

NAG C Library Function Document

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 (e04unc) 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,
    const double a[], Integer tda, const double bl[], const double bu[],
    const double y[],
    void (*objfun)(Integer m, Integer n, const double x[], double f[],
        double fjac[], Integer tdfjac, Nag_Comm comm),
    void (*confun)(Integer n, Integer ncnlin, const double needc[],
        const double x[], double conf[], double conjac[], Nag_Comm comm),
    double x[], double *objf, double f[], Integer fjac[], Integer tdfjac,
    Nag_E04_Opt *options, Nag_Comm *comm, NagError *fail)
```

3 Description

nag_opt_nlin_lsq (e04unc) 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:

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

where $F(x)$ (the *objective function*) is a nonlinear function which can be represented as the sum of squares of m subfunctions $(y_1 - f_1(x)), (y_2 - f_2(x)), \dots, (y_m - f_m(x))$, 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 (e04unc) 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 Nag_XCheckObjCon.)

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 $f(x) = (f_1(x), f_2(x), \dots, f_m(x))^T, c(x)$ 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 Nag_XCheckObjCon 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 Nag_XCheckObjCon) should be used to check the calculation of any known gradients.

4 References

- Dennis J E Jr and Moré J J (1977) Quasi-Newton methods, motivation and theory *SIAM Rev.* **19** 46–89
- Dennis J E Jr and Schnabel R B (1981) A new derivation of symmetric positive-definite secant updates *nonlinear programming* (ed O L Mangasarian, R R Meyer and S M Robinson) **4** 167–199 Academic Press
- Dennis J E Jr and Schnabel R B (1983) *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* Prentice–Hall
- Fletcher R (1987) *Practical Methods of Optimization* (2nd Edition) Wiley
- Gill P E, Hammarling S, Murray W, Saunders M A and Wright M H (1986a) Users' guide for LSSOL (Version 1.0) *Report SOL 86-1* Department of Operations Research, Stanford University
- Gill P E, Murray W, Saunders M A and Wright M H (1983a) Documentation for FDCALC and FDCORE *Technical Report SOL 83-6* Stanford University
- Gill P E, Murray W, Saunders M A and Wright M H (1984a) Users' Guide for SOL/QPSOL Version 3.2 *Report SOL 84-5* Department of Operations Research, Stanford University
- Gill P E, Murray W, Saunders M A and Wright M H (1986b) Some theoretical properties of an augmented Lagrangian merit function *Report SOL 86-6R* Department of Operations Research, Stanford University
- Gill P E, Murray W, Saunders M A and Wright M H (1986c) Users' guide for NPSOL (Version 4.0): a Fortran package for nonlinear programming *Report SOL 86-2* Department of Operations Research, Stanford University
- Gill P E, Murray W and Wright M H (1981) *Practical Optimization* Academic Press
- Hock W and Schittkowski K (1981) *Test Examples for Nonlinear Programming Codes. Lecture Notes in Economics and Mathematical Systems* **187** Springer–Verlag
- Powell M J D (1974) Introduction to constrained optimization *Numerical Methods for Constrained Optimization* (ed P E Gill and W Murray) 1–28 Academic Press
- Powell M J D (1983) Variable metric methods in constrained optimization *Mathematical Programming: The State of the Art* (ed A Bachem, M Grötschel and B Korte) 288–311 Springer–Verlag

5 Arguments

- 1: **m** – Integer *Input*
On entry: m , the number of subfunctions associated with $F(x)$.
Constraint: $m > 0$.

- 2: **n** – Integer *Input*
On entry: n , the number of variables.
Constraint: $n > 0$.
- 3: **nclin** – Integer *Input*
On entry: n_L , the number of general linear constraints.
Constraint: $nclin \geq 0$.
- 4: **ncnlin** – Integer *Input*
On entry: n_N , the number of nonlinear constraints.
Constraint: $ncnlin \geq 0$.
- 5: **a[nclin × tda]** – const double *Input*
On entry: the i th row of **a** must contain the coefficients of the i th general linear constraint (the i th row of the matrix A_L in (1)), for $i = 1, 2, \dots, n_L$.
 If **nclin** = 0 then the array **a** is not referenced.
- 6: **tda** – Integer *Input*
On entry: the second dimension of the array **a** as declared in the function from which `nag_opt_nlin_lsq` (e04unc) is called.
Constraint: if **nclin** > 0, **tda** $\geq n$.
- 7: **bl[n + nclin + ncnlin]** – const double *Input*
 8: **bu[n + nclin + ncnlin]** – const double *Input*
- On entry:* **bl** must contain the lower bounds and **bu** the upper bounds, for all the constraints in the following order. The first n elements of each array must contain the bounds on the variables, the next n_L elements the bounds for the general linear constraints (if any), and the next n_N elements the bounds for the nonlinear constraints (if any). To specify a non-existent lower bound (i.e., $l_j = -\infty$), set **bl**[$j - 1$] $\leq -\mathbf{inf_bound}$, and to specify a non-existent upper bound (i.e., $u_j = +\infty$), set **bu**[$j - 1$] $\geq \mathbf{inf_bound}$, where **inf_bound** is one of the optional parameters (default value 10^{20} , see Section Nag_XCheckObjCon). To specify the j th constraint as an equality, set **bl**[$j - 1$] = **bu**[$j - 1$] = β , say, where $|\beta| < \mathbf{inf_bound}$.
- Constraints:*
- $$\begin{aligned} \mathbf{bl}[j] &\leq \mathbf{bu}[j], \text{ for } j = 0, 1, \dots, n + nclin + ncnlin - 1; \\ \text{if } \mathbf{bl}[j] &= \mathbf{bu}[j] = \beta, |\beta| < \mathbf{inf_bound}. \end{aligned}$$
- 9: **y[m]** – const double *Input*
On entry: the coefficients of the constant vector y in the objective function.
- 10: **objfun** – function, supplied by the user *External Function*
- objfun** must calculate the vector $f(x)$ of subfunctions and (optionally) its Jacobian ($= \frac{\partial f}{\partial x}$) for a specified n element vector x .
- Its specification is:

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

1:	m – Integer	<i>Input</i>
	<i>On entry:</i> <i>m</i> , the number of subfunctions.	
2:	n – Integer	<i>Input</i>
	<i>On entry:</i> <i>n</i> , the number of variables.	
3:	x[n] – const double	<i>Input</i>
	<i>On entry:</i> <i>x</i> , the vector of variables at which $f(x)$ and/or all available elements of its Jacobian are to be evaluated.	
4:	f[m] – double	<i>Output</i>
	<i>On exit:</i> if comm → flag = 0 or 2, objfun must set f [<i>i</i> – 1] to the value of the <i>i</i> th subfunction f_i at the current point <i>x</i> , for some or all $i = 1, 2, \dots, m$ (see the description of the parameter comm → needf below).	
5:	fjac[m × tdfjac] – double	<i>Output</i>
	<i>On exit:</i> if comm → flag = 2, objfun must contain the available elements of the subfunction Jacobian matrix. fjac [(<i>i</i> – 1) × tdfjac + <i>j</i> – 1] must be set to the value of the first derivative $\frac{\partial f_i}{\partial x_j}$ at the current point <i>x</i> , for $i = 1, 2, \dots, m$; $j = 1, 2, \dots, n$.	
	If the optional parameter obj_deriv = NagTrue (the default), all elements of fjac must be set; if obj_deriv = NagFalse , any available elements of the Jacobian matrix must be assigned to the elements of fjac ; the remaining elements <i>must remain unchanged</i> .	
	Any constant elements of fjac may be assigned once only at the first call to objfun , i.e., when comm → first = NagTrue . This is only effective if the optional parameter obj_deriv = NagTrue .	
6:	tdfjac – Integer	<i>Input</i>
	<i>On entry:</i> the second dimension of the array fjac as declared in the function from which nag_opt_nlin_lsq (e04unc) is called.	
7:	comm – Nag_Comm	
	Pointer to structure of type Nag_Comm ; the following members are relevant to objfun .	
	flag – Integer	<i>Input/Output</i>
	<i>On entry:</i> 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.	
	<i>On exit:</i> if objfun resets comm → flag to some negative number then nag_opt_nlin_lsq (e04unc) will terminate immediately with the error indicator NE_USER_STOP . If fail is supplied to nag_opt_nlin_lsq (e04unc), fail.errnum will be set to the user's setting of comm → flag .	
	first – Nag_Boolean	<i>Input</i>
	<i>On entry:</i> will be set to Nag_True on the first call to objfun and Nag_False for all subsequent calls.	
	nf – Integer	<i>Input</i>
	<i>On entry:</i> 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

On entry: if **needf** = 0, **objfun** must set, for all $i = 1, 2, \dots, m$, **f**[$i - 1$] to the value of the i th subfunction f_i at the current point x . If **needf** = i , for $i = 1, 2, \dots, m$, then it is sufficient to set **f**[$i - 1$] to the value of the i th subfunction f_i . Appropriate use of **needf** can save a lot of computational work in some cases. Note that when **comm** → **needf** ≠ 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 will be void * with a C compiler that defines void * and char * otherwise.

Before calling nag_opt_nlin_lsqr (e04unc) these pointers may be allocated memory by the user and initialized with various quantities for use by **objfun** when called from nag_opt_nlin_lsqr (e04unc).

Note: **objfun** should be tested separately before being used in conjunction with nag_opt_nlin_lsqr (e04unc). 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 **Nag_False**.

11: **confun** – function, supplied by the user

External Function

confun must calculate the vector $c(x)$ of nonlinear constraint functions and (optionally) its Jacobian $(= \frac{\partial c}{\partial x})$ for a specified n element vector x . If there are no nonlinear constraints (i.e., **ncnlin** = 0), **confun** will never be called and the NAG defined null void function pointer, NULLFN, can be supplied in the call to nag_opt_nlin_lsqr (e04unc). If there are nonlinear constraints the first call to **confun** will occur before the first call to **objfun**.

Its specification is:

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

1: **n** – Integer

Input

On entry: n , the number of variables.

2: **ncnlin** – Integer

Input

On entry: n_N , the number of nonlinear constraints.

3: **needc**[**ncnlin**] – const double

Input

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

4: **x**[**n**] – const double

Input

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

- 5: **conf**[**ncnlin**] – double *Output*
On exit: if **needc**[$i - 1$] > 0 and **comm** → **flag** = 0 or 2, **conf**[$i - 1$] must contain the value of the i th constraint at x . The remaining elements of **conf**, corresponding to the non-positive elements of **needc**, are ignored.
- 6: **conjac**[**ncnlin** × **n**] – double *Output*
On exit: if **needc**[$i - 1$] > 0 and **comm** → **flag** = 2, the i th row of **conjac** (i.e., the elements **conjac**[($i - 1$) × **n** + $j - 1$], $j = 1, 2, \dots, n$) must contain the available elements of the vector ∇c_i given by
- $$\nabla c_i = \left(\frac{\partial c_i}{\partial x_1}, \frac{\partial c_i}{\partial x_2}, \dots, \frac{\partial c_i}{\partial x_n} \right)^T,$$
