nag_opt_nlin_lsq (e04unc)

1. Purpose

nag_opt_nlin_lsq (e04unc) is designed to minimize an arbitrary smooth sum of squares function subject to constraints (which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints) using a sequential quadratic programming (SQP) method. As many first derivatives as possible should be supplied by the user; any unspecified derivatives are approximated by finite differences. It is not intended for large sparse problems.

nag_opt_nlin_lsq may also be used for unconstrained, bound-constrained and linearly constrained optimization.

2. Specification

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

void nag_opt_nlin_lsq(Integer m, Integer n, Integer nclin, Integer ncnlin,
                      double a[], Integer tda,
                      double bl[], double bu[], double y[],
                      void (*objfun)(Integer m, Integer n, double x[],
                            double f[], double fjac[], Nag_Comm *comm),
                      void (*confun)(Integer n, Integer ncnlin, Integer needc[],
                            double x[], double conf[], double conjac[],
                            Nag_Comm *comm),
                      double x[], double *objf, double f[], double fjac[],
                      Integer tdfjac, Nag_E04_Opt *options, Nag_Comm *comm,
                      NagError *fail)
```

3. Description

nag_opt_nlin_lsq is designed to solve the nonlinear least-squares programming problem – the minimization of a smooth nonlinear sum of squares function subject to a set of constraints on the variables. The problem is assumed to be stated in the following form:

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

where $F(x)$ (the *objective function*) is a nonlinear function which can be represented as the sum of squares of m subfunctions $(y_1 - f_1(x)), (y_2 - f_2(x)), \ldots, (y_m - f_m(x))$, the y_i are constant, 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_nlin_lsq 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 parameter **inf_bound** in Section 8.2.)

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.

The user must supply an initial estimate of the solution to (1), together with functions that define $f(x) = (f_1(x), f_2(x), \ldots, f_m(x))^T, c(x)$ and as many first partial derivatives as possible; unspecified derivatives are approximated by finite differences.

The subfunctions are defined by the array **y** and function **objfun**, and the nonlinear constraints are defined by the function **confun**. On every call, these functions must return appropriate values of $f(x)$ and $c(x)$. The user should also provide the available partial derivatives. Any unspecified derivatives are approximated by finite differences; see Section 8.2 for a discussion of the optional parameters **obj_deriv** and **con_deriv**. Just before either **objfun** or **confun** is called, each element of the current gradient array **fjac** 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 for the user to provide all partial derivatives (see Chapter 8 of Gill *et al* (1981) for a detailed discussion). If all gradients cannot be provided, it is similarly advisable to provide as many as possible. While developing the functions **objfun** and **confun**, the optional parameter **verify_grad** (see Section 8.2) should be used to check the calculation of any known gradients.

nag_opt_nlin_lsq is based on upon nag_opt_nlp (e04ucc); see Section 7 of the documentation for nag_opt_nlp (e04ucc) for details of the algorithm.

4. Parameters

m

Input: m, the number of subfunctions associated with $F(x)$.
Constraint: **m** > 0.

n

Input: n, the number of variables.
Constraint: **n** > 0.

nclin

Input: n_L, the number of general linear constraints.
Constraint: **nclin** ≥ 0.

ncnlin

Input: n_N, the number of nonlinear constraints.
Constraint: **ncnlin** ≥ 0.

a[nclin][tda]

Input: 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)), for $i = 1, 2, \ldots, n_L$.

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

tda

Input: the second dimension of the array **a** as declared in the function from which nag_opt_nlin_lsq is called.
Constraint: **tda** $\geq$ **n** if **nclin** > 0.

bl[n+nclin+ncnlin]
bu[n+nclin+ncnlin]

Input: **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 non-existent lower bound (i.e., $l_j = -\infty$), set **bl**$[j-1] \leq -$**inf_bound**, and to specify a non-existent upper bound (i.e., $u_j = +\infty$), set **bu**$[j-1] \geq$ **inf_bound**, where **inf_bound** is one of the optional parameters (default value 10^{20}, see Section 8.2). To specify the jth constraint as an equality, set **bl**$[j-1] =$ **bu**$[j-1] = \beta$, say, where $|\beta| <$ **inf_bound**.
Constraints:

bl$[j] \leq$ **bu**$[j]$, for $j = 0, 1, \ldots,$**n**+**nclin**+**ncnlin**-1,

$|\beta| <$ **inf_bound** when **bl**$[j] =$ **bu**$[j] = \beta$.

y[m]

Input: the coefficients of the constant vector y in the objective function.

objfun

objfun must calculate the vector $f(x)$ of subfunctions and (optionally) its Jacobian $(= \partial f/\partial x)$ for a specified n element vector x.

The specification for **objfun** is:

