NAG Library Function Document

nag_opt_nlp (e04ucc)

1 Purpose

nag_opt_nlp (e04ucc) 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. You should supply as many first derivatives as possible; any unspecified derivatives are approximated by finite differences. It is not intended for large sparse problems.

nag_opt_nlp (e04ucc) may also be used for unconstrained, bound-constrained and linearly constrained optimization.

2 Specification

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

```
void nag_opt_nlp (Integer n, Integer nclin, Integer ncnlin, const double a[],
     Integer tda, const double bl[], const double bu[],
     void (*objfun)(Integer n, const double x[], double *objf, double g[],
          Nag_Comm *comm),
     void (*confun)(Integer n, Integer ncnlin, const Integer needc[],
          const double x[], double conf[], double conjac[], Nag_Comm *comm),
     double x[], double *objf, double g[], Nag_E04_Opt *options,
     Nag_Comm *comm, NagError *fail)
```

3 Description

nag_opt_nlp (e04ucc) is designed to solve the nonlinear programming problem – the minimization of a smooth nonlinear function subject to a set of constraints on the variables. The problem is assumed to be stated in the following form:

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

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 (e04ucc) 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 optional argument **options.inf_bound** in Section 12.2.)

If there are no nonlinear constraints in (1) and F is linear or quadratic, then one of nag_opt_lp (e04mfc), nag_opt_lin_lsq (e04ncc) or nag_opt_qp (e04nfc) will generally be more efficient.

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 12.2 for a discussion of the optional arguments **options.obj_deriv** and **options.con_deriv**. Just before either **objfun** or **confun** is called, each element of the current gradient array **g** or **conjac** 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 if you 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 optional argument **options.verify_grad** (see Section 12.2) should be used to check the calculation of any known gradients.

The method used by nag_opt_nlp (e04ucc) is described in detail in Section 11.

4 References

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

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

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

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

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

Gill P E, Murray W, Saunders M A and Wright M H (1983) Documentation for FDCALC and FDCORE *Technical Report SOL* 83–6 Stanford University

Gill P E, Murray W, Saunders M A and Wright M H (1984a) Users' Guide for SOL/QPSOL Version 3.2 *Report* SOL 84–5 Department of Operations Research, Stanford University

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

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

Gill P E, Murray W, Saunders M A and Wright M H (1986b) 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 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

Murtagh B A and Saunders M A (1983) MINOS 5.0 user's guide *Report SOL 83-20* Department of Operations Research, Stanford University

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

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

5 Arguments

1: **n** – Integer *Input*

On entry: n, the number of variables.

Constraint: **n** > 0.

2: **nclin** – Integer *Input*

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

Constraint: **nclin** ≥ 0.

3: **ncnlin** – Integer *Input*

On entry: n_N, the number of nonlinear constraints.

Constraint: **ncnlin** ≥ 0.

4: **a**[**nclin** $\times$ **tda**] – const double *Input*

On entry: the ith row of **a** must contain the coefficients of the ith general linear constraint (the ith row of the matrix A_L in (1)). The ijth element of A_L must be stored in **a**[$i - 1 \times$ **tda** $+ j - 1$], for $i = 1, 2, \ldots, n_L$.

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

5: **tda** – Integer *Input*

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

Constraint: if **nclin** > 0, **tda** $\geq$ **n**

6: **bl**[**n** + **nclin** + **ncnlin**] – const double *Input*
7: **bu**[**n** + **nclin** + **ncnlin**] – 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 nonlinear constraints (if any). To specify a nonexistent lower bound (i.e., $l_j = -\infty$), set **bl**[$j - 1$] $\leq -$**options.inf_bound**, and to specify a nonexistent upper bound (i.e., $u_j = +\infty$), set **bu**[$j - 1$] $\geq$ **options.inf_bound**, where **options.inf_bound** is one of the optional arguments (default value 10^{20}, see Section 12.2). To specify the jth constraint as an equality, set **bl**[$j - 1$] = **bu**[$j - 1$] $= \beta$, say, where $|\beta| <$ **options.inf_bound**.

Constraints:

bl[$j - 1$] $\leq$ **bu**[$j - 1$], for $j = 1, 2, \ldots,$ **n** + **nclin** + **ncnlin**;
if **bl**[$j - 1$] = **bu**[$j - 1$] $= \beta$, $|\beta| <$ **options.inf_bound**.

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

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

The specification of **objfun** is:

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

1: **n** – Integer *Input*

On entry: n, the number of variables.

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

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

3: **objf** – double * *Output*

On exit: if **comm→flag** $= 0$ or 2, **objfun** must set **objf** to the value of the objective function F at the current point x. If it is not possible to evaluate F then **objfun** should assign a negative value to **comm→flag**; nag_opt_nlp (e04ucc) will then terminate.

4: **g**[**n**] – double *Output*

On exit: if **comm→flag** $= 2$, **g** must contain the elements of the vector $g(x)$ given by

$$g(x) = \left(\frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, \ldots, \frac{\partial F}{\partial x_n}\right)^{\mathrm{T}},$$

where $\frac{\partial F}{\partial x_i}$ is the partial derivative of the objective function with respect to the ith variable evaluated at the point x , for $i = 1, 2, \ldots, n$.

If the optional argument **options.obj_deriv** = Nag_TRUE (the default), all elements of **g** must be set; if **options.obj_deriv** = Nag_FALSE, any available elements of the vector $g(x)$ must be assigned to the elements of **g**; the remaining elements *must remain unchanged*.

5: **comm** – Nag_Comm *

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

flag – Integer *Input/Output*

On entry: **objfun** is called with **comm→flag** set to 0 or 2.

If **comm→flag** $= 0$ then only **objf** is referenced.

If **comm→flag** $= 2$ then both **objf** and **g** are referenced.

On exit: if **objfun** resets **comm→flag** to some negative number then nag_opt_nlp (e04ucc) will terminate immediately with the error indicator NE_USER_STOP. If **fail** is supplied to nag_opt_nlp (e04ucc), **fail.errnum** will be set to your setting of **comm→flag**.

first – Nag_Boolean *Input*

On entry: will be set to Nag_TRUE on the first call to **objfun** and Nag_FALSE for all subsequent calls.

nf – Integer *Input*

On entry: the number of evaluations of the objective function; this value will be equal to the number of calls made to **objfun** including the current one.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.

Before calling nag_opt_nlp (e04ucc) these pointers may be allocated memory and initialized with various quantities for use by **objfun** when called from nag_opt_nlp (e04ucc).

Note: **objfun** should be tested separately before being used in conjunction with nag_opt_nlp (e04ucc). The optional arguments **options.verify_grad** and **options.max_iter** can be used to assist this process. The array **x** must **not** be changed by **objfun**.

If the function **objfun** does not calculate all of the gradient elements then the optional argument **options.obj_deriv** should be set to Nag_FALSE.

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

confun must calculate the vector $c(x)$ of nonlinear constraint functions and (optionally) its Jacobian ($= \frac{\partial c}{\partial x}$) for a specified n element vector x. If there are no nonlinear constraints (i.e., **ncnlin** $= 0$), **confun** will never be called and the NAG defined null void function pointer, `NULLFN`, can be supplied in the call to nag_opt_nlp (e04ucc). If there are nonlinear constraints the first call to **confun** will occur before the first call to **objfun**.

The specification of **confun** is:

```
void confun (Integer n, Integer ncnlin, const Integer needc[],
     const double x[], double conf[], double conjac[], Nag_Comm *comm)
```

1: **n** – Integer *Input*

On entry: n, the number of variables.

2: **ncnlin** – Integer *Input*

On entry: n_N, the number of nonlinear constraints.

3: **needc**[**ncnlin**] – const Integer *Input*

On entry: the indices of the elements of **conf** and/or **conjac** that must be evaluated by **confun**. If **needc**$[i-1] > 0$ then the ith element of **conf** and/or the available elements of the ith row of **conjac** (see argument **comm→flag** below) must be evaluated at x.

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

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

5: **conf**[**ncnlin**] – double *Output*

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

6: **conjac**[**ncnlin** $\times$ **n**] – double *Output*

On exit: if **needc**$[i-1] > 0$ and **comm→flag** $= 2$, the ith row of **conjac** (i.e., the elements **conjac**$[(i-1) \times$ **n** $+ j - 1]$, for $j = 1, 2, \ldots, n$) 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. The remaining rows of **conjac**, corresponding to non-positive elements of **needc**, are ignored.

If the optional argument **options.con_deriv** = Nag_TRUE (the default), all elements of **conjac** must be set; if **options.con_deriv** = Nag_FALSE, then any available partial derivatives of $c_i(x)$ must be assigned to the elements of **conjac**; the remaining elements *must remain unchanged*.

If all elements of the constraint Jacobian are known (i.e., **options.con_deriv** = Nag_TRUE; see Section 12.2), any constant elements may be assigned to **conjac** one time only at the start of the optimization. An element of **conjac** that is not subsequently assigned in **confun** will retain its initial value throughout.

Constant elements may be loaded into **conjac** during the first call to **confun**. The ability to preload constants is useful when many Jacobian elements are identically zero, in which case **conjac** may be initialized to zero at the first call when **comm→first** = Nag_TRUE.

It must be emphasized that, if **options.con_deriv** = Nag_FALSE, unassigned elements of **conjac** are not treated as constant; they are estimated by finite differences, at non-trivial expense. If you do not supply a value for the optional argument **options.f_diff_int** (the default; see Section 12.2), 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 **conjac**, which are then computed once only by finite differences.

7: **comm** – Nag_Comm *

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

flag – Integer *Input/Output*

On entry: **confun** is called with **comm→flag** set to 0 or 2.

If **comm→flag** = 0 then only **conf** is referenced.

If **comm→flag** = 2 then both **conf** and **conjac** are referenced.

On exit: if **confun** resets **comm→flag** to some negative number then nag_opt_nlp (e04ucc) will terminate immediately with the error indicator NE_USER_STOP. If **fail** is supplied to nag_opt_nlp (e04ucc) **fail.errnum** will be set to your setting of **comm→flag**.

first – Nag_Boolean *Input*

On entry: will be set to Nag_TRUE on the first call to **confun** and Nag_FALSE for all subsequent calls.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.

Before calling nag_opt_nlp (e04ucc) these pointers may be allocated memory and initialized with various quantities for use by **confun** when called from nag_opt_nlp (e04ucc).

Note: **confun** should be tested separately before being used in conjunction with nag_opt_nlp (e04ucc). The optional arguments **options.verify_grad** and **options.max_iter** can be used to assist this process. The array **x** must **not** be changed by **confun**.

If **confun** does not calculate all of the elements of the constraint gradients then the optional argument **options.con_deriv** should be set to Nag_FALSE.

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

On entry: an initial estimate of the solution.

On exit: the final estimate of the solution.

11: **objf** – double * *Output*

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

12: **g**[**n**] – double *Output*

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

13: **options** – Nag_E04_Opt * *Input/Output*

On entry/exit: a pointer to a structure of type Nag_E04_Opt whose members are optional arguments for nag_opt_nlp (e04ucc). These structure members offer the means of adjusting some of the argument values of the algorithm and on output will supply further details of the results. A description of the members of **options** is given below in Section 12. Some of the results returned in **options** can be used by nag_opt_nlp (e04ucc) to perform a 'warm start' (see the member **options**.**start** in Section 12.2).

If any of these optional arguments are required, then the structure **options** should be declared and initialized by a call to nag_opt_init (e04xxc) immediately before being supplied as an argument to nag_opt_nlp (e04ucc).

14: **comm** – Nag_Comm * *Input/Output*

Note: **comm** is a NAG defined type (see Section 3.2.1.1 in the Essential Introduction).

On entry/exit: structure containing pointers for communication to the user-supplied functions **objfun** and **confun**, and the optional user-defined printing function; see the description of **objfun** and **confun** and Section 12.3.1 for details. If you do not need to make use of this communication feature the null pointer `NAGCOMM_NULL` may be used in the call to nag_opt_nlp (e04ucc); **comm** will then be declared internally for use in calls to user-supplied functions.

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

The NAG error argument (see Section 3.6 in the Essential Introduction).

5.1 Description of the Printed Output

Intermediate and final results are printed out by default. The level of printed output can be controlled with the structure members **options**.**print_level** and **options**.**minor_print_level** (see Section 12.2). The default setting of **options**.**print_level** = Nag_Soln_Iter and **options**.**minor_print_level** = Nag_NoPrint provides a single line of output at each iteration and the final result. This section describes the default printout produced by nag_opt_nlp (e04ucc).

The following line of summary output ($<$ 80 characters) is produced at every major iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Maj` is the major iteration count.

`Mnr` is the number of minor iterations required by the feasibility and optimality phases of the QP subproblem. Generally, `Mnr` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 11).

Note that `Mnr` may be greater than the optional argument **options**.**minor_max_iter** (default value $= \max(50, 3(n + n_L + n_N))$; see Section 12.2) if some iterations are required for the feasibility phase.