- where $\frac{\partial c_i}{\partial x_j}$ is the partial derivative of the i th constraint with respect to the j th 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** = **NagTrue** (the default), all elements of **conjac** must be set; if **con_deriv** = **NagFalse**, 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** = **NagTrue**; see Section Nag_XCheckObjCon), 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** = **NagTrue**.
- It must be emphasized that, if **con_deriv** = **NagFalse**, 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 Nag_XCheckObjCon), an interval for each element of x is computed automatically at the start of the optimization. The automatic procedure can usually identify constant elements of **conjac**, which are then computed once only by finite differences.
- 7: **comm** – Nag_Comm
 Pointer to structure of type **Nag_Comm**; the following members are relevant to **confun**.
- flag** – Integer *Input/Output*
On entry: **confun** is called with **comm** → **flag** set to 0 or 2.
 If **comm** → **flag** = 0, only **conf** is referenced.
 If **comm** → **flag** = 2, both **conf** and **conjac** are referenced.
On exit: if **confun** resets **comm** → **flag** to some negative number then **nag_opt_nlin_lsq** (e04unc) will terminate immediately with the error indicator **NE_USER_STOP**. If **fail** is supplied to **nag_opt_nlin_lsq** (e04unc), **fail.errnum** will be set to the user's setting of **comm** → **flag**.
- first** – Nag_Boolean *Input*
On entry: will be set to **Nag_True** on the first call to **confun** and **Nag_False** for all subsequent calls.

user – double *
iuser – Integer *
p – Pointer

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

Before calling nag_opt_nlin_lsq (e04unc) 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 (e04unc).

Note: **confun** should be tested separately before being used in conjunction with nag_opt_nlin_lsq (e04unc). 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 **Nag_False**.

- 12: **x[n]** – double *Input/Output*
On entry: an initial estimate of the solution.
On exit: the final estimate of the solution.
- 13: **objf** – double * *Output*
On exit: the value of the objective function at the final iterate.
- 14: **f[m]** – double *Output*
On exit: the values of the subfunctions f_i , for $i = 1, 2, \dots, m$, at the final iterate.
- 15: **fjac[m × tdfjac]** – Integer *Output*
On exit: the Jacobian matrix of the functions f_1, f_2, \dots, f_m at the final iterate, i.e., **fjac**[$i - 1$][$j - 1$] contains the partial derivative of the i th subfunction with respect to the j th variable, for $i = 1, 2, \dots, m$; $j = 1, 2, \dots, n$. (See also the discussion of parameter **fjac** under **objfun**.)
- 16: **tdfjac** – Integer *Input*
On entry: the second dimension of the array **fjac** as declared in the function from which nag_opt_nlin_lsq (e04unc) is called.
Constraint: **tdfjac** ≥ **n**.
- 17: **options** – Nag_E04_Opt * *Input/Output*
On entry/on exit: a pointer to a structure of type **Nag_E04_Opt** whose members are optional parameters for nag_opt_nlin_lsq (e04unc). 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 11. Some of the results returned in **options** can be used by nag_opt_nlin_lsq (e04unc) to perform a ‘warm start’ (see the member **start** in Section Nag_XCheckObjCon).
 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 (e04unc). However, if the optional parameters are not required the NAG defined null pointer, E04_DEFAULT, can be used in the function call.
- 18: **comm** – Nag_Comm * *Input/Output*
On entry/on exit: structure containing pointers for communication to the user-supplied functions **objfun** and **confun**, and the optional user-defined printing function; see the description of **objfun** and **confun** and Section 11.3.1 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` (e04unc); `comm` will then be declared internally for use in calls to user-supplied functions.

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

The NAG error parameter, see the Essential Introduction.

5.1 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 `print_level` and `minor_print_level` (see Section Nag_XCheckObjCon). 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` (e04unc).

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 10).
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 Nag_XCheckObjCon) 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 10.3). As the solution is approached, Merit function will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by I at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty 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` (e04unc) 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 $\|Z^T g_{FR}\|$, the Euclidean norm of the projected gradient (see Section 10.1). **Norm Gz** will be approximately zero in the neighbourhood of a solution.

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

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

M is printed if the quasi-Newton update was modified to ensure that the Hessian approximation is positive-definite (see Section 10.4).

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

- C is printed if central differences were used to compute the unspecified objective and constraint gradients. If the value of `Step` is zero, the switch to central differences was made because no lower point could be found in the line search. (In this case, the QP subproblem is re-solved with the central difference gradient and Jacobian.) If the value of `Step` is non-zero, central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 10.1).
- 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 Nag_XCheckObjCon). 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 $j = 1, 2, \dots, n$ 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 Nag_XCheckObjCon), `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 $\mathbf{bl}[j - 1] \leq \mathbf{inf_bound}$, where `inf_bound` is the optional parameter.)
- `Upper bound` is the upper bound specified for the variable j . (None indicates that $\mathbf{bu}[j - 1] \geq \mathbf{inf_bound}$, 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 $\mathbf{bl}[j - 1] \leq -\mathbf{inf_bound}$ and $\mathbf{bu}[j - 1] \geq \mathbf{inf_bound}$, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is LL, and non-positive if `State` is UL.
- `Residual` is the difference between the variable `Value` and the nearer of its (finite) bounds $\mathbf{bl}[j - 1]$ and $\mathbf{bu}[j - 1]$. A blank entry indicates that the associated variable is not bounded (i.e., $\mathbf{bl}[j - 1] \leq -\mathbf{inf_bound}$ and $\mathbf{bu}[j - 1] \geq \mathbf{inf_bound}$).

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

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

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

On entry, **tda** = $\langle value \rangle$ while **n** = $\langle value \rangle$. These parameters must satisfy **tda** \geq **n**. This error message is output only if **nclin** $>$ 0.

NE_2_INT_OPT_ARG_CONS

On entry, **con_check_start** = $\langle value \rangle$ while **con_check_stop** = $\langle value \rangle$.
Constraint: **con_check_start** \leq **con_check_stop**.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, parameter **print_level** had an illegal value.

NE_BOUND

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

NE_BOUND_EQ

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

NE_BOUND_EQ_LCON

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

NE_BOUND_EQ_NLCON

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

NE_BOUND_LCON

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

NE_BOUND_NLCON

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

NE_DERIV_ERRORS

Large errors were found in the derivatives of the objective function and/or nonlinear constraints.

NE_INT_ARG_LT

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

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.

NE_INT_OPT_ARG_GT

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

NE_INT_OPT_ARG_LT

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

NE_INVALID_INT_RANGE_1

Value $\langle value \rangle$ given to **h_reset_freq** not valid. Correct range is **h_reset_freq** $>$ 0.

NE_INVALID_REAL_RANGE_EF

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

NE_INVALID_REAL_RANGE_F

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

NE_INVALID_REAL_RANGE_FF

Value $\langle value \rangle$ given to **linesearch_tol** not valid. Correct range is $0.0 \leq$ **linesearch_tol** $<$ 1.0.

NE_NOT_APPEND_FILE

Cannot open file $\langle string \rangle$ for appending.

NE_NOT_CLOSE_FILE

Cannot close file $\langle string \rangle$.

NE_OPT_NOT_INIT

Options structure not initialized.

NE_STATE_VAL

state[$\langle value \rangle$] is out of range. **state**[$\langle value \rangle$] = $\langle value \rangle$.

NE_USER_STOP

User requested termination, user flag value = $\langle value \rangle$.

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.

NW_LIN_NOT_FEASIBLE

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

NW_NONLIN_NOT_FEASIBLE

No feasible point could be found for the nonlinear constraints.

NW_NOT_CONVERGED

Optimal solution found, but the sequence of iterates has not converged with the requested accuracy.

NW_OVERFLOW_WARN

Serious ill conditioning in the working set after adding constraint $\langle value \rangle$. Overflow may occur in subsequent iterations.

NW_TOO_MANY_ITER

The maximum number of iterations, $\langle value \rangle$, have been performed.

7 Accuracy

If **fail.code** = **NE_NOERROR** on exit, then the vector returned in the array **x** is an estimate of the solution to an accuracy of approximately **optim_tol** (default value = $\epsilon_r^{0.8}$, where ϵ_r is the relative precision of $F(x)$; see Section Nag_XCheckObjCon).

8 Further Comments**8.1 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 10.1) to the accuracy requested by the optional parameter **optim_tol** (default value = $\epsilon_r^{0.8}$, see Section Nag_XCheckObjCon).

The user should also examine the printout from `nag_opt_nlin_lsq` (e04unc) (see Section 5.1) 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.

9 Example

There is one example program file, the main program of which calls both examples EX1 and EX2. Example 1 (EX1) shows the simple use of `nag_opt_nlin_lsq` (e04unc) where default values are used for all optional parameters. An example showing the use of optional parameters is given in Example 2 (EX2) and is described in Section 12.

Example 1 (EX1)

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

$$\begin{aligned}x_1 &\geq 0.4 \\x_2 &\geq -4.0\end{aligned}$$

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.

9.1 Program Text