```
void objfun(Integer m, Integer n, double x[], double f[],
            double fjac[], Integer tdfjac, Nag_Comm *comm)
```

m

Input: m, the number of subfunctions.

n

Input: n, the number of variables.

x[n]

Input: x, the vector of variables at which $f(x)$ and/or all available elements of its Jacobian are to be evaluated.

f[m]

Output: if **comm->flag** = 0 or 2, **objfun** must set **f**$[i-1]$ to the value of the ith subfunction f_i at the current point x, for some or all $i = 1, 2, \ldots, m$ (see the description of the parameter **comm->needf** below).

fjac[m*tdfjac]

Output: if **comm->flag** = 2, **objfun** must contain the available elements of the subfunction Jacobian matrix. **fjac**$[(i-1)*$**tdfjac**$+j-1]$ must be set to the value of the first derivative $\partial f_i/\partial x_j$ at the current point x for $i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$.

If the optional parameter **obj_deriv** = **TRUE** (the default), all elements of **fjac** must be set; if **obj_deriv** = **FALSE**, any available elements of the Jacobian matrix must be assigned to the elements of **fjac**; the remaining elements *must remain unchanged.*

Any constant elements of **fjac** may be assigned once only at the first call to **objfun**, i.e., when **comm->first** = **TRUE**. This is only effective if the optional parameter **obj_deriv** = **TRUE**.

tdfjac

Input: the second dimension of the array **fjac** as declared in the function from which nag_opt_nlin_lsq is called.

comm

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

flag – Integer

Input: **objfun** is called with **comm->flag** set to 0 or 2.

If **comm->flag** = 0 then only **f** is referenced. If **comm->flag** = 2 then both **f** and **fjac** are referenced.

Output: if **objfun** resets **comm->flag** to some negative number then nag_opt_nlin_lsq will terminate immediately with the error indicator **NE_USER_STOP**. If **fail** is supplied to nag_opt_nlin_lsq **fail.errnum** will be set to the user's setting of **comm->flag**.

first – Boolean

Input: will be set to **TRUE** on the first call to **objfun** and **FALSE** for all subsequent calls.

nf – Integer

Input: 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.

needf – Integer

Input: if **needf** $= 0$, **objfun** must set, for all $i = 1, 2, \ldots,$**m**, **f**$[i-1]$ to the value of the ith subfunction f_i at the current point **x**. If **needf** $= i$, for $i = 1, 2, \ldots,$**m**, then it is sufficient to set **f**$[i-1]$ to the value of the ith subfunction f_i. Appropriate use of **needf** can save a lot of computational work in some cases. Note that when **comm->needf** $\neq 0$, **comm->flag** will always be 0, hence this does not apply to the Jacobian matrix.

user – double $*$
iuser – Integer $*$
p – Pointer

The type Pointer is `void *`.

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

Note: **objfun** should be tested separately before being used in conjunction with nag_opt_nlin_lsq. The optional parameters **verify_grad** and **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 Jacobian elements then the optional parameter **obj_deriv** should be set to **FALSE**.

confun

confun must calculate the vector $c(x)$ of nonlinear constraint functions and (optionally) its Jacobian ($= \partial c/\partial x$) 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_nlin_lsq. If there are nonlinear constraints the first call to **confun** will occur before the first call to **objfun**.

The specification for **confun** is:

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

n

Input: n, the number of variables.

ncnlin

Input: n_N, the number of nonlinear constraints.

needc[ncnlin]

Input: 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 parameter **comm->flag** below) must be evaluated at x.

x[n]

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

conf[ncnlin]

Output: 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.

conjac[ncnlin*n]

Output: if **needc**$[i-1] > 0$ and **comm->flag** $= 2$, the ith row of **conjac** (i.e., the elements **conjac**$[(i-1)*$**n**$+j-1]$, $j = 1, 2, \ldots, n$) must contain the available elements of the vector ∇c_i given by

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

where $\partial c_i/\partial x_j$ is the partial derivative of the ith constraint with respect to the jth variable, evaluated at the point x. The remaining rows of **conjac**, corresponding to non-positive elements of **needc**, are ignored.

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

If all elements of the constraint Jacobian are known (i.e., **con_deriv** = **TRUE**; see Section 8.2), 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** = **TRUE**.

It must be emphasized that, if **con_deriv** = **FALSE**, unassigned elements of **conjac** are not treated as constant; they are estimated by finite differences, at non-trivial expense. If the user does not supply a value for the optional argument **f_diff_int** (the default; see Section 8.2), 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.

comm

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

flag – Integer

Input: **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.

Output: if **confun** resets **comm->flag** to some negative number then nag_opt_nlin_lsq will terminate immediately with the error indicator **NE_USER_STOP**. If **fail** is supplied to nag_opt_nlin_lsq **fail.errnum** will be set to the user's setting of **comm->flag**.

first – Boolean

Input: will be set to **TRUE** on the first call to **confun** and **FALSE** for all subsequent calls.

user – double *
iuser – Integer *
p – Pointer

The type Pointer is `void *`.

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

Note: **confun** should be tested separately before being used in conjunction with nag_opt_nlin_lsq. The optional parameters **verify_grad** and **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 Jacobian constraint elements then the optional parameter **con_deriv** should be set to **FALSE**.

x[n]

Input: an initial estimate of the solution.
Output: the final estimate of the solution.

objf

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

f[m]

Output: the values of the subfunctions f_i, for $i = 1, 2, \ldots,$**m**, at the final iterate.

fjac[m][tdfjac]

Output: the Jacobian matrix of the functions $f_1, f_2, \ldots, f_m$ at the final iterate, i.e., **fjac**$[i-1][j-1]$ contains the partial derivative of the ith subfunction with respect to the jth variable, for $i = 1, 2, \ldots,$**m**; $j = 1, 2, \ldots,$**n**. (See also the discussion of parameter **fjac** under **objfun**.)

tdfjac

Input: the second dimension of the array **fjac** as declared in the function from which nag_opt_nlin_lsq is called.

options

Input/Output: a pointer to a structure of type Nag_E04_Opt whose members are optional parameters for nag_opt_nlin_lsq. These structure members offer the means of adjusting some of the parameter values of the algorithm and on output will supply further details of the results. A description of the members of **options** is given below in Section 8. Some of the results returned in **options** can be used by nag_opt_nlin_lsq to perform a 'warm start' (see the member **start** in Section 8.2).

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

comm

Input/Output: 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 8.3.1 for details. If the user does not need to make use of this communication feature the null pointer `NAGCOMM_NULL` may be used in the call to nag_opt_nlin_lsq; **comm** will then be declared internally for use in calls to user-supplied functions.

fail

The NAG error parameter, see the Essential Introduction to the NAG C Library.
Users are recommended to declare and initialize **fail** and set **fail.print** = **TRUE** for this function.

4.1. Description of Printed Output

Intermediate and final results are printed out by default. The level of printed output can be controlled by the user with the structure members **options.print_level** and **options.minor_print_level** (see Section 8.2). The default setting of **print_level** = **Nag_Soln_Iter** and **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_nlin_lsq.

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 7 of the documentation for nag_opt_nlp (e04ucc)).

Note that `Mnr` may be greater than the optional parameter **minor_max_iter** (default value = $\max(50{,}3(n + n_L + n_N))$; see Section 8.2) 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 at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see Section 7.3 of the documentation for nag_opt_nlp (e04ucc)). As the solution is approached, `Merit function` will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by `I` at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty parameters. During a sequence of major iterations with infeasible subproblems, the sequence of `Merit Function` values will decrease monotonically until either a feasible subproblem is obtained or nag_opt_nlin_lsq 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 $\|Z^T g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient (see Section 7.1 of the documentation for nag_opt_nlp (e04ucc)). `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 ($H_Z = Z^T H_{\mathrm{FR}} Z = R_Z^T R_Z$; see (6) and (11) in Section 7.1 and Section 7.2, respectively, of the documentation for nag_opt_nlp (e04ucc)). 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 7.4 of the documentation for nag_opt_nlp (e04ucc)).

`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 non-zero, central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 7.1 of the documentation for nag_opt_nlp (e04ucc)).

`L` is printed if the line search has produced a relative change in x greater than the value defined by the optional parameter **step_limit** (default value = 2.0; see

Section 8.2). If this output occurs frequently during later iterations of the run, **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 $j = 1, 2, ..., n$ 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 parameters **lin_feas_tol** and **nonlin_feas_tol** (see Section 8.2), `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 **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**$[j-1] \leq$ **inf_bound**, where **inf_bound** is the optional parameter.)

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

`Lagr Mult` is the value of the Lagrange multiplier for the associated bound constraint. This will be zero if `State` is `FR` unless **bl**$[j-1] \leq -$**inf_bound** and **bu**$[j-1] \geq$ **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**$[j-1] \leq -$**inf_bound** and **bu**$[j-1] \geq$ **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 $j = 1, 2, ..., n_L$ of the linear constraint.

`N Con` gives the name (`N`) and index $(j - n_L)$, for $j = n_L + 1, n_L + 2, ..., n_L + n_N$ 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 **lin_feas_tol** and for nonlinear constraints which violate one of their bounds by more than **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.

5. Comments

A list of possible error exits and warnings from nag_opt_nlin_lsq is given in Section 9. The termination criteria and accuracy of nag_opt_nlin_lsq are considered in Section 10.

6. Example 1

This is based on Problem 57 in Hock and Schittkowski (1981) and involves the minimization of the sum of squares function

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

where

$$f_i(x) = y_i - x_i - (0.49 - x_1)e^{-x_2(a_i-8)}$$

and

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

subject to the bounds

$$x_1 \geq 0.4$$
$$x_2 \geq -4.0$$

to the general linear constraint

$$x_1 + x_2 \geq 1.0,$$

and to the nonlinear constraint

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

The initial point, which is infeasible, is

$$x_0 = (0.4,\ 0.0)^T,$$

and $F(x_0) = 0.002241$.

The optimal solution (to five figures) is

$$x^* = (0.41995,\ 1.28484)^T,$$

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

This example shows the simple use of nag_opt_nlin_lsq where default values are used for all optional parameters. An example showing the use of optional parameters is given in Section 13. There is one example program file, the main program of which calls both examples. The main program and Example 1 are given below.

6.1. Program Text

```
/* nag_opt_nlin_lsq(e04unc) Example Program.
 *
 * Copyright 1998 Numerical Algorithms Group.
 *
 * Mark 5, 1998.
 *
 * Mark 6 revised, 2000.
 */
#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <math.h>
#include <nage04.h>

static void objfun(Integer m, Integer n, double x[], double f[],
                   double fjac[], Integer tdfjac, Nag_Comm *comm);
static void confun(Integer n, Integer ncnlin, Integer needc[], double x[],
                   double conf[], double cjac[], Nag_Comm *comm);
static void user_print(const Nag_Search_State *st, Nag_Comm *comm);
static void ex1(void);
static void ex2(void);


static void objfun(Integer m, Integer n, double x[], double f[],
                   double fjac[], Integer tdfjac, Nag_Comm *comm)
{
#define FJAC(I,J) fjac[(I)*tdfjac + (J)]

  /*  Initialized data */
  static double a[44] = {
    8.0, 8.0, 10.0, 10.0, 10.0, 10.0, 12.0, 12.0, 12.0, 12.0, 14.0, 14.0,
    14.0, 16.0, 16.0, 16.0, 18.0, 18.0, 20.0, 20.0, 20.0, 22.0, 22.0, 22.0,
    24.0, 24.0, 24.0, 26.0, 26.0, 26.0, 28.0, 28.0, 30.0, 30.0, 30.0, 32.0,
    32.0, 34.0, 36.0, 36.0, 38.0, 38.0, 40.0, 42.0 };


  /* Local variables */
  double temp;
  Integer i;
  double x0, x1, ai;


  /* Function to evaluate the objective subfunctions
   * and their 1st derivatives.
   */
  x0 = x[0];
```