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

`Merit function` is the value of the augmented Lagrangian merit function (12) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty arguments (see Section 11.3). As the solution is approached, `Merit function` will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by I at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty arguments. During a sequence of major iterations with infeasible subproblems, the sequence of Merit Function values will decrease monotonically until either a feasible subproblem is obtained or nag_opt_nlp (e04ucc) terminates with **fail.code** = NW_NONLIN_NOT_FEASIBLE (no feasible point could be found for the nonlinear constraints).

If no nonlinear constraints are present (i.e., **ncnlin** = 0), this entry contains Objective, the value of the objective function $F(x)$. The objective function will decrease monotonically to its optimal value when there are no nonlinear constraints.

Violtn is the Euclidean norm of the residuals of constraints that are violated or in the predicted active set (not printed if **ncnlin** is zero). Violtn will be approximately zero in the neighbourhood of a solution.

Norm Gz is $\|Z^{\mathrm{T}}g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient (see Section 11.1). Norm Gz will be approximately zero in the neighbourhood of a solution.

Cond Hz is a lower bound on the condition number of the projected Hessian approximation $H_Z \quad \left(H_Z = Z^{\mathrm{T}}H_{\mathrm{FR}}Z = R_Z^{\mathrm{T}}R_Z\right)$; see (6) and (11). The larger this number, the more difficult the problem.

The line of output may be terminated by one of the following characters:

M is printed if the quasi-Newton update was modified to ensure that the Hessian approximation is positive definite (see Section 11.4).

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

C is printed if central differences were used to compute the unspecified objective and constraint gradients. If the value of Step is zero, the switch to central differences was made because no lower point could be found in the line search. (In this case, the QP subproblem is re-solved with the central difference gradient and Jacobian.) If the value of Step is nonzero, central differences were computed because Norm Gz and Violtn imply that x is close to a Kuhn–Tucker point (see Section 11.1).

L is printed if the line search has produced a relative change in x greater than the value defined by the optional argument **options.step_limit** (default value $= 2.0$; see Section 12.2). If this output occurs frequently during later iterations of the run, **options.step_limit** should be set to a larger value.

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

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

The following describes the printout for each variable.

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

State gives the state of the variable (FR if neither bound is in the active set, EQ if a fixed variable, LL if on its lower bound, UL if on its upper bound). If Value lies outside the upper or lower bounds by more than the feasibility tolerances specified by the optional arguments **options.lin_feas_tol** and **options.nonlin_feas_tol** (see Section 12.2), State will be ++ or -- respectively.

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, 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 **options.lin_feas_tol**.

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

`Lower bound` is the lower bound specified for the variable j. (`None` indicates that $\mathbf{bl}[j-1] \leq$ **options.inf_bound**, where **options.inf_bound** is the optional argument.)

`Upper bound` is the upper bound specified for the variable j. (`None` indicates that $\mathbf{bu}[j-1] \geq$ **options.inf_bound**, where **options.inf_bound** is the optional argument.)

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

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

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

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

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

The `I` key in the `State` column is printed for general linear constraints which currently violate one of their bounds by more than **options.lin_feas_tol** and for nonlinear constraints which violate one of their bounds by more than **options.nonlin_feas_tol**.

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 `Residual` column to become positive.

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

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

On entry, **tda** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These arguments must satisfy **tda** ≥ **n**.
This error message is output only if **nclin** > 0.

NE_2_INT_OPT_ARG_CONS

On entry, **options.con_check_start** = ⟨*value*⟩ while **options.con_check_stop** = ⟨*value*⟩.
Constraint: **options.con_check_start** ≤ **options.con_check_stop**.

On entry, **options.obj_check_start** = ⟨*value*⟩ while **options.obj_check_stop** = ⟨*value*⟩.
Constraint: **options.obj_check_start** ≤ **options.obj_check_stop**.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument **options**.**minor_print_level** had an illegal value.

On entry, argument **options**.**print_deriv** had an illegal value.

On entry, argument **options**.**print_level** had an illegal value.

On entry, argument **options**.**start** had an illegal value.

On entry, argument **options**.**verify_grad** had an illegal value.

NE_BOUND

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

NE_BOUND_EQ

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

NE_BOUND_EQ_LCON

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

NE_BOUND_EQ_NLCON

The lower bound and upper bound for nonlinear constraint ⟨*value*⟩ (array elements **bl**[⟨*value*⟩] and **bu**[⟨*value*⟩]) are equal but they are greater than or equal to **options**.**inf_bound**.

NE_BOUND_LCON

The lower bound for linear constraint ⟨*value*⟩ (array element **bl**[⟨*value*⟩]) is greater than the upper bound.

NE_BOUND_NLCON

The lower bound for nonlinear constraint ⟨*value*⟩ (array element **bl**[⟨*value*⟩]) is greater than the upper bound.

NE_DERIV_ERRORS

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

As a first-step, you should check that the code for the objective and constraint values is correct – for example, by computing the function at a point where the correct value is known. However, care should be taken that the chosen point fully tests the evaluation of the function. It is remarkable how often the values $x = 0$ or $x = 1$ are used to test function evaluation procedures, and how often the special properties of these numbers make the test meaningless.

Gradient checking will be ineffective if the objective function uses information computed by the constraints, since they are not necessarily computed prior to each function evaluation.

Errors in programming the function may be quite subtle in that the function value is 'almost' correct. For example, the function may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which the function depends. A common error on machines where numerical calculations are usually performed in double precision is to include even one single precision constant in the calculation of the function; since some compilers do not convert such constants to double precision, half the correct figures may be lost by such a seemingly trivial error.

NE_INT_ARG_LT

On entry, **n** $= \langle value \rangle$.
Constraint: **n** ≥ 1.

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

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

NE_INT_OPT_ARG_GT

On entry, **options.con_check_start** $= \langle value \rangle$.
Constraint: **options.con_check_start** $\leq$ **n**.

On entry, **options.con_check_stop** $= \langle value \rangle$.
Constraint: **options.con_check_stop** $\leq$ **n**.

On entry, **options.obj_check_start** $= \langle value \rangle$.
Constraint: **options.obj_check_start** $\leq$ **n**.

On entry, **options.obj_check_stop** $= \langle value \rangle$.
Constraint: **options.obj_check_stop** $\leq$ **n**.

NE_INT_OPT_ARG_LT

On entry, **options.con_check_start** $= \langle value \rangle$.
Constraint: **options.con_check_start** ≥ 1.

On entry, **options.con_check_stop** $= \langle value \rangle$.
Constraint: **options.con_check_stop** ≥ 1.

On entry, **options.obj_check_start** $= \langle value \rangle$.
Constraint: **options.obj_check_start** ≥ 1.

On entry, **options.obj_check_stop** $= \langle value \rangle$.
Constraint: **options.obj_check_stop** ≥ 1.

NE_INVALID_INT_RANGE_1

Value $\langle value \rangle$ given to **options.max_iter** not valid. Correct range is **options.max_iter** ≥ 0.

Value $\langle value \rangle$ given to **options.minor_max_iter** not valid. Correct range is **options.minor_max_iter** ≥ 0.

NE_INVALID_REAL_RANGE_EF

Value $\langle value \rangle$ given to **options.c_diff_int** not valid. Correct range is $\epsilon \leq$ **options.c_diff_int** < 1.0.

Value $\langle value \rangle$ given to **options.f_diff_int** not valid. Correct range is $\epsilon \leq$ **options.f_diff_int** < 1.0.

Value $\langle value \rangle$ given to **options.f_prec** not valid. Correct range is $\epsilon \leq$ **options.f_prec** < 1.0.

Value $\langle value \rangle$ given to **options.lin_feas_tol** not valid. Correct range is $\epsilon \leq$ **options.lin_feas_tol** < 1.0.

Value $\langle value \rangle$ given to **options.nonlin_feas_tol** not valid. Correct range is $\epsilon \leq$ **options.nonlin_feas_tol** < 1.0.

Value $\langle value \rangle$ given to **options.optim_tol** not valid. Correct range is **options.f_prec** $\leq$ **options.optim_tol** < 1.0.

NE_INVALID_REAL_RANGE_F

Value $\langle value \rangle$ given to **options.inf_bound** not valid. Correct range is **options.inf_bound** > 0.0.

Value $\langle value \rangle$ given to **options.inf_step** not valid. Correct range is **options.inf_step** > 0.0.

Value $\langle value \rangle$ given to **options.step_limit** not valid. Correct range is **options.step_limit** > 0.0.

NE_INVALID_REAL_RANGE_FF

Value ⟨*value*⟩ given to **options.crash_tol** not valid. Correct range is $0.0 \leq$ **options.crash_tol** ≤ 1.0.

Value ⟨*value*⟩ given to **options.linesearch_tol** not valid. Correct range is $0.0 \leq$ **options.linesearch_tol** < 1.0.

NE_NOT_APPEND_FILE

Cannot open file ⟨*string*⟩ for appending.

NE_NOT_CLOSE_FILE

Cannot close file ⟨*string*⟩.

NE_OPT_NOT_INIT

Options structure not initialized.

NE_STATE_VAL

options.state[⟨*value*⟩] is out of range. **options.state**[⟨*value*⟩] = ⟨*value*⟩.

NE_USER_STOP

User requested termination, user flag value = ⟨*value*⟩.
This exit occurs if you set **comm**→**flag** to a negative value in **objfun** or **confun**. If **fail** is supplied the value of **fail.errnum** will be the same as your setting of **comm**→**flag**.

NE_WRITE_ERROR

Error occurred when writing to file ⟨*string*⟩.

NW_KT_CONDITIONS

The current point cannot be improved upon. The final point does not satisfy the first-order Kuhn–Tucker conditions and no improved point for the merit function could be found during the final line search.
The Kuhn–Tucker conditions are specified in Section 11.1, and the merit function is described in Section 11.3 and Section 12.3.
This sometimes occurs because an overly stringent accuracy has been requested, i.e., the value of the optional argument **options.optim_tol** (default value $= \epsilon_r^{0.8}$, where ϵ_r is the relative precision of $F(x)$; see Section 12.2) is too small. In this case you should apply the four tests described in Section 9.1 to determine whether or not the final solution is acceptable (see Gill *et al.* (1981) for a discussion of the attainable accuracy).
If many iterations have occurred in which essentially no progress has been made and nag_opt_nlp (e04ucc) has failed completely to move from the initial point then functions **objfun** and/or **confun** may be incorrect. You should refer to comments under **fail.code** = NE_DERIV_ERRORS and check the gradients using the optional argument **options.verify_grad** (default value **options.verify_grad** = Nag_SimpleCheck; see Section 12.2). Unfortunately, there may be small errors in the objective and constraint gradients that cannot be detected by the verification process. Finite difference approximations to first derivatives are catastrophically affected by even small inaccuracies. An indication of this situation is a dramatic alteration in the iterates if the finite difference interval is altered. One might also suspect this type of error if a switch is made to central differences even when `Norm Gz` and `Violtn` (see Section 5.1) are large.
Another possibility is that the search direction has become inaccurate because of ill conditioning in the Hessian approximation or the matrix of constraints in the working set; either form of ill conditioning tends to be reflected in large values of `Mnr` (the number of iterations required to solve each QP subproblem; see Section 5.1).
If the condition estimate of the projected Hessian (`Cond Hz`; see Section 5.1) is extremely large, it may be worthwhile rerunning nag_opt_nlp (e04ucc) from the final point with the optional argument **options.start** = Nag_Warm (see Section 12.2). In this situation, the optional arguments

options.state and **options.lambda** should be left unaltered and R (in optional argument **options.h**) should be reset to the identity matrix.
If the matrix of constraints in the working set is ill conditioned (i.e., `Cond T` is extremely large; see Section 12.3), it may be helpful to run nag_opt_nlp (e04ucc) with a relaxed value of the optional arguments **options.lin_feas_tol** and **options.nonlin_feas_tol** (default values $\sqrt{\epsilon}$, $\epsilon^{0.33}$ or $\sqrt{\epsilon}$, respectively, where ϵ is the ***machine precision***; see Section 12.2). (Constraint dependencies are often indicated by wide variations in size in the diagonal elements of the matrix T, whose diagonals will be printed if **options.print_level** = Nag_Soln_Iter_Full (default value **options.print_level** = Nag_Soln_Iter; see Section 12.2).)

NW_LIN_NOT_FEASIBLE

No feasible point was found for the linear constraints and bounds.
nag_opt_nlp (e04ucc) has terminated without finding a feasible point for the linear constraints and bounds, which means that either no feasible point exists for the given value of the optional argument **options.lin_feas_tol** (default value $= \sqrt{\epsilon}$, where ϵ is the ***machine precision***; see Section 12.2), or no feasible point could be found in the number of iterations specified by the optional argument **options.minor_max_iter** (default value $= \max(50, 3(n + n_L + n_N))$; see Section 12.2). You should check that there are no constraint redundancies. If the data for the constraints are accurate only to an absolute precision σ, you should ensure that the value of the optional argument **options.lin_feas_tol** is greater than σ. For example, if all elements of A_L are of order unity and are accurate to only three decimal places, **options.lin_feas_tol** should be at least 10^{-3}.