```
/* nag_opt_nlin_lsq (e04unc) Example Program.
 *
 * Copyright 1998 Numerical Algorithms Group.
 *
 * Mark 5, 1998.
 *
 * Mark 6 revised, 2000.
 * Mark 7 revised, 2001.
 * Mark 8 revised, 2004.
 *
 */
#include <nag.h>
```

```

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

#ifdef __cplusplus
extern "C" {
#endif
    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);
#ifdef __cplusplus
}
#endif
static int ex1(void);
static int 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)
    {
        ai = a[i];
        temp = exp(-x1 * (ai - 8.0));
        /* Evaluate objective subfunction f(i+1) only if required */
        if (comm->needf == i+1 || comm->needf == 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 == Nag_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] = -0.09 - x[0]*x[1] + 0.49*x[1];

        if (comm->flag == 2)
        {
            CJAC(0,0) = -x[1];
            CJAC(0,1) = -x[0] + 0.49;
        }
    }
} /* confun */

int main(void)
{
    Integer exit_status_ex1=0;
    Integer exit_status_ex2=0;

    Vprintf("nag_opt_nlin_lsq (e04unc) Example Program Results\n");

    exit_status_ex1 = ex1();
    exit_status_ex2 = ex2();

    return exit_status_ex1 == 0 && exit_status_ex2 == 0 ? 0 : 1;
}

#undef FJAC
#define A(I,J) a[(I)*tda + J]
#define FJAC(I,J) fjac[(I)*tdfjac + J]

static int ex1(void)
{
    /* Local variables */
    Integer exit_status=0, i, j, m, n, nbnd, nclin, ncnlin, tda, tdfjac;
    NagError fail;
    double *a=0, *bl=0, *bu=0, *f=0, *fjac=0, objf, *x=0, *y=0;