```
  x1 = x[1];
  for (i = 0; i < m; ++i)
    {
      /* Evaluate objective subfunction f(i+1) only if required */
      if (comm->needf == i+1 || comm->needf == 0)
        {
          ai = a[i];
          temp = exp(-x1 * (ai - 8.0));
          f[i] = x0 + (.49 - x0) * temp;
        }
      if (comm->flag == 2)
        {
          FJAC(i,0) = 1.0 - temp;
          FJAC(i,1) = -(.49 - x0) * (ai - 8.0) * temp;
        }
    }
} /* objfun */

static void confun(Integer n, Integer ncnlin, Integer needc[], double x[],
                   double conf[], double cjac[], Nag_Comm *comm)
{
#define CJAC(I,J) cjac[(I)*n + (J)]

  /*  Function to evaluate the nonlinear constraints and its 1st derivatives. */

  if (comm->first == TRUE)
    {
      /*  First call to confun.  Set all Jacobian elements to zero.
       *  Note that this will only work when  options.obj_deriv = TRUE
       *  (the default).
       */
      CJAC(0,0) = CJAC(0,1) = 0.0;
    }

  if (needc[0] > 0)
    {
      conf[0] = -.09 - x[0]*x[1] + 0.49*x[1];

      if (comm->flag == 2)
        {
          CJAC(0,0) = -x[1];
          CJAC(0,1) = -x[0] + .49;
        }
    }
} /* confun */


main()
{
  Vprintf("e04unc Example Program Results\n");
  ex1();
  ex2();
  exit(EXIT_SUCCESS);
}


static void ex1(void)
{
#define NMAX 10
#define MMAX 50
#define NCLIN 10
#define NCNLIN 10
#define MAXBND NMAX+NCLIN+NCNLIN

  /* Local variables */
  double x[NMAX], a[NCLIN][NMAX];
  double f[MMAX], y[MMAX], fjac[MMAX][NMAX];
  double bl[MAXBND], bu[MAXBND];
  double objf;
  Integer tda, tdfjac;
```

```
  Integer i, j, m, n, nclin, ncnlin;
  static NagError fail;

  fail.print = TRUE;

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

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

  if (m <= MMAX && n <= NMAX && nclin <= NCLIN && ncnlin <= NCNLIN)
    {
      tda = NMAX;
      tdfjac = NMAX;
      /*  Read a, y, bl, bu and x from data file */

      if (nclin > 0)
        {
          Vscanf(" %*[^\n]");
          for (i = 0; i < nclin; ++i)
            for (j = 0; j < n; ++j)
              Vscanf("%lf",&a[i][j]);
        }

      /* Read the y vector of the objective */
      Vscanf(" %*[^\n]");
      for (i = 0; i < m; ++i)
        Vscanf("%lf",&y[i]);

      /* Read lower bounds */
      Vscanf(" %*[^\n]");
      for (i = 0; i < n + nclin + ncnlin; ++i)
        Vscanf("%lf",&bl[i]);

      /* Read upper bounds */
      Vscanf(" %*[^\n]");
      for (i = 0; i < n + nclin + ncnlin; ++i)
        Vscanf("%lf",&bu[i]);

      /* Read the initial point x */
      Vscanf(" %*[^\n]");
      for (i = 0; i < n; ++i)
        Vscanf("%lf",&x[i]);

      /*  Solve the problem */
      e04unc(m, n, nclin, ncnlin, (double*)a, tda, bl, bu, y, objfun,
             confun, x, &objf, f, (double*)fjac, tdfjac, E04_DEFAULT,
             NAGCOMM_NULL, &fail);
    }

} /* ex1 */
```

6.2. Program Data

```
e04unc Example Program Data

Data for example 1

Values of m and n
  44  2

Values of nclin and ncnln
  1   1

Linear constraint matrix A
  1.0   1.0
```

```
Objective vector y
  0.49  0.49  0.48  0.47  0.48  0.47  0.46  0.46  0.45  0.43  0.45
  0.43  0.43  0.44  0.43  0.43  0.46  0.45  0.42  0.42  0.43  0.41
  0.41  0.40  0.42  0.40  0.40  0.41  0.40  0.41  0.41  0.40  0.40
  0.40  0.38  0.41  0.40  0.40  0.41  0.38  0.40  0.40  0.39  0.39

Lower bounds
  0.4      -4.0      1.0      0.0

Upper bounds
  1.0e+25   1.0e+25  1.0e+25  1.0e+25

Initial estimate of x
  0.4   0.0
```

6.3. Program Results

```
e04unc Example Program Results

Example 1: default options

Parameters to e04unc
--------------------

Number of variables...........   2
Linear constraints............   1    Nonlinear constraints.........   1

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..................    FALSE    h_reset_freq............         2
h_unit_init.............     FALSE
f_diff_int.............. Automatic    c_diff_int.............. Automatic
obj_deriv...............      TRUE    con_deriv...............      TRUE
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.
----------------------------------------

All objective gradient elements have been set.

Simple Check:

The Jacobian seems to be ok.

The largest relative error was 1.04e-08 in subfunction 3


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

All constraint gradient elements have been set.

Simple Check:

The Jacobian seems to be ok.

The largest relative error was 1.89e-08 in constraint 1

   Maj  Mnr     Step   Merit function  Violtn  Norm Gz  Cond Hz
    0    2  0.0e+00    2.224070e-02  3.6e-02  8.5e-02  1.0e+00
    1    1  1.0e+00    1.455402e-02  9.8e-03  1.5e-03  1.0e+00
    2    1  1.0e+00    1.436491e-02  7.2e-04  4.9e-03  1.0e+00
```

```
    3    1  1.0e+00    1.427013e-02  9.2e-06  2.9e-03  1.0e+00
    4    1  1.0e+00    1.422989e-02  3.6e-05  1.6e-04  1.0e+00
    5    1  1.0e+00    1.422983e-02  6.4e-08  5.4e-07  1.0e+00
    6    1  1.0e+00    1.422983e-02  9.8e-13  3.4e-09  1.0e+00
Exit from NP problem after 6 major iterations, 8 minor iterations.


Varbl State     Value      Lower Bound   Upper Bound    Lagr Mult    Residual
V   1    FR   4.19953e-01   4.00000e-01      None      0.0000e+00   1.9953e-02
V   2    FR   1.28485e+00  -4.00000e+00      None      0.0000e+00   5.2848e+00


L Con State     Value      Lower Bound   Upper Bound    Lagr Mult    Residual
L   1    FR   1.70480e+00   1.00000e+00      None      0.0000e+00   7.0480e-01


N Con State     Value      Lower Bound   Upper Bound    Lagr Mult    Residual
N   1    LL  -9.76774e-13   0.00000e+00      None      3.3358e-02  -9.7677e-13

Optimal solution found.