NW_NONLIN_NOT_FEASIBLE

No feasible point could be found for the nonlinear constraints.
The problem may have no feasible solution. This means that there has been a sequence of QP subproblems for which no feasible point could be found (indicated by `I` at the end of each terse line of output; see Section 5.1). This behaviour will occur if there is no feasible point for the nonlinear constraints. (However, there is no general test that can determine whether a feasible point exists for a set of nonlinear constraints.) If the infeasible subproblems occur from the very first major iteration, it is highly likely that no feasible point exists. If infeasibilities occur when earlier subproblems have been feasible, small constraint inconsistencies may be present. You should check the validity of constraints with negative values of the optional argument **options.state**. If you are convinced that a feasible point does exist, nag_opt_nlp (e04ucc) should be restarted at a different starting point.

NW_NOT_CONVERGED

Optimal solution found, but the sequence of iterates has not converged with the requested accuracy.
The final iterate x satisfies the first-order Kuhn–Tucker conditions (see Section 11.1) to the accuracy requested, but the sequence of iterates has not yet converged. nag_opt_nlp (e04ucc) was terminated because no further improvement could be made in the merit function (see Section 12.3).
This value of **fail.code** may occur in several circumstances. The most common situation is that you ask for a solution with accuracy that is not attainable with the given precision of the problem (as specified by the optional argument **options.f_prec** (default value $= \epsilon^{0.9}$, where ϵ is the ***machine precision***; see Section 12.2)). This condition will also occur if, by chance, an iterate is an 'exact' Kuhn–Tucker point, but the change in the variables was significant at the previous iteration. (This situation often happens when minimizing very simple functions, such as quadratics.)
If the four conditions listed in Section 9.1 are satisfied then x is likely to be a solution of (1) even if **fail.code** = NW_NOT_CONVERGED.

NW_OVERFLOW_WARN

Serious ill conditioning in the working set after adding constraint ⟨*value*⟩. Overflow may occur in subsequent iterations.

If overflow occurs preceded by this warning then serious ill conditioning has probably occurred in the working set when adding a constraint. It may be possible to avoid the difficulty by increasing the magnitude of the optional argument **options.lin_feas_tol** (default value $= \sqrt{\epsilon}$, where ϵ is the ***machine precision***; see Section 12.2) and/or the optional argument **options.nonlin_feas_tol** (default value $\epsilon^{0.33}$ or $\sqrt{\epsilon}$; see Section 12.2), and rerunning the program. If the message recurs even after this change, the offending linearly dependent constraint j must be removed from the problem. If overflow occurs in one of the user-supplied functions (e.g., if the nonlinear functions involve exponentials or singularities), it may help to specify tighter bounds for some of the variables (i.e., reduce the gap between the appropriate l_j and u_j).

NW_TOO_MANY_ITER

The maximum number of iterations, ⟨*value*⟩, have been performed.

The value of the optional argument **options.max_iter** may be too small. If the method appears to be making progress (e.g., the objective function is being satisfactorily reduced), increase the value of the optional argument **options.max_iter** and rerun nag_opt_nlp (e04ucc); alternatively, rerun nag_opt_nlp (e04ucc), setting the optional argument **options.start** = Nag_Warm to specify the initial working set. If the algorithm seems to be making little or no progress, however, then you should check for incorrect gradients or ill conditioning as described under **fail.code** = NW_KT_CONDITIONS.

Note that ill conditioning in the working set is sometimes resolved automatically by the algorithm, in which case performing additional iterations may be helpful. However, ill conditioning in the Hessian approximation tends to persist once it has begun, so that allowing additional iterations without altering R is usually inadvisable. If the quasi-Newton update of the Hessian approximation was reset during the latter iterations (i.e., an `R` occurs at the end of each line of output; see Section 5.1), it may be worthwhile setting **options.start** = Nag_Warm and calling nag_opt_nlp (e04ucc) from the final point.

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 **options.optim_tol** (default value $= \epsilon_r^{0.8}$, where ϵ_r is the relative precision of $F(x)$; see Section 12.2).

8 Parallelism and Performance

Not applicable.

9 Further Comments

9.1 Termination Criteria

The function exits with **fail.code** = NE_NOERROR if iterates have converged to a point x that satisfies the Kuhn–Tucker conditions (see Section 11.1) to the accuracy requested by the optional argument **options.optim_tol** (default value $= \epsilon_r^{0.8}$, see Section 12.2).

You should also examine the printout from nag_opt_nlp (e04ucc) (see Section 5.1 or Section 12.3) to check whether the following four conditions are satisfied:

(i) the final value of `Norm Gz` is significantly less than at the starting point;

(ii) during the final major iterations, the values of `Step` and `Mnr` are both one;

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

(iv) `Cond Hz` is small.

If all these conditions hold, x is almost certainly a local minimum.

10 Example

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

$$F(x) = x_1x_4(x_1 + x_2 + x_3) + x_3$$

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

$$\begin{array}{c} x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 40, \\ x_1x_2x_3x_4 \geq 25. \end{array}$$

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.

The **options** structure is declared and initialized by nag_opt_init (e04xxc). Two options are read from the data file by use of nag_opt_read (e04xyc). nag_opt_nlp (e04ucc) is then called to solve the problem using the function `objfun` and `confun` with elements of the objective gradient not being set at all and only some of the elements of the constraint Jacobian being provided. The memory freeing function nag_opt_free (e04xzc) is used to free the memory assigned to the pointers in the options structure. You must **not** use the standard C function `free()` for this purpose.

10.1 Program Text

```
/* nag_opt_nlp (e04ucc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 4, 1996.
 * Mark 5 revised, 1998.
 * Mark 7 revised, 2001.
 * Mark 8 revised, 2004.
 *
 */

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

#ifdef __cplusplus
extern "C" {
#endif
static void NAG_CALL objfun(Integer n, const double x[], double *objf,
                            double objgrd[], Nag_Comm *comm);
static void NAG_CALL confun(Integer n, Integer ncnlin, const Integer needc[],
                            const double x[], double conf[], double conjac[],
                            Nag_Comm *comm);
```

```
#ifdef __cplusplus
}
#endif

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

  if (comm->flag == 0 || comm->flag == 2)
    *objf = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2];

  /* Note, elements of the objective gradient have not been
     specified.
   */
} /* objfun */

static void NAG_CALL confun(Integer n, Integer ncnlin, const Integer needc[],
                            const double x[], double conf[], double conjac[],
                            Nag_Comm *comm)
{
#define CONJAC(I, J) conjac[((I) -1)*n + (J) -1]

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

  /* Function Body */
  if (needc[0] > 0)
    {
      if (comm->flag == 0 || comm->flag == 2)
        conf[0] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] + x[3] * x[3];

      if (comm->flag == 2)
        {
          CONJAC(1, 3) = x[2] * 2.0;
          /* Note only one constraint gradient has been specified
           * in the first row of the constraint Jacobian.
           */
        }
    }
  if (needc[1] > 0)
    {
      if (comm->flag == 0 || comm->flag == 2)
        conf[1] = x[0] * x[1] * x[2] * x[3];
      if (comm->flag == 2)
        {
          CONJAC(2, 2) = x[0] * x[2] * x[3];
          CONJAC(2, 3) = x[0] * x[1] * x[3];
          /* Note only two constraint gradients have been specified
           * in the second row of the constraint Jacobian.
           */
        }
    }
} /* confun */

#define A(I, J) a[(I) *tda + J]

int main(void)
{
  const char  *optionsfile = "e04ucce.opt" ;
  Integer     exit_status = 0, i, j, n, nclin, ncnlin, tda, totalvars;
  Nag_Comm    comm;
  NagError    fail;
  Nag_E04_Opt options;
  double      *a = 0, *bl = 0, *bu = 0, objf, *objgrd = 0, *x = 0;

  INIT_FAIL(fail);


  printf("nag_opt_nlp (e04ucc) Example Program Results\n");
```

```
  fflush(stdout);
#ifdef _WIN32
  scanf_s(" %*[^\n]");  /* Skip heading in data file */
#else
  scanf(" %*[^\n]");  /* Skip heading in data file */
#endif
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n]", &n, &nclin, &ncnlin);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n]", &n, &nclin, &ncnlin);
#endif
  if (n > 0 && nclin >= 0 && ncnlin >= 0)
    {
      totalvars = n + nclin + ncnlin;
      if (!(x = NAG_ALLOC(n, double)) ||
          !(a = NAG_ALLOC(nclin*n, double)) ||
          !(bl = NAG_ALLOC(totalvars, double)) ||
          !(bu = NAG_ALLOC(totalvars, double)) ||
          !(objgrd = NAG_ALLOC(n, double)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
      tda = n;
    }
  else
    {
      printf("Invalid n or nclin or ncnlin.\n");
      exit_status = 1;
      return exit_status;
    }
  /* Read a, bl, bu and x from data file */

  /* Read the matrix of linear constraint coefficients */
  if (nclin > 0)
    {
      for (i = 0; i < nclin; ++i)
        for (j = 0; j < n; ++j)
#ifdef _WIN32
          scanf_s("%lf", &A(i, j));
#else
          scanf("%lf", &A(i, j));
#endif
    }
#ifdef _WIN32
  scanf_s("%*[^\n]"); /* Remove remainder of line */
#else
  scanf("%*[^\n]"); /* Remove remainder of line */
#endif

  /* Read lower bounds */
  for (i = 0; i < n + nclin + ncnlin; ++i)
#ifdef _WIN32
    scanf_s("%lf", &bl[i]);
#else
    scanf("%lf", &bl[i]);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif

  /* Read upper bounds */
  for (i = 0; i < n + nclin + ncnlin; ++i)
#ifdef _WIN32
    scanf_s("%lf", &bu[i]);
#else
    scanf("%lf", &bu[i]);
#endif
```

```
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif

  /* Read the initial point x */
  for (i = 0; i < n; ++i)
#ifdef _WIN32
    scanf_s("%lf", &x[i]);
#else
    scanf("%lf", &x[i]);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif

  /* nag_opt_init (e04xxc).
   * Initialization function for option setting
   */
  nag_opt_init(&options);
  /* nag_opt_read (e04xyc).
   * Read options from a text file
   */
  nag_opt_read("e04ucc", optionsfile, &options, (Nag_Boolean) Nag_TRUE,
               "stdout", &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_opt_read (e04xyc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* nag_opt_nlp (e04ucc), see above. */
  nag_opt_nlp(n, nclin, ncnlin, a, tda, bl, bu, objfun, confun, x, &objf,
              objgrd, &options, &comm, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_opt_nlp (e04ucc).\n%s\n", fail.message);
      exit_status = 1;
    }

  /* nag_opt_free (e04xzc).
   * Memory freeing function for use with option setting
   */
  nag_opt_free(&options, "all", &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_opt_free (e04xzc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

 END:
  NAG_FREE(x);
  NAG_FREE(a);
  NAG_FREE(bl);
  NAG_FREE(bu);
  NAG_FREE(objgrd);

  return exit_status;
}
```

10.2 Program Data

```
nag_opt_nlp (e04ucc) Example Program Data
  4   1   2                                 :Values of n, nclin and ncnlin
  1.0   1.0   1.0   1.0                                   :End of matrix A
  1.0   1.0   1.0   1.0  -1.0E+25  -1.0E+25  25.0       :End of bl
  5.0   5.0   5.0   5.0  20.0      40.0       1.0E+25   :End of bu
  1.0   5.0   5.0   1.0                                   :End of x
```

```
nag_opt_nlp (e04ucc) Example Program Optional Parameters

Begin e04ucc
con_deriv = Nag_FALSE
obj_deriv = Nag_FALSE
End
```