    INIT_FAIL(fail);

    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>0 && n>0 && nclin>=0 && ncnlin>=0)
    {
        nbnd = n + nclin + ncnlin;
        if ( !( x = NAG_ALLOC(n, double)) ||
            !( a = NAG_ALLOC(nclin*n, double)) ||
            !( f = NAG_ALLOC(m, double)) ||
            !( y = NAG_ALLOC(m, double)) ||
            !( fjac = NAG_ALLOC(m*n, double)) ||
            !( bl = NAG_ALLOC(nbnd, double)) ||
            !( bu = NAG_ALLOC(nbnd, double)) )
        {
            Vprintf("Allocation failure\n");
            exit_status = -1;
            goto END;
        }
        tda = n;
        tdfjac = n;
    }
}

```

```

else
{
    Vprintf("Invalid m or n or nclin or ncnlin.\n");
    exit_status = 1;
    return exit_status;
}
/* 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 */
/* nag_opt_nlin_lsq (e04unc).
 * Solves nonlinear least-squares problems using the
 * sequential QP method
 */
nag_opt_nlin_lsq(m, n, nclin, ncnlin, a, tda, bl, bu, y, objfun,
                confun, x, &objf, f, fjac, tdfjac, E04_DEFAULT,
                NAGCOMM_NULL, &fail);
if (fail.code != NE_NOERROR)
{
    Vprintf("Error from nag_opt_nlin_lsq (e04unc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
END:
if (x) NAG_FREE(x);
if (a) NAG_FREE(a);
if (f) NAG_FREE(f);
if (y) NAG_FREE(y);
if (fjac) NAG_FREE(fjac);
if (bl) NAG_FREE(bl);
if (bu) NAG_FREE(bu);
return exit_status;
} /* ex1 */
static void user_print(const Nag_Search_State *st, Nag_Comm *comm)
{
    static const 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->q_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 %lld\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 %lld\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 int ex2(void)
{
    /* Local variables */
    Integer exit_status=0, i, j, m, n, nbnd, nclin, ncnlin, tda, tdfjac;
    NagError fail;
    Nag_E04_Opt options;
    double *a=0, *bl=0, *bu=0, *f=0, *fjac=0, objf, *x=0, *y=0;

    INIT_FAIL(fail);

    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>0 && n>0 && nclin>=0 && ncnlin>=0)
    {
        nbnd = n + nclin + ncnlin;
        if ( !( x = NAG_ALLOC(n, double)) ||
            !( a = NAG_ALLOC(nclin*n, double)) ||
            !( f = NAG_ALLOC(m, double)) ||
            !( y = NAG_ALLOC(m, double)) ||
            !( fjac = NAG_ALLOC(m*n, double)) ||
            !( bl = NAG_ALLOC(nbnd, double)) ||
            !( bu = NAG_ALLOC(nbnd, double)) )
        {
            Vprintf("Allocation failure\n");
            exit_status = -1;
            goto END;
        }
        tda = n;
        tdfjac = n;
    }
    else
    {
        Vprintf("Invalid m or n nclin or ncnlin.\n");
        exit_status = 1;
        return exit_status;
    }
    /* 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 */
/* nag_opt_init (e04xxc).
 * Initialization function for option setting
 */
nag_opt_init(&options);
options.print_fun = user_print;

/* Solve the problem */
/* nag_opt_nlin_lsq (e04unc), see above. */
nag_opt_nlin_lsq(m, n, nclin, ncnln, a, tda, bl, bu, y, objfun,
                confun, x, &objf, f, fjac, tdfjac, &options,
                NAGCOMM_NULL, &fail);
if (fail.code != NE_NOERROR)
{
    Vprintf("Error from nag_opt_nlin_lsq (e04unc).\n%s\n", fail.message);
    exit_status = 1;
}
/* nag_opt_free (e04xzc).
 * Memory freeing function for use with option setting
 */
nag_opt_free(&options, "all", &fail);
if (fail.code != NE_NOERROR)
{
    Vprintf("Error from nag_opt_free (e04xzc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
}
END:
if (x) NAG_FREE(x);
if (a) NAG_FREE(a);
if (f) NAG_FREE(f);
if (y) NAG_FREE(y);
if (fjac) NAG_FREE(fjac);
if (bl) NAG_FREE(bl);
if (bu) NAG_FREE(bu);
return exit_status;
} /* ex2 */

```

9.2 Program Data

nag_opt_nlin_lsq (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

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

```

9.3 Program Results

nag_opt_nlin_lsq (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..... 4.38e-15
inf_bound..... 1.00e+20  inf_step..... 1.00e+20
max_iter..... 50  minor_max_iter..... 50
hessian..... Nag_FALSE  h_reset_freq..... 2
h_unit_init..... Nag_FALSE
f_diff_int..... Automatic  c_diff_int..... Automatic
obj_deriv..... Nag_TRUE  con_deriv..... Nag_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 1.06e-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 1.91e-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	-9.76813e-13	0.00000e+00	None	3.3358e-02	-9.7681e-13

Optimal solution found.