Final objective value =    1.4229835e-02
```

7. Further Description

nag_opt_nlin_lsq implements a sequential quadratic programmming (SQP) method incorporating an augmented Lagrangian merit function and a BFGS (Broyden–Fletcher–Goldfarb–Shanno) and is based on nag_opt_nlp (e04ucc). The documentation for nag_opt_nlp (e04ucc) and nag_opt_lin_lsq (e04ncc) should be consulted for details of the method.

8. Optional Parameters

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

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

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

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

8.1. Optional Parameter Checklist and Default Values

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

`Nag_Start start`	**Nag_Cold**
`Boolean list`	**TRUE**
`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`	**TRUE**
`Boolean con_deriv`	**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(**n**+**nclin**)+10**ncnlin**)
`Integer minor_max_iter`	max(50,3(**n**+**nclin**+**ncnlin**))
`double f_prec`	$\epsilon^{0.9}$
`double optim_tol`	**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(**inf_bound**,10^{20})
`double *conf`	size **ncnlin**
`double *conjac`	size **ncnlin**$*$**n**
`Integer *state`	size **n**+**nclin**+**ncnlin**
`double *lambda`	size **n**+**nclin**+**ncnlin**
`double *h`	size **n**$*$**n**
`Boolean hessian`	**FALSE**
`Boolean h_unit_init`	**FALSE**
`Integer h_reset_freq`	2
`Integer iter`	
`Integer nf`	

8.2. Description of Optional Parameters

start – Nag_Start Default = **Nag_Cold**

Input: 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 **start** = **Nag_Cold**, nag_opt_nlin_lsq 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 the optional parameter **crash_tol**; see below).

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

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

list – Boolean Default = **TRUE**

Input: if **options.list** = **TRUE** the parameter settings in the call to nag_opt_nlin_lsq will be printed.

print_level – Nag_PrintType Default = **Nag_Soln_Iter**

Input: the level of results printout produced by nag_opt_nlin_lsq at each major iteration. The following values are available.

Nag_NoPrint	No output.
Nag_Soln	The final solution.
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	**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 $A_L x$ 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 (see (5) in Section 7.1 of the documentation for nag_opt_nlp (e04ucc)) of the QP working set, and the diagonal elements of R, the triangular factor of the transformed and re-ordered Hessian (see (6) in Section 7.1 of the documentation for nag_opt_nlp (e04ucc)).

Details of each level of results printout are described in Section 8.3.
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**

Input: the level of results printout produced by the minor iterations of nag_opt_nlin_lsq (i.e., the iterations of the QP subproblem). The following values are available.

Nag_NoPrint	No output.
Nag_Soln	The final solution.
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 (see (4) in Section 7.2 of the documentation for nag_opt_lin_lsq (e04ncc)) 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 8.3 of the function documentation for nag_opt_lin_lsq (e04ncc). (**options.minor_print_level** in the present function is equivalent to **options.print_level** is nag_opt_lin_lsq.)
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 – char[80] Default = `stdout`

Input: 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

Input: printing function defined by the user; the prototype of **print_fun** is

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

See Section 8.3.1 below for further details.

obj_deriv – Boolean Default = **TRUE**

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

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

If **obj_deriv** = **FALSE**, nag_opt_nlin_lsq will approximate unspecified elements of the objective Jacobian, 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* (1981), for a discussion of limiting accuracy).

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

con_deriv – Boolean Default = **TRUE**

Input: this argument indicates whether all elements of the constraint Jacobian are provided by the user in function **confun**. If none or only some of the derivatives are being supplied by **confun** then **con_deriv** should be set to **FALSE**.

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

If **con_deriv** = **FALSE**, nag_opt_nlin_lsq will approximate unspecified elements of the constraint 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 an element provided by the user and '?' indicates an unspecified element, nag_opt_nlin_lsq 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 the user's control.)

verify_grad – Nag_GradChk Default = **Nag_SimpleCheck**

Input: specifies the level of derivative checking to be performed by nag_opt_nlin_lsq on the gradient elements computed by the user-supplied functions **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 **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 **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 parameter **obj_check_start** and **obj_check_stop** for the objective gradient, with default values 1 and **n**, respectively. For the constraint gradient the range is specified by **con_check_start** and **con_check_stop**, with default values 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_nlin_lsq if the optional parameter **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**

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

If a component derivative check has been carried out, then full details will be printed if **print_deriv** = **Nag_D_Full**. For a printout summarizing the results of a component derivative check set **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 **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** is equal to one of **Nag_CheckObj**, **Nag_CheckObjCon**, **Nag_XCheckObj** or **Nag_XCheckObjCon**.

Input: these parameters may be used to control the verification of Jacobian elements computed by the function **objfun**. For example, if the first 30 columns of the objective Jacobian appeared to be correct in an earlier run, so that only column 31 remains questionable, it is reasonable to specify **obj_check_start** = 31. If the first 30 variables appear linearly in the subfunctions, so that the corresponding Jacobian elements are constant, the above choice would also be appropriate.
Constraint: $1 \leq$ **options.obj_check_start** $\leq$ **options.obj_check_stop** $\leq$ **n**.

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

These options take effect only when **options.verify_grad** is equal to one of **Nag_CheckCon**, **Nag_CheckObjCon**, **Nag_XCheckCon** or **Nag_XCheckObjCon**.

Input: these parameters may be used to control the verification of the Jacobian elements computed by the function **confun**. For example, if the first 30 columns of the constraint Jacobian appeared to be correct in an earlier run, so that only column 31 remains questionable, it is reasonable to specify **con_check_start** = 31.
Constraint: $1 \leq$ **options.con_check_start** $\leq$ **options.con_check_stop** $\leq$ **n**.

f_diff_int – double Default = computed automatically

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

(a) For verifying the objective and/or constraint gradients (see the description of the optional parameter **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 $\delta_j = r(1 + |\hat{x}_j|)$, with $\hat{x}$ the first point feasible with respect to the bounds and linear constraints. If the functions are well scaled, the resulting derivative approximation should be accurate to $\mathrm{O}(r)$. See Gill *et al* (1981) for a discussion of the accuracy in finite difference approximations.

If a difference interval is not specified by the user, 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 the user wishes to have nag_opt_nlin_lsq determine constant elements in the objective and constraint gradients (see the descriptions of **confun** and **objfun** in Section 4).
Constraint: $\epsilon \leq$ **options.f_diff_int** < 1.0.

c_diff_int – double Default = computed automatically

Input: if the algorithm switches to central differences because the forward-difference approximation is not sufficiently accurate the value of **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 4.1). The use of finite-differences is discussed under the option **f_diff_int**.
Constraint: $\epsilon \leq$ **options.c_diff_int** < 1.0.

max_iter – Integer Default = max(50,3(**n**+**nclin**)+10**ncnlin**)

Input: the maximum number of major iterations allowed before termination.
Constraint: **options.max_iter** ≥ 0.

minor_max_iter – Integer Default = max(50,3(**n**+**nclin**+**ncnlin**))

Input: 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 = $\epsilon^{0.9}$

Input: this parameter 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_nlin_lsq 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 = **f_prec**$^{0.8}$

Input: specifies the accuracy to which the user wishes the final iterate to approximate a solution of the problem. Broadly speaking, **optim_tol** indicates the number of correct figures desired in the objective function at the solution. For example, if **optim_tol** is 10^{-6} and nag_opt_nlin_lsq terminates successfully, the final value of F should have approximately six correct figures.

nag_opt_nlin_lsq 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 7.1 of the documentation for nag_opt_nlp (e04ucc)). The sequence of iterates is considered to have converged at x if

$$\alpha\|p\| \leq \sqrt{r}(1 + \|x\|), \tag{2}$$

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

$$\|Z^T g_{\mathrm{FR}}\| \leq \sqrt{r}(1 + \max(1 + |F(x)|, \|g_{\mathrm{FR}}\|)) \tag{3}$$

and

$$|res_j| \leq ftol \text{ for all } j, \tag{4}$$

where $Z^T g_{\mathrm{FR}}$ is the projected gradient (see Section 7.1 of the documentation for nag_opt_nlp (e04ucc)), 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 $ftol$ is the value of the optional parameter **nonlin_feas_tol**.
Constraint: **options.f_prec** $\leq$ **options.optim_tol** < 1.0.

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

Input: 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 **lin_feas_tol**.

On entry to nag_opt_nlin_lsq, an iterative procedure is executed in order to find a point that satisfies the linear constraints and bounds on the variables to within the tolerance specified by **lin_feas_tol**. All subsequent iterates will satisfy the constraints to within the same tolerance (unless **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 **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 the optional parameter **con_deriv** = **FALSE**, and $\sqrt{\epsilon}$ otherwise.

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