10.3 Program Results

```
nag_opt_nlp (e04ucc) Example Program Results

Optional parameter setting for e04ucc.
--------------------------------------

Option file: e04ucce.opt

con_deriv set to Nag_FALSE
obj_deriv set to Nag_FALSE

Parameters to e04ucc
--------------------

Number of variables...........   4

Linear constraints............   1    Nonlinear constraints.........   2
start...................  Nag_Cold
step_limit..............  2.00e+00    machine precision.......  1.11e-16
lin_feas_tol............  1.05e-08    nonlin_feas_tol.........  1.05e-08
optim_tol...............  3.26e-12    linesearch_tol..........  9.00e-01
crash_tol...............  1.00e-02    f_prec..................  4.37e-15
inf_bound...............  1.00e+20    inf_step................  1.00e+20
max_iter................        50    minor_max_iter..........        50
hessian................. Nag_FALSE
f_diff_int.............. Automatic    c_diff_int.............. Automatic
obj_deriv............... Nag_FALSE    con_deriv............... Nag_FALSE
verify_grad....... Nag_SimpleCheck    print_deriv............ Nag_D_Full
print_level......... Nag_Soln_Iter    minor_print_level..... Nag_NoPrint
outfile.................    stdout


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

The user sets 0 out of 4 objective gradient elements.
Each iteration 4 gradient elements will be estimated numerically.


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

The user sets 3 out of 8 constraint gradient elements.
Each iteration, 5 gradient elements will be estimated numerically.

Simple Check:

The Jacobian seems to be ok.

The largest relative error was 5.22e-09 in constraint 1


Finite difference intervals.
----------------------------
```

```
    j      x[j]       Forward dx[j]    Central dx[j]    Error Est.
    1   1.00e+00      6.479651e-07     2.645420e-06     2.592083e-06
    2   5.00e+00      7.825142e-07     7.936259e-06     1.565074e-06
    3   5.00e+00      7.936259e-06     7.936259e-05     1.873839e-08
    4   1.00e+00      9.163610e-07     2.645420e-06     1.832879e-06

   Maj  Mnr    Step   Merit function  Violtn  Norm Gz  Cond Hz
    0    5  0.0e+00    1.738281e+01  1.2e+01  7.1e-01  1.0e+00
    1    1  1.0e+00    1.703169e+01  1.9e+00  4.6e-02  1.0e+00
    2    1  1.0e+00    1.701442e+01  8.8e-02  2.1e-02  1.0e+00
    3    1  1.0e+00    1.701402e+01  5.4e-04  3.1e-04  1.0e+00
    4    1  1.0e+00    1.701402e+01  9.9e-08  7.0e-06  1.0e+00
Minor itn 1 -- Re-solve QP subproblem.
    5    2  1.0e+00    1.701402e+01  4.7e-11  6.2e-08  1.0e+00    C
Exit from NP problem after 5 major iterations, 11 minor iterations.


Varbl State     Value      Lower Bound   Upper Bound    Lagr Mult    Residual
V   1    LL   1.00000e+00   1.00000e+00   5.00000e+00   1.0879e+00   0.0000e+00
V   2    FR   4.74300e+00   1.00000e+00   5.00000e+00   0.0000e+00   2.5700e-01
V   3    FR   3.82115e+00   1.00000e+00   5.00000e+00   0.0000e+00   1.1789e+00
V   4    FR   1.37941e+00   1.00000e+00   5.00000e+00   0.0000e+00   3.7941e-01


L Con State     Value      Lower Bound   Upper Bound    Lagr Mult    Residual
L   1    FR   1.09436e+01      None       2.00000e+01   0.0000e+00   9.0564e+00


N Con State     Value      Lower Bound   Upper Bound    Lagr Mult    Residual
N   1    UL   4.00000e+01      None       4.00000e+01  -1.6147e-01  -3.7126e-11
N   2    LL   2.50000e+01   2.50000e+01      None       5.5229e-01  -2.8749e-11

Optimal solution found.

Final objective value =    1.7014017e+01
```

11 Further Description

This section gives a detailed description of the algorithm used in nag_opt_nlp (e04ucc). This, and possibly the next section, Section 12, may be omitted if the more sophisticated features of the algorithm and software are not currently of interest.

11.1 Overview

nag_opt_nlp (e04ucc) is based on the same algorithm as used in subroutine NPSOL described in Gill *et al.* (1986b).

At a solution of (1), some of the constraints will be *active*, i.e., satisfied exactly. An active simple bound constraint implies that the corresponding variable is *fixed* at its bound, and hence the variables are partitioned into *fixed* and *free* variables. Let C denote the m by n matrix of gradients of the active general linear and nonlinear constraints. The number of fixed variables will be denoted by n_{FX}, with n_{FR} ($n_{\text{FR}} = n - n_{\text{FX}}$) the number of free variables. The subscripts 'FX' and 'FR' on a vector or matrix will denote the vector or matrix composed of the elements corresponding to fixed or free variables.

A point x is a *first-order Kuhn–Tucker point* for (1) (see, e.g., Powell (1974)) if the following conditions hold:

(i) x is feasible;

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

$$g = C^{\mathrm{T}}\lambda + \xi \qquad (2)$$

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

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

Let Z denote a matrix whose columns form a basis for the set of vectors orthogonal to the rows of C_{FR}; i.e., $C_{\mathrm{FR}}Z = 0$. An equivalent statement of the condition (2) in terms of Z is

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

The vector $Z^{\mathrm{T}}g_{\mathrm{FR}}$ is termed the *projected gradient* of F at x. Certain additional conditions must be satisfied in order for a first-order Kuhn–Tucker point to be a solution of (1) (see, e.g., Powell (1974)). nag_opt_nlp (e04ucc) implements a sequential quadratic programming (SQP) method. For an overview of SQP methods, see, for example, Fletcher (1987), Gill *et al.* (1981) and Powell (1983).

The basic structure of nag_opt_nlp (e04ucc) involves *major* and *minor* iterations. The major iterations generate a sequence of iterates $\{x_k\}$ that converge to x^*, a first-order Kuhn–Tucker point of (1). At a typical major iteration, the new iterate $\bar{x}$ is defined by

$$\bar{x} = x + \alpha p \tag{3}$$

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

The search direction p in (3) is the solution of a quadratic programming subproblem of the form

$$\underset{p}{\text{Minimize}} \quad g^{\mathrm{T}}p + \frac{1}{2}p^{\mathrm{T}}Hp \quad \text{subject to} \quad \bar{l} \le \left\{ \begin{array}{c} p \\ A_L p \\ A_N p \end{array} \right\} \le \bar{u}, \tag{4}$$

where g is the gradient of F at x, the matrix H is a positive definite quasi-Newton approximation to the Hessian of the Lagrangian function (see Section 11.4), and A_N is the Jacobian matrix of c evaluated at x. (Finite difference estimates may be used for g and A_N; see the optional arguments **options.obj_deriv** and **options.con_deriv** in Section 12.2.) Let l in (1) be partitioned into three sections: l_B, l_L and l_N, corresponding to the bound, linear and nonlinear constraints. The vector $\bar{l}$ in (4) is similarly partitioned, and is defined as

$$\bar{l}_B = l_B - x, \bar{l}_L = l_L - A_L x, \text{ and } \bar{l}_N = l_N - c,$$

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

The estimated Lagrange multipliers at each major iteration are the Lagrange multipliers from the subproblem (4) (and similarly for the predicted active set). (The numbers of bounds, general linear and nonlinear constraints in the QP active set are the quantities `Bnd`, `Lin` and `Nln` in the output of nag_opt_nlp (e04ucc); see Section 12.3.) In nag_opt_nlp (e04ucc), (4) is solved using the same algorithm as used in function nag_opt_lin_lsq (e04ncc). Since solving a quadratic program is an iterative procedure, the minor iterations of nag_opt_nlp (e04ucc) are the iterations of nag_opt_lin_lsq (e04ncc). (More details about solving the subproblem are given in Section 11.2.)

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

$$C_{\mathrm{FR}}Q_{\mathrm{FR}} = \begin{pmatrix} 0 & T \end{pmatrix}, \tag{5}$$

where T is a nonsingular m by m reverse-triangular matrix (i.e., $t_{ij} = 0$ if $i + j < m$, and the nonsingular n_{FR} by n_{FR} matrix Q_{FR} is the product of orthogonal transformations (see Gill *et al.* (1984a)). Second, we have the upper triangular Cholesky factor R of the *transformed and re-ordered* Hessian matrix

$$R^{\mathrm{T}}R = H_Q \equiv Q^{\mathrm{T}}\tilde{H}Q, \tag{6}$$

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

$$Q = \begin{pmatrix} Q_{\mathrm{FR}} & \\ & I_{\mathrm{FX}} \end{pmatrix} \tag{7}$$

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

$$Q_{\mathrm{FR}} = \begin{pmatrix} Z & Y \end{pmatrix},$$

the $n_Z \quad (n_Z \equiv n_{\mathrm{FR}} - m)$ columns of Z form a basis for the null space of C_{FR}. The matrix Z is used to compute the projected gradient $Z^{\mathrm{T}} g_{\mathrm{FR}}$ at the current iterate. (The values `Nz`, `Norm Gf` and `Norm Gz` printed by nag_opt_nlp (e04ucc) give n_Z and the norms of g_{FR} and $Z^{\mathrm{T}} g_{\mathrm{FR}}$; see Section 12.3.)

A theoretical characteristic of SQP methods is that the predicted active set from the QP subproblem (4) is identical to the correct active set in a neighbourhood of x^*. In nag_opt_nlp (e04ucc), this feature is exploited by using the QP active set from the previous iteration as a prediction of the active set for the next QP subproblem, which leads in practice to optimality of the subproblems in only one iteration as the solution is approached. Separate treatment of bound and linear constraints in nag_opt_nlp (e04ucc) also saves computation in factorizing C_{FR} and H_Q.

Once p has been computed, the major iteration proceeds by determining a step length α that produces a 'sufficient decrease' in an augmented Lagrangian *merit function* (see Section 11.3). Finally, the approximation to the transformed Hessian matrix H_Q is updated using a modified BFGS quasi-Newton update (see Section 11.4) to incorporate new curvature information obtained in the move from x to $\bar{x}$.

On entry to nag_opt_nlp (e04ucc), an iterative procedure from nag_opt_lin_lsq (e04ncc) is executed, starting with the initial point you provided, to find a point that is feasible with respect to the bounds and linear constraints (using the tolerance specified by **options.lin_feas_tol**; see Section 12.2). If no feasible point exists for the bound and linear constraints, (1) has no solution and nag_opt_nlp (e04ucc) terminates. Otherwise, the problem functions will thereafter be evaluated only at points that are feasible with respect to the bounds and linear constraints. The only exception involves variables whose bounds differ by an amount comparable to the finite difference interval (see the discussion of **options.f_diff_int** in Section 12.2). In contrast to the bounds and linear constraints, it must be emphasized that *the nonlinear constraints will not generally be satisfied until an optimal point* is reached.

Facilities are provided to check whether the gradients you provided appear to be correct (see the optional argument **options.verify_grad** in Section 12.2). In general, the check is provided at the first point that is feasible with respect to the linear constraints and bounds. However, you may request that the check be performed at the initial point.

In summary, the method of nag_opt_nlp (e04ucc) first determines a point that satisfies the bound and linear constraints. Thereafter, each iteration includes:

(a) the solution of a quadratic programming subproblem (see Section 11.2);

(b) a linesearch with an augmented Lagrangian merit function (see Section 11.3); and

(c) a quasi-Newton update of the approximate Hessian of the Lagrangian function (Section 11.4).

11.2 Solution of the Quadratic Programming Subproblem

The search direction p is obtained by solving (4) using the algorithm of nag_opt_lin_lsq (e04ncc) (see Gill *et al.* (1986)), which was specifically designed to be used within an SQP algorithm for nonlinear programming.

The method of nag_opt_lin_lsq (e04ncc) is a two-phase (primal) quadratic programming method. The two phases of the method are: finding an initial feasible point by minimizing the sum of infeasibilities (the *feasibility phase*), and minimizing the quadratic objective function within the feasible region (the *optimality phase*). The computations in both phases are performed by the same segments of code. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function.

In general, a quadratic program must be solved by iteration. Let p denote the current estimate of the solution of (4); the new iterate $\bar{p}$ is defined by

$$\bar{p} = p + \sigma d \tag{8}$$

where, as in (3), σ is a non-negative step length and d is a search direction.

At the beginning of each iteration of nag_opt_lin_lsq (e04ncc), a *working* set is defined of constraints (general and bound) that are satisfied exactly. The vector d is then constructed so that the values of constraints in the working set remain *unaltered* for any move along d. For a bound constraint in the working set, this property is achieved by setting the corresponding element of d to zero, i.e., by fixing the variable at its bound. As before, the subscripts 'FX' and 'FR' denote selection of the elements associated with the fixed and free variables.

Let C denote the sub-matrix of rows of

$$\begin{pmatrix} A_L \\ A_N \end{pmatrix}$$

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

$$C_{\mathrm{FR}} d_{\mathrm{FR}} = 0 \tag{9}$$

which is equivalent to defining d_{FR} as

$$d_{\mathrm{FR}} = Z d_Z \tag{10}$$

for some vector d_Z, where Z is the matrix associated with the TQ factorization (5) of C_{FR}.

The definition of d_Z in (10) depends on whether the current p is feasible. If not, d_Z is zero except for an element γ in the jth position, where j and γ are chosen so that the sum of infeasibilities is decreasing along d. (For further details, see Gill *et al.* (1986).) In the feasible case, d_Z satisfies the equations

$$R_Z^{\mathrm{T}} R_Z d_Z = -Z^{\mathrm{T}} q_{\mathrm{FR}} \tag{11}$$

where R_Z is the Cholesky factor of $Z^{\mathrm{T}} H_{\mathrm{FR}} Z$ and q is the gradient of the quadratic objective function $(q = g + Hp)$. (The vector $Z^{\mathrm{T}} q_{\mathrm{FR}}$ is the projected gradient of the QP.) With (11), $p + d$ is the minimizer of the quadratic objective function subject to treating the constraints in the working set as equalities.

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

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

The choice of step length σ in the QP iteration (8) is based on remaining feasible with respect to the satisfied constraints. During the optimality phase, if $p + d$ is feasible, σ will be taken as unity. (In this case, the projected gradient at $\bar{p}$ will be zero.) Otherwise, σ is set to σ_M, the step to the 'nearest' constraint, which is added to the working set at the next iteration.