Final objective value = 1.4229835e-02

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..... 4.38e-15
inf_bound..... 1.00e+20     inf_step..... 1.00e+20
max_iter..... 50           minor_max_iter..... 50
hessian..... Nag_FALSE     h_reset_freq..... 2
h_unit_init..... Nag_FALSE
f_diff_int..... Automatic
obj_deriv..... Nag_TRUE     c_diff_int..... Automatic
verify_grad..... Nag_TRUE   con_deriv..... Nag_TRUE
print_level..... Nag_D_Full  print_deriv..... Nag_D_Full
outfile..... stdout         minor_print_level..... Nag_NoPrint

```

Derivative Checks

Simple check of objective gradients: OK
Max error was 1.06e-08 in subfunction 3

Simple check of constraint gradients: OK
Max error was 1.91e-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	-9.768135e-13	3.3358e-02	

10 Further Description

This section gives a detailed description of the algorithm used in `nag_opt_nlin_lsq` (e04unc). This, and possibly the next section, Section 11, may be omitted if the more sophisticated features of the algorithm and software are not currently of interest.

10.1 Overview

`nag_opt_nlin_lsq` (e04unc) is based on the same algorithm as used in subroutine NPSOL described in Gill *et al.* (1986c).

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

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

- (i) x is feasible;
- (ii) there exist vectors ξ and λ (*the Lagrange multiplier vectors for the bound and general constraints*) such that

$$g = C^T \lambda + \xi \quad (2)$$

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

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

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

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

The vector $Z^T g_{\text{FR}}$ is termed the *projected gradient* of F at x . Certain additional conditions must be satisfied in order for a first-order Kuhn–Tucker point to be a solution of (1) (see, e.g., Powell (1974)). `nag_opt_nlin_lsq` (e04unc) implements a sequential quadratic programming (SQP) method. For an overview of SQP methods, see, for example, Fletcher (1987), Gill *et al.* (1981) and Powell (1983).

The basic structure of `nag_opt_nlin_lsq` (e04unc) involves *major* and *minor* iterations. The major iterations generate a sequence of iterates $\{x_k\}$ that converge to x^* , a first-order Kuhn–Tucker point of (1). At a typical major iteration, the new iterate \bar{x} is defined by

$$\bar{x} = x + \alpha p \quad (3)$$

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

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

$$\underset{p}{\text{Minimize}} \quad g^T p + \frac{1}{2} p^T H p \quad \text{subject to} \quad \bar{l} \leq \begin{Bmatrix} p \\ A_L p \\ A_N p \end{Bmatrix} \leq \bar{u}, \quad (4)$$

where g is the gradient of F at x , the matrix H is a positive-definite quasi-Newton approximation to the Hessian of the Lagrangian function (see Section 10.4), and A_N is the Jacobian matrix of c evaluated at x . (Finite difference estimates may be used for g and A_N ; see the optional parameters **obj_deriv** and **con_deriv** in Section Nag_XCheckObjCon.) Let l in (1) be partitioned into three sections: l_B , l_L and l_N , corresponding to the bound, linear and nonlinear constraints. The vector \bar{l} in (4) is similarly partitioned, and is defined as

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

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

The estimated Lagrange multipliers at each major iteration are the Lagrange multipliers from the subproblem (4) (and similarly for the predicted active set). (The numbers of bounds, general linear and nonlinear constraints in the QP active set are the quantities `Bnd`, `Lin` and `Nln` in the output of `nag_opt_nlin_lsq` (e04unc); see Section `Itns`.) In `nag_opt_nlin_lsq` (e04unc), (4) is solved using the same algorithm as used in function `nag_opt_lin_lsq` (e04ncc). Since solving a quadratic program is an iterative procedure, the minor iterations of `nag_opt_nlin_lsq` (e04unc) are the iterations of `nag_opt_lin_lsq` (e04ncc). (More details about solving the subproblem are given in Section 10.2.)

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

$$C_{\text{FR}} Q_{\text{FR}} = (0 \quad T), \quad (5)$$

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

$$R^T R = H_Q \equiv Q^T \tilde{H} Q, \quad (6)$$

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

$$Q = \begin{pmatrix} Q_{\text{FR}} & \\ & I_{\text{FX}} \end{pmatrix} \quad (7)$$

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

$$Q_{\text{FR}} = (Z \quad Y),$$

the n_Z ($n_Z \equiv n_{\text{FR}} - m$) columns of Z form a basis for the null space of C_{FR} . The matrix Z is used to compute the projected gradient $Z^T g_{\text{FR}}$ at the current iterate. (The values `Nz`, `Norm Gf` and `Norm Gz` printed by `nag_opt_nlin_lsq` (e04unc) give n_Z and the norms of g_{FR} and $Z^T g_{\text{FR}}$; see Section `Itns`.)

A theoretical characteristic of SQP methods is that the predicted active set from the QP subproblem (4) is identical to the correct active set in a neighbourhood of x^* . In `nag_opt_nlin_lsq` (e04unc), this feature is

exploited by using the QP active set from the previous iteration as a prediction of the active set for the next QP subproblem, which leads in practice to optimality of the subproblems in only one iteration as the solution is approached. Separate treatment of bound and linear constraints in nag_opt_nlin_lsq (e04unc) also saves computation in factorizing C_{FR} and H_Q .

Once p has been computed, the major iteration proceeds by determining a step length α that produces a 'sufficient decrease' in an augmented Lagrangian *merit function* (see Section 10.3). Finally, the approximation to the transformed Hessian matrix H_Q is updated using a modified BFGS quasi-Newton update (see Section 10.4) to incorporate new curvature information obtained in the move from x to \bar{x} .

On entry to nag_opt_nlin_lsq (e04unc), an iterative procedure from nag_opt_lin_lsq (e04ncc) is executed, starting with the user-provided initial point, to find a point that is feasible with respect to the bounds and linear constraints (using the tolerance specified by **lin_feas_tol**; see Section Nag_XCheckObjCon). If no feasible point exists for the bound and linear constraints, (1) has no solution and nag_opt_nlin_lsq (e04unc) terminates. Otherwise, the problem functions will thereafter be evaluated only at points that are feasible with respect to the bounds and linear constraints. The only exception involves variables whose bounds differ by an amount comparable to the finite difference interval (see the discussion of **f_diff_int** in Section Nag_XCheckObjCon). In contrast to the bounds and linear constraints, it must be emphasized that *the nonlinear constraints will not generally be satisfied until an optimal point is reached*.

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

In summary, the method of nag_opt_nlin_lsq (e04unc) first determines a point that satisfies the bound and linear constraints. Thereafter, each iteration includes:

- (a) the solution of a quadratic programming subproblem (see Section 10.2);
- (b) a linesearch with an augmented Lagrangian merit function (see Section 10.3); and
- (c) a quasi-Newton update of the approximate Hessian of the Lagrangian function (Section 10.4).

10.2 Solution of the Quadratic Programming Subproblem

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

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

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

$$\bar{p} = p + \sigma d \quad (8)$$

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

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

Let C denote the sub-matrix of rows of

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

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

$$C_{\text{FR}}d_{\text{FR}} = 0 \quad (9)$$

which is equivalent to defining d_{FR} as

$$d_{\text{FR}} = Zd_Z \quad (10)$$

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

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

$$R_Z^T R_Z d_Z = -Z^T q_{\text{FR}} \quad (11)$$

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

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

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

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

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

10.3 The Merit Function

After computing the search direction as described in Section 10.2, each major iteration proceeds by determining a step length α in (3) that produces a ‘sufficient decrease’ in the augmented Lagrangian merit function

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

where x , λ and s vary during the *linesearch*. The summation terms in (12) involve only the *nonlinear* constraints. The vector λ is an estimate of the Lagrange multipliers for the nonlinear constraints of (1). The non-negative *slack variables* $\{s_i\}$ allow nonlinear inequality constraints to be treated without introducing discontinuities. The solution of the QP subproblem (4) provides a vector triple that serves as a direction of search for the three sets of variables. The non-negative vector ρ of *penalty parameters* is initialized to zero at the beginning of the first major iteration. Thereafter, selected elements are increased