This 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 **nonlin_feas_tol** acts as a partial termination criterion for the iterative sequence generated by nag_opt_nlin_lsq (see also the discussion of the optional parameter **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

Input: controls the accuracy with which the step α taken during each iteration approximates a minimum of the merit function along the search direction (the smaller the value of **linesearch_tol**, the more accurate the 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: $0.0 \le$ **options.linesearch_tol** < 1.0.

step_limit – double Default = 2.0

Input: 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 parameter **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

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

where r is the value of **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 4.1). If `L` is printed for most of the iterations, **step_limit** should be set to a larger value.

Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at wild values. The default value of **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 **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

Input: **crash_tol** is used during a 'cold start' when nag_opt_nlin_lsq 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 **crash_tol** of their bounds. In particular, a constraint of the form $a_j^T x \ge l$ will be included in the initial working set if $|a_j^T x - l| \le$ **crash_tol** $\times$ $(1 + |l|)$.
Constraint: $0.0 \le$ **options.crash_tol** ≤ 1.0.

inf_bound – double Default = 10^{20}

Input: **inf_bound** defines the 'infinite' bound in the definition of the problem constraints. Any upper bound greater than or equal to **inf_bound** will be regarded as plus infinity (and similarly any lower bound less than or equal to $-$**inf_bound** will be regarded as minus infinity).
Constraint: **options.inf_bound** > 0.0.

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

Input: **inf_step** specifies the magnitude of the change in variables that will be considered a step to an unbounded solution. If the change in x during an iteration would exceed the value of **inf_step**, the objective function is considered to be unbounded below in the feasible region.
Constraint: **options.inf_step** > 0.0.

conf – double * Default memory = **ncnlin**

Input: **ncnlin** values of memory will be automatically allocated by nag_opt_nlin_lsq and this is the recommended method of use of **options.conf**. However a user may supply memory from the calling program.
Output: 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 memory = **ncnlin*n**

Input: **ncnlin*n** values of memory will be automatically allocated by nag_opt_nlin_lsq and this is the recommended method of use of **options.conjac**. However a user may supply memory from the calling program.

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

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

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

Input: **state** need not be set if the default option of **start** = **Nag_Cold** is used as **n+nclin+ncnlin** values of memory will be automatically allocated by nag_opt_nlin_lsq.

If the option **start** = **Nag_Warm** has been chosen, **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_nlin_lsq from the calling program, with **start** = **Nag_Cold** and the same values of **n**, **nclin** and **ncnlin**. If a previous call has not been made, sufficient memory must be allocated by the user.

When a 'warm start' is chosen **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 **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 **state**$[j]$ are as follows:

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_nlin_lsq has been called previously with the same values of **n**, **nclin** and **ncnlin**, then **state** already contains satisfactory information. (See also the description of the optional parameter **start**.) The function also adjusts (if necessary) the values supplied in **x** to be consistent with the values supplied in **state**.

Constraint: $-2 \leq$ **options.state**$[j] \leq 4$, for $j = 0, 1, 2, \ldots,$**n+nclin+ncnlin**-1.

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

state$[j]$	**Meaning**
-2	The constraint violates its lower bound by more than the appropriate feasibility tolerance (see the options **lin_feas_tol** and **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 **lin_feas_tol** and **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 **state** can occur only when **bl**$[j]$ = **bu**$[j]$.

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

Input: **lambda** need not be set if the default option **start** = **Nag_Cold** is used as **n+nclin+ncnlin** values of memory will be automatically allocated by nag_opt_nlin_lsq.

If the option **start** = **Nag_Warm** has been chosen, **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_nlin_lsq from the calling program, with **start** = **Nag_Cold** and the same values of **n**, **nclin** and **ncnlin**. If a previous call has not been made with sufficient memory must be allocated by the user.

When a 'warm start' is chosen **lambda**$[j-1]$ must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by **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 **state** array (i.e., **state**$[j-1] = 1$), **lambda**$[j-1]$ should be zero; if the jth constraint is an inequality active at its lower bound (i.e., **state**$[j-1] = 0$), **lambda**$[j-1]$ should be non-negative; if the jth constraint is an inequality active at its upper bound (i.e., **state**$[j-1] = 2$), **lambda**$[j-1]$ should be non-positive. If necessary, the function will modify lambda to match these rules.
Output: the values of the Lagrange multipliers from the last QP subproblem. **lambda**$[j-1]$ should be non-negative if **state**$[j-1] = 1$ and non-positive if **state**$[j-1] = 2$.

h – double * Default memory = **n*n**

Input: **h** need not be set if the default option of **start** = **Nag_Cold** is used as **n*n** values of memory will be automatically allocated by nag_opt_nlin_lsq.

If the option **start** = **Nag_Warm** has been chosen, **h** must point to a minimum of **n*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_nlin_lsq with **start** = **Nag_Cold** and the same value of **n**. If a previous call has not been made sufficient memory must be allocated to by the user.

When **start** = **Nag_Warm** is chosen the memory pointed to by **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 **hessian** = **TRUE**, then **h** will already have been set correctly.

Output: if **hessian** = **FALSE**, **h** contains the upper triangular Cholesky factor R of $Q^T\tilde{H}Q$, an estimate of the transformed and re-ordered Hessian of the Lagrangian at x (see (6) in Section 7.1 of the documentation for nag_opt_nlp (e04ucc)).

If **hessian** = **TRUE**, **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 – Boolean Default = **FALSE**

Input: controls the contents of the optional parameter **h** on return from nag_opt_nlin_lsq. nag_opt_nlin_lsq works exclusively with the *transformed* and *re-ordered* Hessian H_Q, and hence extra computation is required to form the Hessian itself. If **hessian** = **FALSE**, **h** contains the Cholesky factor of the transformed and re-ordered Hessian. If **hessian** = **TRUE**, the Cholesky factor of the approximate Hessian itself is formed and stored in **h**. This information is required by nag_opt_nlin_lsq if the next call to nag_opt_nlin_lsq will be made with optional parameter **start** = **Nag_Warm**.

h_unit_init – Boolean Default = **FALSE**

Input: if **h_unit_init** = **FALSE** the initial value of the upper triangular matrix R is set to J^TJ, where J denotes the objective Jacobian matrix $\nabla f(x)$. J^TJ is often a good approximation to the objective Hessian matrix $\nabla^2F(x)$. If **h_unit_init** = **TRUE** then the initial value of R is the unit matrix.

h_reset_freq – Integer Default = 2

Input: this parameter allows the user to reset the approximate Hessian matrix to J^TJ every **h_reset_freq** iterations, where J is the objective Jacobian matrix $\nabla f(x)$.

At any point where there are no nonlinear constraints active and the values of f are small in magnitude compared to the norm of J, J^TJ will be a good approximation to the objective

Hessian matrix $\nabla^2 F(x)$. Under these circumstances, frequent resetting can significantly improve the convergence rate of nag_opt_nlin_lsq.

Resetting is suppressed at any iteration during which there are nonlinear constraints active.

Constraint: **options.h_reset_freq** > 0.

iter – Integer

Output: the number of major iterations which have been performed in nag_opt_nlin_lsq.

nf – Integer

Output: 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.

8.3. Description of Printed Output

The level of printed output can be controlled by the user with the structure members **options.list**, **options.print_deriv**, **options.print_level** and **options.minor_print_level** (see Section 8.2). If **list** = **TRUE** then the parameter values to nag_opt_nlin_lsq are listed, followed by the result of any derivative check if **print_deriv** = **Nag_D_Sum** or **Nag_D_Full**. The printout of results is governed by the values of **print_level** and **minor_print_level**. The default of **print_level** = **Nag_Soln_Iter** and **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_nlin_lsq.

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

When a component derivative check (see **verify_grad** in Section 8.2) is selected the element with the largest relative error is identified for the objective and the constraint Jacobian.

If **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.
`Jacobian value`	the 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 Jacobian element and finite difference approximation are in agreement. If the derivatives are believed to be in error nag_opt_nlin_lsq will exit with **fail.code** set to **NE_DERIV_ERRORS**.

When **print_level** = **Nag_Iter** or **Nag_Soln_Iter** the following line of output 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 7 of the documentation for nag_opt_nlp (e04ucc)). Note that `Mnr` may be greater than the optional parameter **minor_max_iter** (default value = $\max(50{,}3(n+n_L+n_N))$; see Section 8.2) 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 at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see Section 7.3 of the documentation for nag_opt_nlp

(e04ucc)). As the solution is approached, `Merit function` will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by `I` at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty parameters. During a sequence of major iterations with infeasible subproblems, the sequence of `Merit Function` values will decrease monotonically until either a feasible subproblem is obtained or nag_opt_nlin_lsq terminates with the error indicator **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 $\|Z^T g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient (see Section 7.1 of the documentation for nag_opt_nlp (e04ucc)). `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 ($H_Z = Z^T H_{\mathrm{FR}} Z = R_Z^T R_Z$; see (6) and (11) in Section 7.1 and Section 7.2, respectively, of the documentation for nag_opt_nlp (e04ucc)). 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 7.4 of the documentation for nag_opt_nlp (e04ucc)).