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

11.3 The Merit Function

After computing the search direction as described in Section 11.2, each major iteration proceeds by determining a step length α in (3) that produces a 'sufficient decrease' in the augmented Lagrangian merit function

$$L(x, \lambda, s) = F(x) - \sum_i \lambda_i (c_i(x) - s_i) + \frac{1}{2} \sum_i \rho_i (c_i(x) - s_i)^2, \qquad (12)$$

where x, λ and s vary during the *linesearch*. The summation terms in (12) involve only the *nonlinear* constraints. The vector λ is an estimate of the Lagrange multipliers for the nonlinear constraints of (1). The non-negative *slack variables* $\{s_i\}$ allow nonlinear inequality constraints to be treated without introducing discontinuities. The solution of the QP subproblem (4) provides a vector triple that serves as a direction of search for the three sets of variables. The non-negative vector ρ of *penalty arguments* is initialized to zero at the beginning of the first major iteration. Thereafter, selected elements are increased whenever necessary to ensure descent for the merit function. Thus, the sequence of norms of ρ (the printed quantity `Penalty`; see Section 12.3) is generally nondecreasing, although each ρ_i may be reduced a limited number of times.

The merit function (12) and its global convergence properties are described in Gill *et al.* (1986a).

11.4 The Quasi–Newton Update

The matrix H in (4) is a *positive definite quasi-Newton* approximation to the Hessian of the Lagrangian function. (For a review of quasi-Newton methods, see Dennis and Schnabel (1983).) At the end of each major iteration, a new Hessian approximation $\bar{H}$ is defined as a rank-two modification of H. In nag_opt_nlp (e04ucc), the BFGS quasi-Newton update is used:

$$\bar{H} = H - \frac{1}{s^{\mathrm{T}} H s} H s s^{\mathrm{T}} H + \frac{1}{y^{\mathrm{T}} s} y y^{\mathrm{T}}, \qquad (13)$$

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

In nag_opt_nlp (e04ucc), H is required to be positive definite. If H is positive definite, $\bar{H}$ defined by (13) will be positive definite if and only if $y^{\mathrm{T}} s$ is positive (see, e.g., Dennis and Moré (1977)). Ideally, y in (13) would be taken as y_L, the change in gradient of the Lagrangian function

$$y_L = \bar{g} - \bar{A}_N^{\mathrm{T}} \mu_N - g + A_N^{\mathrm{T}} \mu_N \qquad (14)$$

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

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

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

Rather than modifying H itself, the Cholesky factor of the *transformed Hessian* H_Q (6) is updated, where Q is the matrix from (5) associated with the active set of the QP subproblem. The update (12) is equivalent to the following update to H_Q:

$$\bar{H}_Q = H_Q - \frac{1}{s_Q^{\mathrm{T}} H_Q s_Q} H_Q s_Q s_Q^{\mathrm{T}} H_Q + \frac{1}{y_Q^{\mathrm{T}} s_Q} y_Q y_Q^{\mathrm{T}}, \qquad (15)$$

where $y_Q = Q^{\mathrm{T}} y$, and $s_Q = Q^{\mathrm{T}} s$. This update may be expressed as a *rank-one* update to R (see Dennis and Schnabel (1981)).

12 Optional Arguments

A number of optional input and output arguments to nag_opt_nlp (e04ucc) are available through the structure argument **options**, type Nag_E04_Opt. An argument may be selected by assigning an appropriate value to the relevant structure member; those arguments not selected will be assigned default values. If no use is to be made of any of the optional arguments you should use the NAG defined null pointer, `E04_DEFAULT`, in place of **options** when calling nag_opt_nlp (e04ucc); the default settings will then be used for all arguments.

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

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

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

12.1 Optional Argument Checklist and Default Values

For easy reference, the following list shows the members of **options** which are valid for nag_opt_nlp (e04ucc) together with their default values where relevant. The number ϵ is a generic notation for ***machine precision*** (see nag_machine_precision (X02AJC)).

`Nag_Start start`	Nag_Cold
`Boolean list`	Nag_TRUE
`Nag_PrintType print_level`	Nag_Soln_Iter
`Nag_PrintType minor_print_level`	Nag_NoPrint
`char outfile[80]`	`stdout`
`void (*print_fun)()`	**NULL**
`Boolean obj_deriv`	Nag_TRUE
`Boolean con_deriv`	Nag_TRUE
`Nag_GradChk verify_grad`	Nag_SimpleCheck
`Nag_DPrintType print_deriv`	Nag_D_Full
`Integer obj_check_start`	1
`Integer obj_check_stop`	**n**
`Integer con_check_start`	1
`Integer con_check_stop`	**n**
`double f_diff_int`	Computed automatically
`double c_diff_int`	Computed automatically
`Integer max_iter`	$\max(50, 3(\mathbf{n} + \mathbf{nclin}) + 10\mathbf{ncnlin})$
`Integer minor_max_iter`	$\max(50, 3(\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}))$
`double f_prec`	$\epsilon^{0.9}$
`double optim_tol`	$\mathbf{options.f\_prec}^{0.8}$
`double lin_feas_tol`	$\sqrt{\epsilon}$
`double nonlin_feas_tol`	$\epsilon^{0.33}$ or $\sqrt{\epsilon}$
`double linesearch_tol`	0.9
`double step_limit`	2.0
`double crash_tol`	0.01
`double inf_bound`	10^{20}
`double inf_step`	$\max\left(\mathbf{options.inf\_bound}, 10^{20}\right)$
`double *conf`	size **ncnlin**
`double *conjac`	size **ncnlin*n**
`Integer *state`	size $\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}$
`double *lambda`	size $\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}$
`double *h`	size **n*n**
`Boolean hessian`	Nag_FALSE
`Integer iter`	
`Integer nf`	

12.2 Description of the Optional Arguments

start – Nag_Start Default = Nag_Cold

On entry: specifies how the initial working set is chosen in both the procedure for finding a feasible point for the linear constraints and bounds, and in the first QP subproblem thereafter. With **options**.**start** = Nag_Cold, nag_opt_nlp (e04ucc) chooses the initial working set based on the values of the variables and constraints at the initial point. Broadly speaking, the initial working set will include equality constraints and bounds or inequality constraints that violate or 'nearly' satisfy their bounds (to within the value of optional argument **options**.**crash_tol**; see below).

With **options**.**start** = Nag_Warm, you must provide a valid definition of every array element of the optional arguments **options**.**state**, **options**.**lambda** and **options**.**h** (see below for their definitions). The **options**.**state** 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. The **options**.**state** 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 (e04ucc) will override your specification of **options**.**state** if necessary, so that a poor choice of the working set will not cause a fatal error. For instance, any elements of **options**.**state** 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 (e04ucc) is called repeatedly to solve related problems.

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

list – Nag_Boolean Default = Nag_TRUE

On entry: if **options**.**list** = Nag_TRUE the argument settings in the call to nag_opt_nlp (e04ucc) will be printed.

print_level – Nag_PrintType Default = $Nag_{S}oln_{I}ter$

On entry: the level of results printout produced by nag_opt_nlp (e04ucc) at each major iteration. The following values are available:

Nag_NoPrint	No output.
Nag_Soln	The final solution only.
Nag_Iter	One line of output for each iteration.
Nag_Iter_Long	A longer line of output for each iteration with more information (line exceeds 80 characters).
Nag_Soln_Iter	The final solution and one line of output for each iteration.
Nag_Soln_Iter_Long	The final solution and one long line of output for each iteration (line exceeds 80 characters).
Nag_Soln_Iter_Const	As Nag_Soln_Iter_Long with the objective function, the values of the variables, the Euclidean norm of the nonlinear constraint violations, the nonlinear constraint values, c, and the linear constraint values A_Lx also printed at each iteration.
Nag_Soln_Iter_Full	As Nag_Soln_Iter_Const with the diagonal elements of the upper triangular matrix T associated with the TQ factorization (5) of the QP working set, and the diagonal elements of R, the triangular factor of the transformed and re-ordered Hessian (6).

Details of each level of results printout are described in Section 12.3.

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

minor_print_level – Nag_PrintType Default = Nag_NoPrint

On entry: the level of results printout produced by the minor iterations of nag_opt_nlp (e04ucc) (i.e., the iterations of the QP subproblem). The following values are available:

Nag_NoPrint	No output.
Nag_Soln	The final solution only.
Nag_Iter	One line of output for each iteration.
Nag_Iter_Long	A longer line of output for each iteration with more information (line exceeds 80 characters).
Nag_Soln_Iter	The final solution and one line of output for each iteration.
Nag_Soln_Iter_Long	The final solution and one long line of output for each iteration (line exceeds 80 characters).
Nag_Soln_Iter_Const	As Nag_Soln_Iter_Long with the Lagrange multipliers, the variables x, the constraint values $A_L x$ and the constraint status also printed at each iteration.
Nag_Soln_Iter_Full	As Nag_Soln_Iter_Const with the diagonal elements of the upper triangular matrix T associated with the TQ factorization (4) of the working set, and the diagonal elements of the upper triangular matrix R printed at each iteration.

Details of each level of results printout are described in Section 12 in nag_opt_lin_lsq (e04ncc). (**options.minor_print_level** in the present function is equivalent to **options.print_level**.)

Constraint: **options.minor_print_level** = Nag_NoPrint, Nag_Soln, Nag_Iter, Nag_Soln_Iter, Nag_Iter_Long, Nag_Soln_Iter_Long, Nag_Soln_Iter_Const or Nag_Soln_Iter_Full.

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

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

print_fun – pointer to function Default = **NULL**

On entry: printing function defined by you; the prototype of **options.print_fun** is

```
void (*print_fun)(const Nag_Search_State *st, Nag_Comm *comm);
```

See Section 12.3.1 for further details.

obj_deriv – Nag_Boolean Default = Nag_TRUE

On entry: this argument indicates whether you have provided all the derivatives of the objective function in **objfun**. If none or only some of the derivatives are being supplied by **objfun** then **options.obj_deriv** should be set to Nag_FALSE.

Whenever possible you should supply all derivatives, since nag_opt_nlp (e04ucc) is more reliable and will usually be more efficient when all derivatives are exact.

If **options.obj_deriv** = Nag_FALSE, nag_opt_nlp (e04ucc) will approximate the unspecified components of the objective gradient, using finite differences. The computation of finite difference approximations usually increases the total run-time, since a call to **objfun** is required for each unspecified element. Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill *et al.* (1986b), for a discussion of limiting accuracy).

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

con_deriv – Nag_Boolean Default = Nag_TRUE

On entry: this argument indicates whether you have provided all derivatives for the constraint Jacobian in **confun**. If none or only some of the derivatives are being supplied by **confun** then **options.con_deriv** should be set to Nag_FALSE.

Whenever possible you should supply all derivatives, since nag_opt_nlp (e04ucc) is more reliable and will usually be more efficient when all derivatives are exact.

If **options.con_deriv** = Nag_FALSE, nag_opt_nlp (e04ucc) will approximate unspecified elements of the Jacobian. One call to **confun** is needed for each variable for which partial derivatives are not available.

For example, if the constraint Jacobian has the form

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

where ' $*$ ' indicates a provided element and '?' indicates an unspecified element, nag_opt_nlp (e04ucc) will call **confun** twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1 and 4 are known, they require no calls to **confun**.)

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

verify_grad – Nag_GradChk Default = Nag_SimpleCheck

On entry: specifies the level of derivative checking to be performed by nag_opt_nlp (e04ucc) on the gradient elements computed by **objfun** and **confun**.

The following values are available:

Nag_NoCheck	No derivative checking is performed.
Nag_SimpleCheck	Perform a simple check of both the objective and constraint gradients.
Nag_CheckObj	Perform a component check of the objective gradient elements.
Nag_CheckCon	Perform a component check of the constraint gradient elements.
Nag_CheckObjCon	Perform a component check of both the objective and constraint gradient elements.
Nag_XSimpleCheck	Perform a simple check of both the objective and constraint gradients at the initial value of x specified in **x**.
Nag_XCheckObj	Perform a component check of the objective gradient elements at the initial value of x specified in **x**.
Nag_XCheckCon	Perform a component check of the constraint gradient elements at the initial value of x specified in **x**.
Nag_XCheckObjCon	Perform a component check of both the objective and constraint gradient elements at the initial value of x specified in **x**.

If **options.verify_grad** = Nag_SimpleCheck or Nag_XSimpleCheck then a simple 'cheap' test is performed, which requires only one call to **objfun** and one call to **confun**. If **options.verify_grad** = Nag_CheckObj, Nag_CheckCon or Nag_CheckObjCon then a more reliable (but more expensive) test will be made on individual gradient components. This component check will be made in the range specified by the optional arguments **options.obj_check_start** and **options.obj_check_stop** for the objective gradient, with default values being 1 and **n** respectively. For the constraint gradient the range is specified by **options.con_check_start** and **options.con_check_stop**, with default values being 1 and **n**.