whenever necessary to ensure descent for the merit function. Thus, the sequence of norms of ρ (the printed quantity `Penalty`; see Section `Itns`) is generally non-decreasing, although each ρ_i may be reduced a limited number of times.

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

10.4 The Quasi-Newton Update

The matrix H in (4) is a *positive-definite quasi-Newton* approximation to the Hessian of the Lagrangian function. (For a review of quasi-Newton methods, see Dennis and Schnabel (1983).) At the end of each major iteration, a new Hessian approximation \bar{H} is defined as a rank-two modification of H . In `nag_opt_nlin_lsq` (e04unc), the BFGS quasi-Newton update is used:

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

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

In `nag_opt_nlin_lsq` (e04unc), H is required to be positive-definite. If H is positive-definite, \bar{H} defined by (13) will be positive-definite if and only if $y^T s$ is positive (see, e.g., Dennis and Moré (1977)). Ideally, y in (13) would be taken as y_L , the change in gradient of the Lagrangian function

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

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

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

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

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

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

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

11 Optional Parameters

A number of optional input and output parameters to `nag_opt_nlin_lsq` (e04unc) 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` (e04unc); 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).

11.1 Optional Argument Checklist and Default Values

For easy reference, the following list shows the members of **options** which are valid for `nag_opt_nlin_lsq` (e04unc) together with their default values where relevant. The number ϵ is a generic notation for *machine precision* (see `nag_machine_precision` (X02AJC)).

<code>Nag_Start</code> start	Nag_Cold
Boolean list	Nag_True
<code>Nag_PrintType</code> print_level	Nag_Soln_Iter
<code>Nag_PrintType</code> minor_print_level	Nag_NoPrint
char outfile[80]	stdout
void (*print_fun)()	NULL
Boolean obj_deriv	Nag_True
Boolean con_deriv	Nag_True
<code>Nag_GradChk</code> verify_grad	Nag_SimpleCheck
<code>Nag_DPrintType</code> print_deriv	Nag_D_Full
Integer obj_check_start	1
Integer obj_check_stop	n
Integer con_check_start	1
Integer con_check_stop	n
double f_diff_int	Computed automatically
double c_diff_int	Computed automatically
Integer max_iter	$\max(50, 3(\mathbf{n} + \mathbf{nclin}) + 10\mathbf{ncnlin})$
Integer minor_max_iter	$\max(50, 3(\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}))$
double f_prec	$\epsilon^{0.9}$
double optim_tol	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	$\max(\mathbf{inf_bound}, 10^{20})$
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	Nag_False
Boolean h_unit_init	Nag_False
Integer h_reset_freq	2
Integer iter	
Integer nf	

11.2 Description of the Optional Arguments

– `Nag_Start`

Default = **Nag_Cold**

On entry: specifies how the initial working set is chosen in both the procedure for finding a feasible point for the linear constraints and bounds, and in the first QP subproblem thereafter. With **start** = **Nag_Cold**, `nag_opt_nlin_lsq` (e04unc) 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`

(e04unc) 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` (e04unc) is called repeatedly to solve related problems.

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

– **Nag_Boolean** Default = **NagTrue**

On entry: if **list** = **NagTrue** the parameter settings in the call to `nag_opt_nlin_lsq` (e04unc) will be printed.

– **Nag_PrintType** Default = **Nag_Soln_Iter**

On entry: the level of results printout produced by `nag_opt_nlin_lsq` (e04unc) 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	As Nag_Soln_Iter_Long with the objective function, the values of the variables, the Euclidean norm of the nonlinear constraint violations, the nonlinear constraint values, c , and the linear constraint values A_Lx also printed at each iteration.
Nag_Soln_Iter_Full	As Nag_Soln_Iter_Const with the diagonal elements of the upper triangular matrix T associated with the TQ factorization (see (5)) of the QP working set, and the diagonal elements of R , the triangular factor of the transformed and re-ordered Hessian (see (6)).

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

Constraint: **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**.

– **Nag_PrintType** Default = **Nag_NoPrint**

On entry: the level of results printout produced by the minor iterations of `nag_opt_nlin_lsq` (e04unc) (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_Lx 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) of the document 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 11.3 of the document for `nag_opt_lin_lsq` (e04ncc). (**minor_print_level** in the present function is equivalent to **print_level** in `nag_opt_lin_lsq` (e04ncc).)

Constraint: **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**.

– const char[80] Default = stdout

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

– pointer to function Default = NULL

On entry: 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 11.3.1 below for further details.

– Nag_Boolean Default = NagTrue

On entry: 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 **Nag_False**.

Whenever possible all elements should be supplied, since `nag_opt_nlin_lsq` (e04unc) is more reliable and will usually be more efficient when all derivatives are exact.

If **obj_deriv** = **Nag_False**, `nag_opt_nlin_lsq` (e04unc) 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 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.)

– Nag_Boolean Default = NagTrue

On entry: 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 **Nag_False**.

Whenever possible all elements should be supplied, since `nag_opt_nlin_lsq` (e04unc) is more reliable and will usually be more efficient when all derivatives are exact.

If **con_deriv** = **Nag_False**, `nag_opt_nlin_lsq` (e04unc) 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` (e04unc) 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.)

– Nag_GradChk Default = Nag_SimpleCheck

On entry: specifies the level of derivative checking to be performed by nag_opt_nlin_lsq (e04unc) 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.* (1983a).) The result of the test is printed out by nag_opt_nlin_lsq (e04unc) if the optional parameter **print_deriv** \neq **Nag_D_NoPrint**.

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

– Nag_DPrintType Default = Nag_D_Full

On entry: 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 summarising 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: **print_deriv** = **Nag_D_NoPrint**, **Nag_D_Sum** or **Nag_D_Full**.

– Integer i Default = 1
 – Integer i Default = **n**

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

On entry: 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: $1 \leq \mathbf{obj_check_start} \leq \mathbf{obj_check_stop} \leq \mathbf{n}$.

– Integer i Default = 1
 – Integer i Default = **n**

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

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

Constraint: $1 \leq \mathbf{con_check_start} \leq \mathbf{con_check_stop} \leq \mathbf{n}$.

– double r Default = computed automatically

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

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

In general, using the notation $r = \mathbf{f_diff_int}$, a derivative with respect to the j th 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 $O(r)$. See Chapter 8 of 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** (e04unc) determine constant elements in the objective and constraint gradients (see the descriptions of **confun** and **objfun** in Section 5).

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

– double r Default = computed automatically

On entry: 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 5.1). The use of finite-differences is discussed under the option **f_diff_int**.

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

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

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

Constraint: **max_iter** ≥ 0 .

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

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

Constraint: **minor_max_iter** ≥ 0 .

– double r Default = $\epsilon^{0.9}$

On entry: 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` (e04unc) will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

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

– double r Default = $\mathbf{f_prec}^{0.8}$

On entry: 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` (e04unc) terminates successfully, the final value of F should have approximately six correct figures.

`nag_opt_nlin_lsq` (e04unc) 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 10.1). The sequence of iterates is considered to have converged at x if

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

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

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

and