`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 non-zero, central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 7.1 of the documentation for nag_opt_nlp (e04ucc)).

`L` is printed if the line search has produced a relative change in x greater than the value defined by the optional parameter **step_limit** (default value = 2.0; see Section 8.2). If this output occurs frequently during later iterations of the run, **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 **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 7.1 of the documentation for nag_opt_nlp (e04ucc)). The value of `Nz` is the number of variables minus the number of constraints in the predicted active set; i.e., `Nz` $= n - ($`Bnd`$+$`Lin`$+$`Nln`$)$.

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

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

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

`Penalty` is the Euclidean norm of the vector of penalty parameters 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 (2)−(4) defined in the description of the optional parameter **optim_tol** in Section 8.2. Each letter is `T` if the test is satisfied, and `F` otherwise. The three tests indicate whether:

(a) the sequence of iterates has converged;

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

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

If any of these indicators is `F` when nag_opt_nlin_lsq terminates with the error indicator **NE_NOERROR**, the user should check the solution carefully.

When **print_level** = **Nag_Soln_Iter_Const** or **Nag_Soln_Iter_Full** more detailed results are given at each iteration. If **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 **print_level** = **Nag_Soln_Iter_Full** then the diagonal elements of the matrix T associated with the TQ factorization (see (5) in Section 7.1 of the documentation for nag_opt_nlp (e04ucc)) of the QP working set and the diagonal elements of R, the triangular factor of the transformed and re-ordered Hessian (see (6) in Section 7.1 of the documentation for nag_opt_nlp (e04ucc)) are also output at each iteration.

When **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_nlin_lsq 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 $j = 1, 2, ..., n$ 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 parameters **lin_feas_tol** and **nonlin_feas_tol** (see Section 8.2), `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 **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**$[j-1] \leq$ **inf_bound**, where **inf_bound** is the optional parameter.)

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

`Lagr Mult` is the value of the Lagrange multiplier for the associated bound constraint. This will be zero if `State` is `FR` unless **bl**$[j-1] \leq -$**inf_bound** and **bu**$[j-1] \geq$ **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**$[j-1] \leq -$**inf_bound** and **bu**$[j-1] \geq$ **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 $j = 1, 2, ..., n_L$ of the linear constraint.

`N Con` gives the name (`N`) and index $(j - n_L)$, for $j = n_L + 1, n_L + 2, ..., n_L + n_N$ 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 **lin_feas_tol** and for nonlinear constraints which violate one of their bounds by more than **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 **minor_print_level**, the user is referred to the documentation for nag_opt_lin_lsq (e04ncc). The option **minor_print_level** in the current document is equivalent to **options.print_level** in the documentation for nag_opt_lin_lsq (e04ncc).

If **options.print_level** = **Nag_NoPrint** then printout will be suppressed; the user can print the final solution when nag_opt_nlin_lsq returns to the calling program.

8.3.1. Output of Results via a User-defined Printing Function

Users may also specify their 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 by users who only 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_nlin_lsq. 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 **print_fun**.

If **comm->it_maj_prt** = **TRUE** then results from the last major iteration of nag_opt_nlin_lsq are provided through **st**. Note that **print_fun** will be called with **comm->it_maj_prt** = **TRUE** only if **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
$\|Z^T g_{\mathrm{FR}}\|$, the Euclidean norm of the projected gradient.

nz – Integer
the number of columns of Z (see Section 7.1 of the documentation for nag_opt_nlp (e04ucc)).

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 parameters 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 – Boolean
TRUE if the sequence of iterates has converged, i.e., convergence condition (2) (see description of **options.optim_tol** in Section 8.2) is satisfied.

norm_gz_small – Boolean
TRUE if the projected gradient is sufficiently small, i.e., convergence condition (3) (see description of **options.optim_tol** in Section 8.2) is satisfied.

violtn_small – Boolean
TRUE if the violations of the nonlinear constraints are sufficiently small, i.e., convergence condition (4) (see description of **options.optim_tol** in Section 8.2) is satisfied.

update_modified – Boolean
TRUE if the quasi-Newton update was modified to ensure that the Hessian is positive-definite.

qp_not_feasible – Boolean
TRUE if the QP subproblem has no feasible point.

c_diff – Boolean
TRUE if central differences were used to compute the unspecified objective and constraint gradients.

step_limit_exceeded – Boolean
TRUE if the line search produced a relative change in x greater than the value defined by the optional parameter **step_limit**.

refactor – Boolean
TRUE if the approximate Hessian has been refactorized.

x – double *
contains the components **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 8.2 for a description of the possible status values.

ax – double *
if **st->nclin** > 0, **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, **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** = **TRUE** then the final result from nag_opt_nlin_lsq is provided through **st**. Note that **print_fun** will be called with **comm->sol_sqp_prt** = **TRUE** only if **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 8.2 for a description of the possible status values.

ax – double $*$
if **st->nclin** > 0, $\mathbf{ax}[j-1]$ contains the final value of the jth linear constraint, for $j = 1, 2, \ldots,$**st->nclin**.

cx – double $*$
if **st->ncnlin** > 0, $\mathbf{cx}[j-1]$ 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** = **TRUE** then the results from derivative checking are provided through **st**. Note that **print_fun** will be called with **comm->g_prt** only if **print_deriv** = **Nag_D_Sum** or **Nag_D_Full**. The following members of **st** are set:

m – Integer
the number of subfunctions.

n – Integer
the number of variables.

ncnlin – Integer
the number of nonlinear constraints.

x – double $*$
contains the components $\mathbf{x}[j-1]$ of the initial point x_0, for $j = 1, 2, \ldots,$**st->n**.

fjac – double $*$
contains elements of the Jacobian of F at the initial point x_0 ($\partial f_i/\partial x_j$ is held at location **fjac**$[(i-1)*$**st->tdfjac**$+j-1]$, $i = 1, 2, \ldots,$**st->m**, $j = 1, 2, \ldots,$**st->n**).

tdfjac – Integer
the trailing dimension of **fjac**.

conjac – double $*$
contains the elements of the Jacobian matrix of nonlinear constraints at the initial point x_0 ($\partial c_i/\partial x_j$ is held at location **conjac**$[(i-1)*$**st->n**$+j-1]$, $i = 1, 2, \ldots,$**st->ncnlin**, $j = 1, 2, \ldots,$**st->n**).

In this case the details of any derivative check performed by nag_opt_nlin_lsq are held in the following substructure of **st**:

gprint – Nag_GPrintSt $*$
which in turn contains three substructures **g_chk**, **f_sim**, **c_sim** and two pointers to arrays of substructures, **f_comp** and **c_comp**.

g_chk – Nag_Grad_Chk_St
the substructure **g_chk** contains the members:

type – Nag_GradChk
the type of derivative check performed by nag_opt_nlin_lsq. This will be the same value as in **options.verify_grad**.

g_error – int

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 column of the objective Jacobian at which any component check started. This value will be equal to **options.obj_check_start**.

obj_stop – Integer

specifies the column of the objective Jacobian at which any component check 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, **gprint->g_chk.type** = **Nag_SimpleCheck**, will be held in this substructure in members:

n_elements – Integer

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

correct – Boolean

if **TRUE** then the objective 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 subfunction gradient and its finite difference approximation.

max_subfunction – Integer

the subfunction which has the maximum error between its norm and its finite difference approximation.

c_sim – Nag_SimSt

The result of a simple derivative check of the constraint Jacobian, **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 – Boolean

if **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 Jacobian of the objective function subfunctions, **st->gprint.g_chk.type** = **Nag_CheckObj** or **Nag_CheckObjCon**, will be held in the array of **st->m**$*$**st->n** substructures of type **Nag_CompSt** pointed to by **f_comp**. The element **st->gprint.f_comp**$[(i-1)*$**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**; $j = 1, 2, \ldots,$**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).)

correct – Boolean
: if **TRUE** then this gradient element is consistent with its finite difference approximation.

hopt – double
: the optimal finite difference interval. This is `dx[i]` in the default derivative checking printout (see Section 8.3).

gdiff – double
: the finite difference approximation for this 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 **c_comp**. The element **st->gprint.f_comp**$[(i-1)*$**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**; $j = 1, 2, \ldots,$**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 **c_comp** are as for **f_comp**.

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

g_prt – Boolean
: will be **TRUE** only when the print function is called with the result of the derivative check of **objfun** and **confun**.

it_maj_prt – Boolean
: will be **TRUE** when the print function is called with information about the current major iteration.

sol_sqp_prt – Boolean
: will be **TRUE** when the print function is called with the details of the final solution.

it_prt – Boolean
: will be **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 – Boolean
: will be **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 – Boolean