The procedure for the derivative check is based on finding an interval that produces an acceptable estimate of the second derivative, and then using that estimate to compute an interval that should produce a reasonable forward-difference approximation. The gradient element is then compared with the difference approximation. (The method of finite difference interval estimation is based on Gill *et al.* (1983).) The result of the test is printed out by nag_opt_nlp (e04ucc) if optional argument **options.print_deriv** $\neq$ Nag_D_NoPrint.

Constraint: **options.verify_grad** = Nag_NoCheck, Nag_SimpleCheck, Nag_CheckObj, Nag_CheckCon, Nag_CheckObjCon, Nag_XSimpleCheck, Nag_XCheckObj, Nag_XCheckCon or Nag_XCheckObjCon.

print_deriv – Nag_DPrintType Default $= Nag_D Full$

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

If a component derivative check has been carried out, then full details will be printed if **options.print_deriv** = Nag_D_Full. For a printout summarising the results of a component derivative check set **options.print_deriv** = Nag_D_Sum. If only a simple derivative check is requested then Nag_D_Sum and Nag_D_Full will give the same level of output. To prevent any printout from a derivative check set **options.print_deriv** = Nag_D_NoPrint.

Constraint: **options.print_deriv** = Nag_D_NoPrint, Nag_D_Sum or Nag_D_Full.

obj_check_start – Integer Default = 1
obj_check_stop – Integer Default = **n**

These options take effect only when **options.verify_grad** = Nag_CheckObj, Nag_CheckObjCon, Nag_XCheckObj or Nag_XCheckObjCon.

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

Constraint: 1 ≤ **options.obj_check_start** ≤ **options.obj_check_stop** ≤ **n**.

con_check_start – Integer Default = 1
con_check_stop – Integer Default = **n**

These options take effect only when **options.verify_grad** = Nag_CheckCon, Nag_CheckObjCon, Nag_XCheckCon or Nag_XCheckObjCon.

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

Constraint: 1 ≤ **options.con_check_start** ≤ **options.con_check_stop** ≤ **n**.

f_diff_int – double Default = computed automatically

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

(a) For verifying the objective and/or constraint gradients (see the description of the optional argument **options.verify_grad**).

(b) For estimating unspecified elements of the objective and/or constraint Jacobian matrix.

In general, using the notation $r =$ **options.f_diff_int**, a derivative with respect to the jth variable is approximated using the interval δ_j, where $\delta_j = r\left(1 + \left|\hat{x}_j\right|\right)$, with $\hat{x}$ the first point feasible with respect to the bounds and linear constraints. If the functions are well scaled, the resulting derivative approximation should be accurate to $O(r)$. See Gill *et al.* (1981) for a discussion of the accuracy in finite difference approximations.

If you do not specify a difference interval, a finite difference interval will be computed automatically for each variable by a procedure that requires up to six calls of **confun** and **objfun** for each element. This option is recommended if the function is badly scaled or you wish to have nag_opt_nlp (e04ucc) determine constant elements in the objective and constraint gradients (see the descriptions of **confun** and **objfun** in Section 5).

Constraint: $\epsilon \leq$ **options.f_diff_int** < 1.0.

c_diff_int – double Default = computed automatically

On entry: if the algorithm switches to central differences because the forward-difference approximation is not sufficiently accurate the value of **options.c_diff_int** is used as the difference interval for every element of x. The switch to central differences is indicated by `C` at the end of each line of intermediate printout produced by the major iterations (see Section 5.1). The use of finite differences is discussed under the option **options.f_diff_int**.

Constraint: $\epsilon \leq$ **options.c_diff_int** < 1.0.

max_iter – Integer Default $= \max(50, 3(\mathbf{n} + \mathbf{nclin}) + 10\mathbf{ncnlin})$

On entry: the maximum number of major iterations allowed before termination.

Constraint: **options.max_iter** ≥ 0.

minor_max_iter – Integer Default $= \max(50, 3(\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}))$

On entry: the maximum number of iterations for finding a feasible point with respect to the bounds and linear constraints (if any). The value also specifies the maximum number of minor iterations for the optimality phase of each QP subproblem.

Constraint: **options.minor_max_iter** ≥ 0.

f_prec – double Default $= \epsilon^{0.9}$

On entry: this argument defines ϵ_r, which is intended to be a measure of the accuracy with which the problem functions $F(x)$ and $c(x)$ can be computed.

The value of ϵ_r should reflect the relative precision of $1 + |F(x)|$; i.e., ϵ_r acts as a relative precision when $|F|$ is large, and as an absolute precision when $|F|$ is small. For example, if $F(x)$ is typically of order 1000 and the first six significant digits are known to be correct, an appropriate value for ϵ_r would be 10^{-6}. In contrast, if $F(x)$ is typically of order 10^{-4} and the first six significant digits are known to be correct, an appropriate value for ϵ_r would be 10^{-10}. The choice of ϵ_r can be quite complicated for badly scaled problems; see Chapter 8 of Gill *et al.* (1981), for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However, when the accuracy of the computed function values is known to be significantly worse than full precision, the value of ϵ_r should be large enough so that nag_opt_nlp (e04ucc) will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

Constraint: $\epsilon \leq$ **options.f_prec** < 1.0.

optim_tol – double Default $=$ **options.f_prec**$^{0.8}$

On entry: specifies the accuracy to which you wish the final iterate to approximate a solution of the problem. Broadly speaking, **options.optim_tol** indicates the number of correct figures desired in the objective function at the solution. For example, if **options.optim_tol** is 10^{-6} and nag_opt_nlp (e04ucc) terminates successfully, the final value of F should have approximately six correct figures.

nag_opt_nlp (e04ucc) will terminate successfully if the iterative sequence of x-values is judged to have converged and the final point satisfies the first-order Kuhn–Tucker conditions (see Section 11.1). The sequence of iterates is considered to have converged at x if

$$\alpha\|p\| \leq \sqrt{r}(1 + \|x\|), \tag{16}$$

where p is the search direction and α the step length from (3), and r is the value of **options.optim_tol**. An iterate is considered to satisfy the first-order conditions for a minimum if

$$\left\|Z^{\mathrm{T}} g_{\mathrm{FR}}\right\| \leq \sqrt{r}(1 + \max(1 + |F(x)|, \|g_{\mathrm{FR}}\|)) \tag{17}$$

and

$$\left|res_j\right| \leq ftol \text{ for all } j, \tag{18}$$

where $Z^{\mathrm{T}}{}_{\mathrm{FR}} g_{\mathrm{FR}}$ is the projected gradient (see Section 11.1), g_{FR} is the gradient of $F(x)$ with respect to the free variables, res_j is the violation of the jth active nonlinear constraint, and $ftol$ the value of the optional argument **options.nonlin_feas_tol**.

Constraint: **options.f_prec** $\leq$ **options.optim_tol** < 1.0.

lin_feas_tol – double Default $= \sqrt{\epsilon}$

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

On entry to nag_opt_nlp (e04ucc), an iterative procedure is executed in order to find a point that satisfies the linear constraints and bounds on the variables to within the tolerance specified by **options.lin_feas_tol**. All subsequent iterates will satisfy the constraints to within the same tolerance (unless **options.lin_feas_tol** is comparable to the finite difference interval).

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

Constraint: $\epsilon \leq$ **options.lin_feas_tol** < 1.0.

nonlin_feas_tol – double Default $= \epsilon^{0.33}$ or $\sqrt{\epsilon}$

The default is $\epsilon^{0.33}$ if **options.con_deriv** $=$ Nag_FALSE, and $\sqrt{\epsilon}$ otherwise.

On entry: defines the maximum acceptable violations in the nonlinear constraints at a 'feasible' point; i.e., a nonlinear constraint is considered satisfied if its violation does not exceed **options.nonlin_feas_tol**.

The tolerance defines the largest constraint violation that is acceptable at an optimal point. Since nonlinear constraints are generally not satisfied until the final iterate, the value of **options.nonlin_feas_tol** acts as a partial termination criteria for the iterative sequence generated by nag_opt_nlp (e04ucc) (see the discussion of **options.optim_tol**).

This tolerance should reflect the precision of the nonlinear constraint functions calculated by **confun**.

Constraint: $\epsilon \leq$ **options.nonlin_feas_tol** < 1.0.

linesearch_tol – double Default $= 0.9$

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

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

Constraint: $0.0 \leq$ **options.linesearch_tol** < 1.0.

step_limit – double Default $= 2.0$

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

$$\|\tilde{x} - x\|_2 \leq r\big(1 + \|x\|_2\big),$$

where r is the value of **options.step_limit**.

The line search may go on and evaluate F and c at points further from x if this will result in a lower value of the merit function. In this case, the character `L` is printed at the end of each line of output produced by the major iterations (see Section 5.1). If `L` is printed for most of the iterations, **options.step_limit** should be set to a larger value.

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

Constraint: **options.step_limit** > 0.0.

crash_tol – double Default $= 0.01$

On entry: **options.crash_tol** is used during a 'cold start' when nag_opt_nlp (e04ucc) selects an initial working set (**options.start** $=$ Nag_Cold). The initial working set will include (if possible) bounds or general inequality constraints that lie within **options.crash_tol** of their bounds. In particular, a constraint of the form $a_j^{\mathrm{T}}x \geq l$ will be included in the initial working set if $\left|a_j^{\mathrm{T}}x - l\right| \leq$ **options.crash_tol** $\times (1 + |l|)$.

Constraint: $0.0 \leq$ **options.crash_tol** ≤ 1.0.

inf_bound – double Default $= 10^{20}$

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

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

inf_step – double Default $= \max($**options.inf_bound**$, 10^{20})$

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

Constraint: **options.inf_step** > 0.0.

conf – double Default $=$ **ncnlin**

On entry: **ncnlin** values of memory will be automatically allocated by nag_opt_nlp (e04ucc) and this is the recommended method of use of **conf**. However you may supply memory from the calling program.

On exit: if **ncnlin** > 0, **conf**$[i-1]$ contains the value of the ith nonlinear constraint function c_i at the final iterate.

If **ncnlin** $= 0$ then **conf** will not be referenced.

conjac – double Default $=$ **ncnlin** $\times$ **n**

On entry: **ncnlin** $\times$ **n** values of memory will be automatically allocated by nag_opt_nlp (e04ucc) and this is the recommended method of use of **options.conjac**. However you may supply memory from the calling program.

On exit: if **ncnlin** > 0, **conjac** contains the Jacobian matrix of the nonlinear constraint functions at the final iterate, i.e., **conjac**$[(i-1) \times$ **n** $+ j - 1]$ contains the partial derivative of the ith constraint function with respect to the jth variable, for $i = 1, 2, \ldots,$ **ncnlin** and $j = 1, 2, \ldots,$ **n**. (See the discussion of the argument **conjac** under **confun**.)

If **ncnlin** $= 0$ then **conjac** will not be referenced.

state – Integer Default $=$ **n** $+$ **nclin** $+$ **ncnlin**

On entry: **options.state** need not be set if the default option of **options.start** $=$ Nag_Cold is used as **n** $+$ **nclin** $+$ **ncnlin** values of memory will be automatically allocated by nag_opt_nlp (e04ucc).

If the option **options.start** $=$ Nag_Warm has been chosen, **options.state** must point to a minimum of **n** $+$ **nclin** $+$ **ncnlin** elements of memory. This memory will already be available if the **options** structure has been used in a previous call to nag_opt_nlp (e04ucc) from the calling program, with **options.start** $=$ Nag_Cold and the same values of **n**, **nclin** and **ncnlin**. If a previous call has not been made, you must allocate sufficient memory.

When a 'warm start' is chosen **options.state** should specify the status of the bounds and linear constraints at the start of the feasibility phase. More precisely, the first **n** elements of **options.state** refer to the upper and lower bounds on the variables, the next **nclin** elements refer to the general linear constraints and the following **ncnlin** elements refer to the nonlinear constraints. Possible values for **options.state**$[j]$ are as follows:

options.state$[j]$	**Meaning**
0	The corresponding constraint is *not* in the initial QP working set.
1	This inequality constraint should be in the initial working set at its lower bound.
2	This inequality constraint should be in the initial working set at its upper bound.
3	This equality constraint should be in the initial working set. This value must only be specified if **bl**$[j]$ = **bu**$[j]$.

The values -2, -1 and 4 are also acceptable but will be reset to zero by the function, as will any elements which are set to 3 when the corresponding elements of **bl** and **bu** are not equal. If nag_opt_nlp (e04ucc) has been called previously with the same values of **n**, **nclin** and **ncnlin**, then **options.state** already contains satisfactory information. (See also the description of the optional argument **options.start**.) The function also adjusts (if necessary) the values supplied in **x** to be consistent with the values supplied in **options.state**.

Constraint: $-2 \leq$ **options.state**$[j-1] \leq 4$, for $j = 1, 2, \ldots,$ **n** + **nclin** + **ncnlin**.