$$|res_j| \leq ftol \text{ for all } j, \quad (18)$$

where $Z^T g_{FR}$ is the projected gradient (see Section 10.1), g_{FR} is the gradient of $F(x)$ with respect to the free variables, res_j is the violation of the j th active nonlinear constraint, and $ftol$ is the value of the optional parameter **nonlin_feas_tol**.

Constraint: **f_prec** $\leq \mathbf{optim_tol} < 1.0$.

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

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

On entry to `nag_opt_nlin_lsq` (e04unc), 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: $\epsilon \leq \mathbf{lin_feas_tol} < 1.0$.

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

The default is $\epsilon^{0.33}$ if the optional parameter **con_deriv** = **NagFalse**, and $\sqrt{\epsilon}$ otherwise.

On entry: 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 (e04unc) (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: $\epsilon \leq \mathbf{nonlin_feas_tol} < 1.0$.

– double r Default = 0.9

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

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

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

– double r Default = 2.0

On entry: specifies the maximum change in the variables at the first step of the line search. In some cases, such as $F(x) = ae^{bx}$ or $F(x) = ax^b$, even a moderate change in the elements of x can lead to floating-point overflow. The 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

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

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 5.1). 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: **step_limit** > 0.0.

– double r Default = 0.01

On entry: **crash_tol** is used during a ‘cold start’ when nag_opt_nlin_lsq (e04unc) selects an initial working set (**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 $a_j^T x \geq l$ will be included in the initial working set if $|a_j^T x - l| \leq \mathbf{crash_tol} \times (1 + |l|)$.

Constraint: $0.0 \leq \mathbf{crash_tol} \leq 1.0$.

– double r Default = 10^{20}

On entry: **inf_bound** defines the ‘infinite’ bound in the definition of the problem constraints. Any upper bound greater than or equal to **inf_bound** will be regarded as plus infinity (and similarly any lower bound less than or equal to **-inf_bound** will be regarded as minus infinity).

Constraint: **inf_bound** > 0.0.

– double r Default = $\max(\mathbf{inf_bound}, 10^{20})$

On entry: **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: **inf_step** > 0.0.

– double * r Default memory = **nclin**

On entry: **nclin** values of memory will be automatically allocated by nag_opt_nlin_lsqr (e04unc) and this is the recommended method of use of **conf**. However a user may supply memory from the calling program.

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

If **nclin** = 0 then **conf** will not be referenced.

– double * r Default memory = **nclin** × **n**

On entry: **nclin** × **n** values of memory will be automatically allocated by nag_opt_nlin_lsqr (e04unc) and this is the recommended method of use of **conjac**. However a user may supply memory from the calling program.

On exit: if **nclin** > 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 i th constraint function with respect to the j th variable, for $i = 1, 2, \dots, \mathbf{nclin}$; $j = 1, 2, \dots, \mathbf{n}$. (See the discussion of the parameter **conjac** under **confun**.)

If **nclin** = 0 then **conjac** will not be referenced.

– Integer * i Default memory = **n** + **nclin** + **nclin**

On entry: **state** need not be set if the default option of **start** = **Nag_Cold** is used as **n** + **nclin** + **nclin** values of memory will be automatically allocated by nag_opt_nlin_lsqr (e04unc).

If the option **start** = **Nag_Warm** has been chosen, **state** must point to a minimum of **n** + **nclin** + **nclin** elements of memory. This memory will already be available if the **options** structure has been used in a previous call to nag_opt_nlin_lsqr (e04unc) from the calling program, with **start** = **Nag_Cold** and the same values of **n**, **nclin** and **nclin**. 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 **nclin** elements refer to the nonlinear constraints. Possible values for **state**[j] are as follows:

state [j]	Meaning
0	The corresponding constraint is <i>not</i> 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 $\mathbf{bl}[j] = \mathbf{bu}[j]$.

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 \mathbf{bl} and \mathbf{bu} are not equal. If `nag_opt_nlin_lsq` (e04unc) has been called previously with the same values of \mathbf{n} , \mathbf{nclin} and \mathbf{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 \mathbf{x} to be consistent with the values supplied in `state`.

Constraint: $-2 \leq \mathbf{state}[j] \leq 4$, for $j = 0, 1, \dots, \mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin} - 1$.

On exit: the status of the constraints in the QP working set at the point returned in \mathbf{x} . The significance of each possible value of `state[j]` is as follows:

<code>state[j]</code>	Meaning
-2	The constraint violates its lower bound by more than the appropriate feasibility tolerance (see the options <code>lin_feas_tol</code> and <code>nonlin_feas_tol</code>). 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 <code>lin_feas_tol</code> and <code>nonlin_feas_tol</code>). 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 <code>state</code> can occur only when $\mathbf{bl}[j] = \mathbf{bu}[j]$.

– double * r Default memory = $\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}$

On entry: `lambda` need not be set if the default option `start = Nag_Cold` is used as $\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}$ values of memory will be automatically allocated by `nag_opt_nlin_lsq` (e04unc).

If the option `start = Nag_Warm` has been chosen, `lambda` must point to a minimum of $\mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}$ 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` (e04unc) from the calling program, with `start = Nag_Cold` and the same values of \mathbf{n} , \mathbf{nclin} and \mathbf{ncnlin} . If a previous call has not been made, sufficient memory must be allocated by the user.

When a ‘warm start’ is chosen `lambda[j – 1]` must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by `state`, for $j = \mathbf{n} + \mathbf{nclin} + 1, \mathbf{n} + \mathbf{nclin} + 2, \dots, \mathbf{n} + \mathbf{nclin} + \mathbf{ncnlin}$. The remaining elements need not be set.

Note that if the j th 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 j th constraint is an inequality active at its lower bound (i.e., `state[j – 1] = 0`), `lambda[j – 1]` should be non-negative; if the j th 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.

On exit: 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`.

– double * r Default memory = $\mathbf{n} \times \mathbf{n}$

On entry: `h` need not be set if the default option of `start = Nag_Cold` is used as $\mathbf{n} \times \mathbf{n}$ values of memory will be automatically allocated by `nag_opt_nlin_lsq` (e04unc).

If the option `start = Nag_Warm` has been chosen, `h` must point to a minimum of $\mathbf{n} \times \mathbf{n}$ elements of memory. This memory will already be available if the calling program has used the `options` structure in a previous call to `nag_opt_nlin_lsq` (e04unc) with `start = Nag_Cold` and the same value of \mathbf{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 = NagTrue**, then **h** will already have been set correctly.

On exit: if **hessian = NagFalse**, **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)).

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

– Nag_Boolean Default = **NagFalse**

On entry: controls the contents of the optional parameter **h** on return from `nag_opt_nlin_lsq` (e04unc). `nag_opt_nlin_lsq` (e04unc) works exclusively with the *transformed* and *re-ordered* Hessian \tilde{H}_Q , and hence extra computation is required to form the Hessian itself. If **hessian = NagFalse**, **h** contains the Cholesky factor of the transformed and re-ordered Hessian. If **hessian = NagTrue**, the Cholesky factor of the approximate Hessian itself is formed and stored in **h**. This information is required by `nag_opt_nlin_lsq` (e04unc) if the next call to `nag_opt_nlin_lsq` (e04unc) will be made with optional parameter **start = NagWarm**.

– Nag_Boolean Default = **NagFalse**

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

– Integer i Default = 2

On entry: this parameter allows the user to reset the approximate Hessian matrix to $J^T J$ 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 , $J^T J$ 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` (e04unc).

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

Constraint: **h_reset_freq** > 0.

– Integer i

On exit: the number of major iterations which have been performed in `nag_opt_nlin_lsq` (e04unc).

– Integer i

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

11.3 Description of Printed Output

