`Upper Bound` is the upper bound specified for the variable. `None` indicates that **bu**$[j] \geq bigbnd$.

`Reduced Gradnt` is the value of the reduced gradient $d_j = g_j - \pi^{\mathrm{T}} a_j$ where a_j is the jth column of the constraint matrix. For FP problems, d_j is set to zero.

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

12.6 The Summary File

If **Summary File** > 0, the following information is output to the **Summary File** (see Section 11.2). (It is a brief summary of the output directed to **Print File**):

– the optional arguments supplied via the option setting functions, if any;

– the basis file loaded, if any;

– a brief major iteration log (see Section);

– a brief minor iteration log (see Section);

– the exit condition, **fail**;

– a summary of the final iterate.