On exit: the status of the constraints in the QP working set at the point returned in **x**. The significance of each possible value of **options.state**$[j]$ is as follows:

options.state$[j]$	**Meaning**
-2	The constraint violates its lower bound by more than the appropriate feasibility tolerance (see the options **options.lin_feas_tol** and **options.nonlin_feas_tol**). This value can occur only when no feasible point can be found for a QP subproblem.
-1	The constraint violates its upper bound by more than the appropriate feasibility tolerance (see the options **options.lin_feas_tol** and **options.nonlin_feas_tol**). This value can occur only when no feasible point can be found for a QP subproblem.
0	The constraint is satisfied to within the feasibility tolerance, but is not in the QP working set.
1	This inequality constraint is included in the QP working set at its lower bound.
2	This inequality constraint is included in the QP working set at its upper bound.
3	This constraint is included in the working set as an equality. This value of **options.state** can occur only when **bl**$[j]$ = **bu**$[j]$.

lambda – double Default = **n** + **nclin** + **ncnlin**

On entry: **options.lambda** need not be set if the default option of **options.start** = Nag_Cold is used as **n** + **nclin** + **ncnlin** values of memory will be automatically allocated by nag_opt_nlp (e04ucc).

If the option **options.start** = Nag_Warm has been chosen, **options.lambda** must point to a minimum of **n** + **nclin** + **ncnlin** elements of memory. This memory will already be available if the **options** structure has been used in a previous call to nag_opt_nlp (e04ucc) from the calling program, with **options.start** = Nag_Cold and the same values of **n**, **nclin** and **ncnlin**. If a previous call has not been made, you must allocate sufficient memory.

When a 'warm start' is chosen **options.lambda**$[j-1]$ must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by **options.state**, for $j =$ **n** + **nclin** + 1, **n** + **nclin** + 2,. . ., **n** + **nclin** + **ncnlin**. The remaining elements need not be set.

Note that if the jth constraint is defined as 'inactive' by the initial value of the **options.state** array (i.e., **options.state**$[j-1] = 0$), **options.lambda**$[j-1]$ should be zero; if the jth constraint is an inequality active at its lower bound (i.e., **options.state**$[j-1] = 1$), **options.lambda**$[j-1]$ should be non-negative; if the jth constraint is an inequality active at its upper bound (i.e., **options.state**$[j-1] = 2$), **options.lambda**$[j-1]$ should be non-positive. If necessary, the function will modify **options.lambda** to match these rules.

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

h – double Default $=\mathbf{n}\times\mathbf{n}$

On entry: **options.h** need not be set if the default option of **options.start** = Nag_Cold is used, as $\mathbf{n}\times\mathbf{n}$ values of memory will be automatically allocated by nag_opt_nlp (e04ucc).

If the option **options.start** = Nag_Warm has been chosen, **options.h** must point to a minimum of $\mathbf{n}\times\mathbf{n}$ elements of memory. This memory will already be available if the calling program has used the **options** structure in a previous call to nag_opt_nlp (e04ucc) with **options.start** = Nag_Cold and the same value of **n**. If a previous call has not been made you must allocate sufficient memory.

When **options.start** = Nag_Warm is chosen, the memory pointed to by **options.h** must contain the upper triangular Cholesky factor R of the initial approximation of the Hessian of the Lagrangian function, with the variables in the natural order. Elements not in the upper triangular part of R are assumed to be zero and need not be assigned. If a previous call has been made, with **options.hessian** = Nag_TRUE, then **options.h** will already have been set correctly.

On exit: if **options.hessian** = Nag_FALSE, **options.h** contains the upper triangular Cholesky factor R of $Q^{\mathrm{T}}\tilde{H}Q$, an estimate of the transformed and re-ordered Hessian of the Lagrangian at x (see (6)).

If **options.hessian** = Nag_TRUE, **options.h** contains the upper triangular Cholesky factor R of H, the approximate (untransformed) Hessian of the Lagrangian, with the variables in the natural order.

hessian – Nag_Boolean Default = Nag_FALSE

On entry: controls the contents of the optional argument **options.h** on return from nag_opt_nlp (e04ucc). nag_opt_nlp (e04ucc) works exclusively with the *transformed* and *re-ordered* Hessian H_Q, and hence extra computation is required to form the Hessian itself. If **options.hessian** = Nag_FALSE, **options.h** contains the Cholesky factor of the transformed and re-ordered Hessian. If **options.hessian** = Nag_TRUE, the Cholesky factor of the approximate Hessian itself is formed and stored in **options.h**. This information is required by nag_opt_nlp (e04ucc) if the next call to nag_opt_nlp (e04ucc) will be made with optional argument **options.start** = Nag_Warm.

iter – Integer

On exit: the number of major iterations which have been performed in nag_opt_nlp (e04ucc).

nf – Integer

On exit: the number of times the objective function has been evaluated (i.e., number of calls of **objfun**). The total excludes any calls made to **objfun** for purposes of derivative checking.

12.3 Description of Printed Output

The level of printed output can be controlled with the structure members **options.list**, **options.print_deriv**, **options.print_level** and **options.minor_print_level** (see Section 12.2). If **options.list** = Nag_TRUE then the argument values to nag_opt_nlp (e04ucc) are listed, followed by the result of any derivative check if **options.print_deriv** = Nag_D_Sum or Nag_D_Full. The printout of results is governed by the values of **options.print_level** and **options.minor_print_level**. The default of **options.print_level** = Nag_Soln_Iter and **options.minor_print_level** = Nag_NoPrint provides a single line of output at each iteration and the final result. This section describes all of the possible levels of results printout available from nag_opt_nlp (e04ucc).

If a simple derivative check, **options.verify_grad** = Nag_SimpleCheck, is requested then a statement indicating success or failure is given. The largest error found in the constraint Jacobian is output together with the directional derivative, $g^{\mathrm{T}}p$, of the objective gradient and its finite difference approximation, where p is a random vector of unit length.

When a component derivative check (see **options.verify_grad** in Section 12.2) is selected the element with the largest relative error is identified for the objective gradient and the constraint Jacobian.

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

`x[i]`	the element of x.
`dx[i]`	the optimal finite difference interval.
`g[i]` or `Jacobian value`	the gradient/Jacobian element.
`Difference approxn.`	the finite difference approximation.
`Itns`	the number of trials performed to find a suitable difference interval.

The indicator, `OK` or `BAD?`, states whether the gradient/Jacobian element and finite difference approximation are in agreement. If the derivatives are believed to be in error nag_opt_nlp (e04ucc) will exit with **fail.code** = NE_DERIV_ERRORS.

When **options.print_level** = Nag_Iter or Nag_Soln_Iter the following line of output is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Maj` is the major iteration count.

`Mnr` is the number of minor iterations required by the feasibility and optimality phases of the QP subproblem. Generally, `Mnr` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 11).

Note that `Mnr` may be greater than the optional argument **options.minor_max_iter** (default value $= \max(50, 3(n + n_L + n_N))$; see Section 12.2) if some iterations are required for the feasibility phase.

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

`Merit function` is the value of the augmented Lagrangian merit function (12) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty arguments (see Section 11.3). As the solution is approached, `Merit function` will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by `I` at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty arguments. During a sequence of major iterations with infeasible subproblems, the sequence of `Merit Function` values will decrease monotonically until either a feasible subproblem is obtained or nag_opt_nlp (e04ucc) terminates with **fail.code** = NW_NONLIN_NOT_FEASIBLE (no feasible point could be found for the nonlinear constraints).

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

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

`Norm Gz` is $\|Z^{\mathrm{T}} g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient (see Section 11.1). `Norm Gz` will be approximately zero in the neighbourhood of a solution.

`Cond Hz` is a lower bound on the condition number of the projected Hessian approximation H_Z $\left(H_Z = Z^{\mathrm{T}} H_{\mathrm{FR}} Z = R_Z^{\mathrm{T}} R_Z\right)$; see (6) and (11). The larger this number, the more difficult the problem.

The line of output may be terminated by one of the following characters:

`M` is printed if the quasi-Newton update was modified to ensure that the Hessian approximation is positive definite (see Section 11.4).

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

`C` is printed if central differences were used to compute the unspecified objective and constraint gradients. If the value of `Step` is zero, the switch to central differences was made because no lower point could be found in the line search. (In this case, the QP subproblem is re-solved with the central difference gradient and Jacobian.) If the value of `Step` is nonzero, central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 11.1).

`L` is printed if the line search has produced a relative change in x greater than the value defined by the optional argument **options.step_limit** (default value $= 2.0$; see Section 12.2). If this output occurs frequently during later iterations of the run, **options.step_limit** should be set to a larger value.

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

If **options.print_level** = Nag_Iter_Long, Nag_Soln_Iter_Long, Nag_Soln_Iter_Const or Nag_Soln_Iter_Full the line of printout at every iteration is extended to give the following additional information. (Note this longer line extends over more than 80 characters.)

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

`Nz` is the number of columns of Z (see Section 11.1). The value of `Nz` is the number of variables minus the number of constraints in the predicted active set; i.e., $\texttt{Nz} = n - (\texttt{Bnd} + \texttt{Lin} + \texttt{Nln})$.

`Bnd` is the number of simple bound constraints in the predicted active set.

`Lin` is the number of general linear constraints in the predicted active set.

`Nln` is the number of nonlinear constraints in the predicted active set (not printed if **ncnlin** is zero).

`Penalty` is the Euclidean norm of the vector of penalty arguments used in the augmented Lagrangian merit function (not printed if **ncnlin** is zero).

`Norm Gf` is the Euclidean norm of g_{FR}, the gradient of the objective function with respect to the free variables.

`Cond H` is a lower bound on the condition number of the Hessian approximation H.

`Cond T` is a lower bound on the condition number of the matrix of predicted active constraints.

`Conv` is a three-letter indication of the status of the three convergence tests (16) – (18) defined in the description of the optional argument **options.optim_tol** in Section 12.2. Each letter is `T` if the test is satisfied, and `F` otherwise. The three tests indicate whether:

(i) the sequence of iterates has converged;

(ii) the projected gradient (`Norm Gz`) is sufficiently small; and

(iii) the norm of the residuals of constraints in the predicted active set (`Violtn`) is small enough.

If any of these indicators is `F` when nag_opt_nlp (e04ucc) terminates with the error indicator **fail.code** = NE_NOERROR, you should check the solution carefully.

When **options.print_level** = Nag_Soln_Iter_Const or Nag_Soln_Iter_Full more detailed results are given at each iteration. If **options.print_level** = Nag_Soln_Iter_Const these additional values are: the value of x currently held in **x**; the current value of the objective function; the Euclidean norm of nonlinear constraint violations; the values of the nonlinear constraints (the vector c); and the values of the linear constraints, (the vector $A_L x$).

If **options.print_level** = Nag_Soln_Iter_Full then the diagonal elements of the matrix T associated with the TQ factorization (5) of the QP working set and the diagonal elements of R, the triangular factor of the transformed and re-ordered Hessian (6) (see Section 11.1) are also output at each iteration.

When **options.print_level** = Nag_Soln, Nag_Soln_Iter, Nag_Soln_Iter_Long, Nag_Soln_Iter_Const or Nag_Soln_Iter_Full the final printout from nag_opt_nlp (e04ucc) includes a listing of the status of every variable and constraint. The following describes the printout for each variable.

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

`State` gives the state of the variable (`FR` if neither bound is in the active set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound). If `Value` lies outside the upper or lower bounds by more than the feasibility tolerances specified by the optional arguments **options.lin_feas_tol** and **options.nonlin_feas_tol** (see Section 12.2), `State` will be `++` or `--` respectively.

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, 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 **options.lin_feas_tol**.

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

`Lower bound` is the lower bound specified for the variable j. (`None` indicates that **bl**$[j-1] \leq$ **options.inf_bound**, where **options.inf_bound** is the optional argument.)

`Upper bound` is the upper bound specified for the variable j. (`None` indicates that **bu**$[j-1] \geq$ **options.inf_bound**, where **options.inf_bound** is the optional argument.)

`Lagr Mult` is the value of the Lagrange multiplier for the associated bound constraint. This will be zero if `State` is `FR` unless **bl**$[j-1] \leq -$**options.inf_bound** and **bu**$[j-1] \geq$ **options.inf_bound**, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is `LL`, and non-positive if `State` is `UL`.

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

The meaning of the printout for linear and nonlinear constraints is the same as that given above for variables, with 'variable' replaced by 'constraint', **bl**$[j-1]$ and **bu**$[j-1]$ are replaced by **bl**$[n+j-1]$ and **bu**$[n+j-1]$ respectively, and with the following changes in the heading:

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

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

The `I` key in the `State` column is printed for general linear constraints which currently violate one of their bounds by more than **options.lin_feas_tol** and for nonlinear constraints which violate one of their bounds by more than **options.nonlin_feas_tol**.

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 `Residual` column to become positive.

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

For the output governed by **options.minor_print_level**, you are referred to the documentation for nag_opt_lin_lsq (e04ncc). This option is equivalent to **options.print_level**.

If **options.print_level** = Nag_NoPrint then printout will be suppressed; you can print the final solution when nag_opt_nlp (e04ucc) returns to the calling program.

12.3.1 Output of results via a user-defined printing function

You may also specify your own print function for output of iteration results and the final solution by use of the **options.print_fun** function pointer, which has prototype

```
void (*print_fun)(const Nag_Search_State *st, Nag_Comm *comm);
```

This section may be skipped if you wish to use the default printing facilities.