: will be **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 by the user either before entry to nag_opt_nlin_lsq or during a call to **objfun**, **confun** or **print_fun**. The type Pointer is `void *`.

9. Error Indications and Warnings

NE_USER_STOP

User requested termination, user flag value = ⟨*value*⟩.

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

NE_INT_OPT_ARG_LT

On entry, **options.obj_check_start** = ⟨*value*⟩.
Constraint: **options.obj_check_start** ≥ 1.

On entry, **options.obj_check_stop** = ⟨*value*⟩.
Constraint: **options.obj_check_stop** ≥ 1.

On entry, **options.con_check_start** = ⟨*value*⟩.
Constraint: **options.con_check_start** ≥ 1.

On entry, **options.con_check_stop** = ⟨*value*⟩.
Constraint: **options.con_check_stop** ≥ 1.

NE_INT_OPT_ARG_GT

On entry, **options.obj_check_start** = ⟨*value*⟩.
Constraint: **options.obj_check_start** $\leq$ **n**.

On entry, **options.obj_check_stop** = ⟨*value*⟩.
Constraint: **options.obj_check_stop** $\leq$ **n**.

On entry, **options.con_check_start** = ⟨*value*⟩.
Constraint: **options.con_check_start** $\leq$ **n**.

On entry, **options.con_check_stop** = ⟨*value*⟩.
Constraint: **options.con_check_stop** $\leq$ **n**.

NE_2_INT_OPT_ARG_CONS

On entry, **options.con_check_start** = ⟨*value*⟩ while **options.con_check_stop** = ⟨*value*⟩.
Constraint: **options.con_check_start** $\leq$ **options.con_check_stop**.

On entry, **options.obj_check_start** = ⟨*value*⟩ while **options.obj_check_stop** = ⟨*value*⟩.
Constraint: **options.obj_check_start** $\leq$ **options.obj_check_stop**.

NE_INT_ARG_LT

On entry, **m** must not be less than 1: **m** = ⟨*value*⟩.
On entry, **n** must not be less than 1: **n** = ⟨*value*⟩.
On entry, **nclin** must not be less than 0: **nclin** = ⟨*value*⟩.
On entry, **ncnlin** must not be less than 0: **ncnlin** = ⟨*value*⟩.

NE_2_INT_ARG_LT

On entry, **tda** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These parameters must satisfy **tda** $\geq$ **n**.

NE_OPT_NOT_INIT

Options structure not initialized.

NE_BAD_PARAM

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

NE_INVALID_INT_RANGE_1

Value ⟨*value*⟩ given to **options.h_reset_freq** not valid. Correct range is **h_reset_freq** > 0.
Value ⟨*value*⟩ given to **options.max_iter** not valid. Correct range is **max_iter** ≥ 0.
Value ⟨*value*⟩ given to **options.minor_max_iter** not valid. Correct range is **minor_max_iter** ≥ 0.

NE_INVALID_REAL_RANGE_F

Value ⟨*value*⟩ given to **options.step_limit** not valid. Correct range is **step_limit** > 0.0.
Value ⟨*value*⟩ given to **options.inf_bound** not valid. Correct range is **inf_bound** > 0.0.
Value ⟨*value*⟩ given to **options.inf_step** not valid. Correct range is **inf_step** > 0.0.

NE_INVALID_REAL_RANGE_EF

Value ⟨*value*⟩ given to **options.f_prec** not valid. Correct range is $\epsilon \leq$ **f_prec** < 1.0.
Value ⟨*value*⟩ given to **options.optim_tol** not valid. Correct range is **f_prec** $\leq$ **optim_tol** < 1.0.
Value ⟨*value*⟩ given to **options.c_diff_int** not valid. Correct range is $\epsilon \leq$ **c_diff_int** < 1.0.
Value ⟨*value*⟩ given to **options.f_diff_int** not valid. Correct range is $\epsilon \leq$ **f_diff_int** < 1.0.
Value ⟨*value*⟩ given to **options.lin_feas_tol** not valid. Correct range is $\epsilon \leq$ **lin_feas_tol** < 1.0.
Value ⟨*value*⟩ given to **options.nonlin_feas_tol** not valid. Correct range is $\epsilon \leq$ **nonlin_feas_tol** < 1.0.

NE_INVALID_REAL_RANGE_FF

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

NE_BOUND

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

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

NE_ALLOC_FAIL

Memory allocation failed.

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 to the accuracy requested, but the sequence of iterates has not yet converged. nag_opt_nlin_lsq was terminated because no further improvement could be made in the merit function (see Section 7 of the documentation for nag_opt_nlp (e04ucc) for more details).

This value of **fail.code** may occur in several circumstances. The most common situation is that the user asks for a solution with accuracy that is not attainable with the given precision of the problem (as specified by the optional parameter **f_prec** (default value $= \epsilon^{0.9}$, where ϵ is the ***machine precision***; see Section 8.2). 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 10.1 are satisfied then x is likely to be a solution of (1) even if **fail.code** = **NW_NOT_CONVERGED**.

NW_LIN_NOT_FEASIBLE

No feasible point was found for the linear constraints and bounds.

nag_opt_nlin_lsq 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 parameter **lin_feas_tol** (default value $= \sqrt{\epsilon}$, where ϵ is the ***machine precision***; see Section 8.2), or no feasible point could be found in the number of iterations specified by the optional parameter **minor_max_iter** (default value $= \max(50{,}3(n + n_L + n_N))$; see Section 8.2). The user should check that there are no constraint redundancies. If the data for the constraints are accurate only to an absolute precision σ, the user should ensure that the value of the optional parameter **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, **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 4.1). 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. The user should check the validity of constraints with negative values of the optional parameter **state**. If the user is convinced that a feasible point does exist, nag_opt_nlin_lsq should be restarted at a different starting point.

NW_TOO_MANY_ITER

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

The value of the optional parameter **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 **options.max_iter** and rerun nag_opt_nlin_lsq; alternatively, rerun nag_opt_nlin_lsq, setting the optional parameter **start** = **Nag_Warm** to specify the initial working set. If the algorithm seems to be making little or no progress, however, then the user should check for incorrect gradients or ill conditioning as described below 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 terse line; see Section 4.1), it may be worthwhile setting **start** = **Nag_Warm** and calling nag_opt_nlin_lsq from the final point.

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 and the merit function described in Section 7.1 and Section 7.3 of the function documentation for nag_opt_nlp (e04ucc).

This sometimes occurs because an overly stringent accuracy has been requested, i.e., the value of the optional parameter **optim_tol** (default value $= \epsilon_r^{0.8}$, where ϵ_r is the relative precision of $F(x)$; see Section 8.2) is too small. In this case the user should apply the four tests described in Section 10.1 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_nlin_lsq has failed completely to move from the initial point then functions **objfun** and/or **confun** may be incorrect. The user should refer to comments below under **fail.code** = **NE_DERIV_ERRORS** and check the gradients using the optional parameter **verify_grad** (default value = **Nag_Simple_Check**; see Section 8.2). 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 4.1) 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 4.1).

If the condition estimate of the projected Hessian (`Cond Hz`; see Section 4.1) is extremely large, it may be worthwhile rerunning nag_opt_nlin_lsq from the final point with the optional parameter **start** = **Nag_Warm** (see Section 8.2). In this situation, the optional parameters **state** and **lambda** should be left unaltered and R (in optional parameter **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 8.3), it may be helpful to run nag_opt_nlin_lsq with a relaxed value of the optional parameters **lin_feas_tol** and **nonlin_feas_tol** (default values $\sqrt{\epsilon}$, and $\epsilon^{0.33}$ or $\sqrt{\epsilon}$, respectively, where ϵ is the ***machine precision***; see Section 8.2). (Constraint dependencies are often indicated by wide variations in size in the diagonal elements of the matrix T, whose diagonals will be printed if the optional parameter **print_level** = **Nag_Soln_Iter_Full** (default value = **Nag_Soln_Iter**; see Section 8.2)).

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. The user should refer to the printed output to determine which elements are suspected to be in error.

As a first-step, the user 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.

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.

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 parameter **lin_feas_tol** (default value = $\sqrt{\epsilon}$, where ϵ is the ***machine precision***; see Section 8.2)and/or the optional parameter **nonlin_feas_tol** (default value $\epsilon^{0.33}$ or $\sqrt{\epsilon}$; see Section 8.2), 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).

NE_NOT_APPEND_FILE
Cannot open file ⟨*string*⟩ for appending.

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

NE_NOT_CLOSE_FILE
Cannot close file ⟨*string*⟩.

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

10. Further Comments

10.1 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 7.1 of the documentation for nag_opt_nlp (e04ucc)) to the accuracy requested by the optional parameter **optim_tol** (default value = $\epsilon_r^{0.8}$, see Section 8.2).

The user should also examine the printout from nag_opt_nlin_lsq (see Section 4.1) 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.

10.2. 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 8.2).

11. References

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

Gill P E, Murray W, Saunders M A and Wright M H (1983) Documentation of FDCORE and FDCALC *Report SOL 83–6.* Department of Operations Research, Stanford University.

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

12. See Also

nag_opt_lp (e04mfc)
nag_opt_lin_lsq (e04ncc)
nag_opt_qp (e04nfc)
nag_opt_nlp (e04ucc)
nag_opt_init (e04xxc)
nag_opt_read (e04xyc)
nag_opt_free (e04xzc)

13. Example 2

This example illustrates the use of the optional parameter, **print_fun**, which allows the user to customize the output from nag_opt_nlin_lsq. The same problem is solved as in Example 1. The **options** structure is declared and initialized by nag_opt_init (e04xxc). The **print_fun** option is set to the user defined print function `user_print`. This checks the values of various Boolean members of the **comm** structure parameter to determine what type of printout is required, i.e., derivative checking printout if **comm->g_prt** = **TRUE**, major iteration printout if **comm->it_maj_prt** = **TRUE** and final solution printout if **comm->sol_sqp_prt** = **TRUE**. According to which flag, if any, is set, `user_print` then accesses the relevant members of its **st** structure parameter and outputs the information. On return from nag_opt_nlin_lsq, the memory freeing function nag_opt_free (e04xzc) is used to free the memory assigned to the pointers in the options structure. Users should **not** use the standard C function `free()` for this purpose.

13.1. Program Text