When a user-defined function is assigned to **options.print_fun** this will be called in preference to the internal print function of nag_opt_nlp (e04ucc). Calls to the user-defined function are again controlled by means of the **options.print_level**, **options.minor_print_level** and **options.print_deriv** members. Information is provided through **st** and **comm**, the two structure arguments to **options.print_fun**.

If **comm→it_maj_prt** = Nag_TRUE then results from the last major iteration of nag_opt_nlp (e04ucc) are provided through **st**. Note that **options.print_fun** will be called with **comm→it_maj_prt** = Nag_TRUE only if **options.print_level** = Nag_Iter, Nag_Soln_Iter, Nag_Soln_Iter_Long, Nag_Soln_Iter_Const or Nag_Soln_Iter_Full. The following members of **st** are set:

n – Integer

The number of variables.

nclin – Integer

The number of linear constraints.

ncnlin – Integer

The number of nonlinear constraints.

nactiv – Integer

The total number of active elements in the current set.

iter – Integer

The major iteration count.

minor_iter – Integer

The minor iteration count for the feasibility and the optimality phases of the QP subproblem.

step – double

The step taken along the computed search direction.

nfun – Integer

The cumulative number of objective function evaluations needed for the line search.

merit – double

The value of the augmented Lagrangian merit function at the current iterate.

objf – double

The current value of the objective function.

norm_nlnviol – double

The Euclidean norm of nonlinear constraint violations (only available if **st→ncnlin** > 0).

violtn – double

The Euclidean norm of the residuals of constraints that are violated or in the predicted active set (only available if **st→ncnlin** > 0).

norm_gz – double

$\|Z^{\mathrm{T}} g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient.

nz – Integer

The number of columns of Z (see Section 11.1).

bnd – Integer

The number of simple bound constraints in the predicted active set.

lin – Integer

The number of general linear constraints in the predicted active set.

nln – Integer

The number of nonlinear constraints in the predicted active set (only available if **st→ncnlin** > 0).

penalty – double

The Euclidean norm of the vector of penalty arguments used in the augmented Lagrangian merit function (only available if **st→ncnlin** > 0).

norm_gf – double

The Euclidean norm of g_{FR}, the gradient of the objective function with respect to the free variables.

cond_h – double

A lower bound on the condition number of the Hessian approximation H.

cond_hz – double

A lower bound on the condition number of the projected Hessian approximation H_Z.

cond_t – double

A lower bound on the condition number of the matrix of predicted active constraints.

iter_conv – Nag_Boolean

Nag_TRUE if the sequence of iterates has converged, i.e., convergence condition (16) (see the description of **options.optim_tol**) is satisfied.

norm_gz_small – Nag_Boolean

Nag_TRUE if the projected gradient is sufficiently small, i.e., convergence condition (17) (see the description of **options.optim_tol**) is satisfied.

violtn_small – Nag_Boolean

Nag_TRUE if the violations of the nonlinear constraints are sufficiently small, i.e., convergence condition (18) (see the description of **options.optim_tol**) is satisfied.

update_modified – Nag_Boolean

Nag_TRUE if the quasi-Newton update was modified to ensure that the Hessian is positive definite.

qp_not_feasible – Nag_Boolean

Nag_TRUE if the QP subproblem has no feasible point.

c_diff – Nag_Boolean

Nag_TRUE if central differences were used to compute the unspecified objective and constraint gradients.

step_limit_exceeded – Nag_Boolean

Nag_TRUE if the line search produced a relative change in x greater than the value defined by the optional argument **options.step_limit**.

refactor – Nag_Boolean

Nag_TRUE if the approximate Hessian has been refactorized.

x – double *

Contains the components $\mathbf{x}[j-1]$ of the current point x, for $j = 1, 2, \ldots,$ **st→n**.

state – Integer *

Contains the status of the **st→n** variables, **st→nclin** linear, and **st→ncnlin** nonlinear constraints (if any). See Section 12.2 for a description of the possible status values.

ax – double *

If **st→nclin** > 0, **st→ax**$[j-1]$ contains the current value of the jth linear constraint, for $j = 1, 2, \ldots,$ **st→nclin**.

cx – double *

If **st→ncnlin** > 0, **st→cx**$[j-1]$ contains the current value of nonlinear constraint c_j, for $j = 1, 2, \ldots,$ **st→ncnlin**.

diagt – double *

If **st→nactiv** > 0, the **st→nactiv** elements of the diagonal of the matrix T.

diagr – double *

Contains the **st→n** elements of the diagonal of the upper triangular matrix R.

If **comm→sol_sqp_prt** = Nag_TRUE then the final result from nag_opt_nlp (e04ucc) is provided through **st**. Note that **options.print_fun** will be called with **comm→sol_sqp_prt** = Nag_TRUE only if **options.print_level** = Nag_Soln, Nag_Soln_Iter, Nag_Soln_Iter_Long, Nag_Soln_Iter_Const or Nag_Soln_Iter_Full. The following members of **st** are set:

iter – Integer

The number of iterations performed.

n – Integer

The number of variables.

nclin – Integer

The number of linear constraints.

ncnlin – Integer

The number of nonlinear constraints.

x – double *

Contains the components $\mathbf{x}[j-1]$ of the final point x, for $j = 1, 2, \ldots,$ **st→n**.

state – Integer *

Contains the status of the **st→n** variables, **st→nclin** linear, and **st→ncnlin** nonlinear constraints (if any). See Section 12.2 for a description of the possible status values.

ax – double *

If **st→nclin** > 0, **st→ax**$[j-1]$ contains the final value of the jth linear constraint, for $j = 1, 2, \ldots,$ **st→nclin**.

cx – double *

If **st→ncnlin** > 0, **st→cx**$[j-1]$ contains the final value of nonlinear constraint c_j, for $j = 1, 2, \ldots,$ **st→ncnlin**.

bl – double *

Contains the **st→n** + **st→nclin** + **st→ncnlin** lower bounds on the variables.

bu – double *

Contains the **st→n** + **st→nclin** + **st→ncnlin** upper bounds on the variables.

lambda – double *

Contains the **st→n** + **st→nclin** + **st→ncnlin** final values of the Lagrange multipliers.

If **comm→g_prt** = Nag_TRUE then the results from derivative checking are provided through **st**. Note that **options.print_fun** will be called with **comm→g_prt** only if **options.print_deriv** = Nag_D_Sum or Nag_D_Full. The following members of **st** are set:

n – Integer

The number of variables.

ncnlin – Integer

The number of nonlinear constraints.

x – double *

Contains the components **x**$[j-1]$ of the initial point x_0, for $j = 1, 2, \ldots,$ **st→n**.

g – double *

Contains the components **g**$[j-1]$ of the gradient vector $g(x) = \left(\frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, \ldots, \frac{\partial F}{\partial x_n}\right)^{\mathrm{T}}$ at the initial point x_0, for $j = 1, 2, \ldots,$ **st→n**.

conjac – double *

Contains the elements of the Jacobian matrix of nonlinear constraints at the initial point x_0 ($\frac{\partial f_i}{\partial x_j}$ is held at location **conjac**$[(i-1) \times$ **st→n** $+ j - 1]$, for $i = 1, 2, \ldots,$ **st→ncnlin** and $j = 1, 2, \ldots,$ **st→n**).

In this case details of the derivative check performed by nag_opt_nlp (e04ucc) are held in the following substructure of **st**:

gprint – Nag_GPrintSt *

Which in turn contains three substructures **st→g_chk**, **st→f_sim**, **st→c_sim** and two pointers to arrays of substructures, **st→f_comp** and **st→c_comp**.

g_chk – Nag_Grad_Chk_St *

The substructure **st→g_chk** contains the members:

type – Nag_GradChk

The type of derivative check performed by nag_opt_nlp (e04ucc). This will be the same value as in **options.verify_grad**.

g_error – Integer

This member will be equal to one of the error codes NE_NOERROR or NE_DERIV_ERRORS according to whether the derivatives were found to be correct or not.

obj_start – Integer

Specifies the gradient element at which any component check started. This value will be equal to **options.obj_check_start**.

obj_stop – Integer

Specifies the gradient element at which any component check ended. This value specifies the element at which any component check of the constraint gradient ended. This value will be equal to **options.obj_check_stop**.

con_start – Integer

Specifies the element at which any component check of the constraint gradient started. This value will be equal to **options.con_check_start**.

con_stop – Integer

Specifies the element at which any component check of the constraint gradient ended. This value will be equal to **options.con_check_stop**.

f_sim – Nag_SimSt *

The result of a simple derivative check of the objective gradient, **st→gprint→g_chk.type** = Nag_SimpleCheck, will be held in this substructure in members:

correct – Nag_Boolean

If Nag_TRUE then the objective gradient is consistent with the finite difference approximation according to a simple check.

dir_deriv – double

The directional derivative $g^{\mathrm{T}}p$ where p is a random vector of unit length with elements of approximately equal magnitude.

fd_approx – double

The finite difference approximation, $(F(x+hp)-F(x))/h$, to the directional derivative.

c_sim – Nag_SimSt *

The result of a simple derivative check of the constraint Jacobian, **st→gprint→g_chk.type** = Nag_SimpleCheck, will be held in this substructure in members:

n_elements – Integer

The number of columns of the constraint Jacobian for which a simple check has been carried out, i.e., those columns which do not contain unknown elements.

correct – Nag_Boolean

If Nag_TRUE then the Jacobian is consistent with the finite difference approximation according to a simple check.

max_error – double

The maximum error found between the norm of a constraint gradient and its finite difference approximation.

max_constraint – Integer

The constraint gradient which has the maximum error between its norm and its finite difference approximation.

f_comp – Nag_CompSt *

The results of a requested component derivative check of the objective gradient, **st**→**gprint**→**g_chk**.**type** = Nag_CheckObj or Nag_CheckObjCon, will be held in the array of **st**→**n** substructures of type Nag_CompSt pointed to by **st**→**gprint**→**f_comp**. The procedure for the derivative check is based on finding an interval that produces an acceptable estimate of the second derivative, and then using that estimate to compute an interval that should produce a reasonable forward-difference approximation. The gradient element is then compared with the difference approximation. (The method of finite difference interval estimation is based on Gill *et al.* (1983).)

correct – Nag_Boolean

If Nag_TRUE then this gradient element is consistent with its finite difference approximation.

hopt – double

The optimal finite difference interval.

gdiff – double

The finite difference approximation for this gradient component.

iter – Integer

The number of trials performed to find a suitable difference interval.

comment – char

A character string which describes the possible nature of the reason for which an estimation of the finite difference interval failed to produce a satisfactory relative condition error of the second-order difference. Possible strings are: `"Constant?"`, `"Linear or odd?"`, `"Too nonlinear?"` and `"Small derivative?"`.

c_comp – Nag_CompSt *

The results of a requested component derivative check of the Jacobian of nonlinear constraint functions, **st**→**gprint**→**g_chk**.**type** = Nag_CheckCon or Nag_CheckObjCon, will be held in the array of **st**→**ncnlin** × **st**→**n** substructures of type Nag_CompSt pointed to by **st**→**gprint**→**c_comp**. The element **st**→**gprint**→**f_comp**$[(i-1) \times$ **st**→**n** $+ j - 1]$ will hold the details of the component derivative check for Jacobian element i, j, for $i = 1, 2, \ldots,$ **st**→**ncnlin** and $j = 1, 2, \ldots,$ **st**→**n**. The procedure for the derivative check is based on finding an interval that produces an acceptable estimate of the second derivative, and then using that estimate to compute an interval that should produce a reasonable forward-difference approximation. The Jacobian element is then compared with the difference approximation. (The method of finite difference interval estimation is based on Gill *et al.* (1983).)

The members of **st**→**gprint**→**c_comp** are as for **st**→**gprint**→**f_comp** where **st**→**gprint**→**f_comp**.**gdiff** gives the difference approximation for the Jacobian element.

The relevant members of the structure **comm** are:

g_prt – Nag_Boolean

Will be Nag_TRUE only when the print function is called with the result of the derivative check of **objfun** and **confun**.

it_maj_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with information about the current major iteration.

sol_sqp_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with the details of the final solution.

it_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with information about the current minor iteration (i.e., an iteration of the current QP subproblem). See the documentation for nag_opt_lin_lsq (e04ncc) for details of which members of **st** are set.

new_lm – Nag_Boolean

Will be Nag_TRUE when the Lagrange multipliers have been updated in a QP subproblem. See the documentation for nag_opt_lin_lsq (e04ncc) for details of which members of **st** are set.

sol_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with the details of the solution of a QP subproblem, i.e., the solution at the end of a major iteration. See the documentation for nag_opt_lin_lsq (e04ncc) for details of which members of **st** are set.

user – double
iuser – Integer
p – Pointer

Pointers for communication of user information. If used they must be allocated memory either before entry to nag_opt_nlp (e04ucc) or during a call to **objfun**, **confun** or **options**.**print_fun**. The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.