```
static void user_print(const Nag_Search_State *st, Nag_Comm *comm)
{
  static char *states[] = {"IL", "IU", "FR", "LL", "UL", "EQ"};
  Integer i, ixl, ixn;
  Nag_SimSt c_sim, f_sim;


  /* Derivative checking (only handle simple checks here ) */
  if (comm->g_prt)
    {
      /* First ensure that a check has been performed */
      if (st->gprint->g_chk.type != Nag_NoCheck)
        {
          Vprintf("\n\nDerivative Checks\n-----------------\n\n");

          /* Objective check */
          f_sim = st->gprint->f_sim;
          Vprintf("Simple check of objective gradients: ");
          if (f_sim.correct)
            Vprintf("OK\n");
          else
            Vprintf("ERROR!\n");

          Vprintf("Max error was %9.2e in subfunction %1ld\n\n",
                  f_sim.max_error, f_sim.max_subfunction);

          /* Similarly for constraints */
          c_sim = st->gprint->c_sim;
          Vprintf("Simple check of constraint gradients: ");
          if (c_sim.correct)
            Vprintf("OK\n");
          else
            Vprintf("ERROR!\n");

          Vprintf("Max error was %9.2e in constraint %1ld\n",
                  c_sim.max_error, c_sim.max_constraint);
        }
    }

  /* Major iteration output */
  if (comm->it_maj_prt)
    {
      if (st->first)
        {
          Vprintf("\nIterations\n----------\n");
          Vprintf("\n   Maj  Mnr    Step   Merit function\n");
        }
      Vprintf(" %4ld %4ld %8.1e  %14.6e\n", st->iter, st->minor_iter,
              st->step, st->merit);
    }

  /* Solution output */
  if (comm->sol_sqp_prt)
```

```
    {
      Vprintf("\nSolution\n--------\n");

      /* Variable results */
      Vprintf("\nVarbl State     Value      Lagr Mult\n");
      for (i = 0; i < st->n; ++i)
        Vprintf("V%4ld  %2s %14.6e %12.4e\n", i+1, states[st->state[i] + 2],
                st->x[i], st->lambda[i]);

      /* Linear constraint results */
      Vprintf("\nL Con State     Value      Lagr Mult\n");
      for (i = st->n; i < st->n + st->nclin; ++i)
        {
          ixl = i - st->n;
          Vprintf("V%4ld  %2s %14.6e %12.4e\n", ixl+1, states[st->state[i] + 2],
                  st->ax[ixl], st->lambda[i]);
        }

      /* Nonlinear constraint results */
      Vprintf("\nN Con State     Value      Lagr Mult\n");
      for (i = st->n + st->nclin; i < st->n + st->nclin + st->ncnlin; ++i)
        {
          ixn = i - st->n - st->nclin;
          Vprintf("V%4ld  %2s %14.6e %12.4e\n", ixn+1, states[st->state[i] + 2],
                  st->cx[ixn], st->lambda[i]);
        }
    }
} /* user_print */

static void ex2(void)
{
  /* Local variables */
  double x[NMAX], a[NCLIN][NMAX];
  double f[MMAX], y[MMAX], fjac[MMAX][NMAX];
  double bl[MAXBND], bu[MAXBND];
  double objf;
  Integer tda, tdfjac;
  Integer i, j, m, n, nclin, ncnlin;
  Nag_E04_Opt options;
  static NagError fail, fail1;

  fail.print = TRUE;
  fail1.print = TRUE;

  Vprintf("\nExample 2: user defined printing option\n");
  Vscanf(" %*[^\n]"); /* Skip heading in data file */

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

  if (m <= MMAX && n <= NMAX && nclin <= NCLIN && ncnlin <= NCNLIN)
    {
      tda = NMAX;
      tdfjac = NMAX;
      /*  Read a, y, bl, bu and x from data file */

      if (nclin > 0)
        {
          Vscanf(" %*[^\n]");
          for (i = 0; i < nclin; ++i)
            for (j = 0; j < n; ++j)
              Vscanf("%lf",&a[i][j]);
        }

      /* Read the y vector of the objective */
      Vscanf(" %*[^\n]");
      for (i = 0; i < m; ++i)
```

```
      Vscanf("%lf",&y[i]);

    /* Read lower bounds */
    Vscanf(" %*[^\n]");
    for (i = 0; i < n + nclin + ncnlin; ++i)
      Vscanf("%lf",&bl[i]);

    /* Read upper bounds */
    Vscanf(" %*[^\n]");
    for (i = 0; i < n + nclin + ncnlin; ++i)
      Vscanf("%lf",&bu[i]);

    /* Read the initial point x */
    Vscanf(" %*[^\n]");
    for (i = 0; i < n; ++i)
      Vscanf("%lf",&x[i]);

    /* Set an option */
    e04xxc(&options);
    options.print_fun = user_print;

    /*  Solve the problem */
    e04unc(m, n, nclin, ncnlin, (double*)a, tda, bl, bu, y, objfun,
           confun, x, &objf, f, (double*)fjac, tdfjac, &options,
           NAGCOMM_NULL, &fail);

    e04xzc(&options, "all", &fail1);
  }

} /* ex2 */
```

13.2. Program Data

```
Data for example 2

Values of m and n
  44  2

Values of nclin and ncnln
  1   1

Linear constraint matrix A
  1.0   1.0

Objective vector y
  0.49  0.49  0.48  0.47  0.48  0.47  0.46  0.46  0.45  0.43  0.45
  0.43  0.43  0.44  0.43  0.43  0.46  0.45  0.42  0.42  0.43  0.41
  0.41  0.40  0.42  0.40  0.40  0.41  0.40  0.41  0.41  0.40  0.40
  0.40  0.38  0.41  0.40  0.40  0.41  0.38  0.40  0.40  0.39  0.39

Lower bounds
  0.4      -4.0      1.0      0.0

Upper bounds
  1.0e+25   1.0e+25  1.0e+25  1.0e+25

Initial estimate of x
  0.4   0.0
```

13.3. Program Results

```
Example 2: user defined printing option

Parameters to e04unc
--------------------

Number of variables...........   2
Linear constraints............   1    Nonlinear constraints.........   1

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..................    FALSE    h_reset_freq............         2
h_unit_init.............     FALSE
f_diff_int.............. Automatic    c_diff_int.............. Automatic
obj_deriv...............      TRUE    con_deriv...............      TRUE
verify_grad....... Nag_SimpleCheck    print_deriv............ Nag_D_Full
print_level......... Nag_Soln_Iter    minor_print_level..... Nag_NoPrint
outfile.................    stdout


Derivative Checks
-----------------

Simple check of objective gradients: OK
Max error was  1.04e-08 in subfunction 3

Simple check of constraint gradients: OK
Max error was  1.89e-08 in constraint 1

Iterations
----------

   Maj  Mnr    Step   Merit function
    0    2  0.0e+00    2.224070e-02
    1    1  1.0e+00    1.455402e-02
    2    1  1.0e+00    1.436491e-02
    3    1  1.0e+00    1.427013e-02
    4    1  1.0e+00    1.422989e-02
    5    1  1.0e+00    1.422983e-02
    6    1  1.0e+00    1.422983e-02

Solution
--------

Varbl State     Value      Lagr Mult
V   1  FR   4.199527e-01   0.0000e+00
V   2  FR   1.284845e+00   0.0000e+00

L Con State     Value      Lagr Mult
V   1  FR   1.704798e+00   0.0000e+00

N Con State     Value      Lagr Mult
V   1  LL  -9.767742e-13   3.3358e-02
```