The level of printed output can be controlled by the user with the structure members **list**, **print_deriv**, **print_level** and **minor_print_level** (see Section `Nag_XCheckObjCon`). If **list = NagTrue** then the parameter values to `nag_opt_nlin_lsq` (e04unc) are listed, followed by the result of any derivative check if **print_deriv = NagD_Sum** or **NagD_Full**. The printout of results is governed by the values of **print_level** and **minor_print_level**. The default of **print_level = NagSoln_Iter** and **minor_print_level = NagNoPrint** 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` (e04unc).

If a simple derivative check, **verify_grad = NagSimpleCheck**, 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 Nag_XCheckObjCon) 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** (e04unc) will exit with **fail** 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 10). 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 Nag_XCheckObjCon) 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 10.3). As the solution is approached, **Merit function** will converge to the value of the objective function at the solution.

If the QP subproblem does not have a feasible point (signified by I at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty 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** (e04unc) 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 $\|Z^T g_{FR}\|$, the Euclidean norm of the projected gradient (see Section 10.1). **Norm Gz** will be approximately zero in the neighbourhood of a solution.

Cond Hz is a lower bound on the condition number of the projected Hessian approximation H_Z ($H_Z = Z^T H_{FR} Z = R_Z^T R_Z$); see (6) and (11), respectively). 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 10.4).
- I is printed if the QP subproblem has no feasible point.
- C is printed if central differences were used to compute the unspecified objective and constraint gradients. If the value of `Step` is zero, the switch to central differences was made because no lower point could be found in the line search. (In this case, the QP subproblem is re-solved with the central difference gradient and Jacobian.) If the value of `Step` is non-zero, central differences were computed because `Norm Gz` and `Violtn` imply that x is close to a Kuhn–Tucker point (see Section 10.1).
- 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 Nag_XCheckObjCon). 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 10.1). 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 (16)–(18) defined in the description of the optional parameter `optim_tol` in Section Nag_XCheckObjCon. Each letter is T if the test is satisfied, and F otherwise. The three tests indicate whether:
- the sequence of iterates has converged;
 - the projected gradient (`Norm Gz`) is sufficiently small; and
 - 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 (e04unc)` 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 \mathbf{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_Lx).

If **print_level** = **Nag_Soln_Iter_Full** then the diagonal elements of the matrix T associated with the TQ factorization (see (5)) of the QP working set and the diagonal elements of R , the triangular factor of the transformed and re-ordered Hessian (see (6)) 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` (e04unc) 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 $j = 1, 2, \dots, n$ 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 `Nag_XCheckObjCon`), **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 $\mathbf{bl}[j-1] \leq \mathbf{inf_bound}$, where **inf_bound** is the optional parameter.)

Upper bound is the upper bound specified for the variable j . (None indicates that $\mathbf{bu}[j-1] \geq \mathbf{inf_bound}$, 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 $\mathbf{bl}[j-1] \leq -\mathbf{inf_bound}$ and $\mathbf{bu}[j-1] \geq \mathbf{inf_bound}$, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if **State** is LL, and non-positive if **State** is UL.

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

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

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

N Con gives the name (N) and index $(j - n_L)$, for $j = n_L + 1, n_L + 2, \dots, n_L + n_N$ 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 **print_level** in the documentation for `nag_opt_lin_lsq` (e04ncc).

If **print_level** = **Nag_NoPrint** then printout will be suppressed; the user can print the final solution when `nag_opt_nlin_lsq` (e04unc) returns to the calling program.

11.3.1 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 **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 **print_fun** this will be called in preference to the internal print function of `nag_opt_nlin_lsq` (e04unc). Calls to the user-defined function are again controlled by means of the **print_level**, **minor_print_level** and **print_deriv** members. Information is provided through **st** and **comm**, the two structure arguments to **print_fun**.

If **comm** → **it_maj_prt** = **NagTrue** then results from the last major iteration of `nag_opt_nlin_lsq` (e04unc) are provided through **st**. Note that **print_fun** will be called with **comm** → **it_maj_prt** = **NagTrue** 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_nlinviol – 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

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

nz – Integer

The number of columns of Z (see Section 10.1).

bnd – Integer

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

lin – Integer

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

nln – Integer

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

penalty – double

The Euclidean norm of the vector of penalty 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 – Nag_Boolean

Nag_True if the sequence of iterates has converged, i.e., convergence condition (16) (see the description of **optim_tol** in Section Nag_XCheckObjCon) is satisfied.

norm_gz_small – Nag_Boolean

Nag_True if the projected gradient is sufficiently small, i.e., convergence condition (17) (see the description of **optim_tol** in Section Nag_XCheckObjCon) is satisfied.

violtn_small – Nag_Boolean

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

update_modified – Nag_Boolean

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

qp_not_feasible – Nag_Boolean

Nag_True if the QP subproblem has no feasible point.

c_diff – Nag_Boolean

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

step_limit_exceeded – Nag_Boolean

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

refactor – Nag_Boolean

Nag_True if the approximate Hessian has been refactorized.

x – double *

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

state – Integer *

Contains the status of the $\mathbf{st} \rightarrow \mathbf{n}$ variables, $\mathbf{st} \rightarrow \mathbf{nclin}$ linear, and $\mathbf{st} \rightarrow \mathbf{ncnlin}$ nonlinear constraints (if any). See Section Nag_XCheckObjCon for a description of the possible status values.

ax – double *

If $\mathbf{st} \rightarrow \mathbf{nclin} > 0$, $\mathbf{ax}[j-1]$ contains the current value of the j th linear constraint, for $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{nclin}$.

cx – double *

If $\mathbf{st} \rightarrow \mathbf{ncnlin} > 0$, $\mathbf{cx}[j-1]$ contains the current value of nonlinear constraint c_j , for $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{ncnlin}$.

diagt – double *

If $\mathbf{st} \rightarrow \mathbf{nactiv} > 0$, the $\mathbf{st} \rightarrow \mathbf{nactiv}$ elements of the diagonal of the matrix T .

diagr – double *

Contains the $\mathbf{st} \rightarrow \mathbf{n}$ elements of the diagonal of the upper triangular matrix R .

If $\mathbf{comm} \rightarrow \mathbf{sol_sqp_prt} = \mathbf{NagTrue}$ then the final result from nag_opt_nlin_lsq (e04unc) is provided through **st**. Note that **print_fun** will be called with $\mathbf{comm} \rightarrow \mathbf{sol_sqp_prt} = \mathbf{NagTrue}$ 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 $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{n}$.

state – Integer *

Contains the status of the $\mathbf{st} \rightarrow \mathbf{n}$ variables, $\mathbf{st} \rightarrow \mathbf{nclin}$ linear, and $\mathbf{st} \rightarrow \mathbf{ncnlin}$ nonlinear constraints (if any). See Section Nag_XCheckObjCon for a description of the possible status values.

ax – double *

If $\mathbf{st} \rightarrow \mathbf{nclin} > 0$, $\mathbf{ax}[j-1]$ contains the final value of the j th linear constraint, for $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{nclin}$.

cx – double *

If $\mathbf{st} \rightarrow \mathbf{ncnlin} > 0$, $\mathbf{cx}[j-1]$ contains the final value of nonlinear constraint c_j , for $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{ncnlin}$.

bl – double *

Contains the $\mathbf{st} \rightarrow \mathbf{n} + \mathbf{st} \rightarrow \mathbf{nclin} + \mathbf{st} \rightarrow \mathbf{ncnlin}$ lower bounds on the variables.

bu – double *

Contains the $\mathbf{st} \rightarrow \mathbf{n} + \mathbf{st} \rightarrow \mathbf{nclin} + \mathbf{st} \rightarrow \mathbf{ncnlin}$ upper bounds on the variables.

lambda – double *

Contains the $\mathbf{st} \rightarrow \mathbf{n} + \mathbf{st} \rightarrow \mathbf{nclin} + \mathbf{st} \rightarrow \mathbf{ncnlin}$ final values of the Lagrange multipliers.

If **comm** \rightarrow **g_prt** = **NagTrue** then the results from derivative checking are provided through **st**. Note that **print_fun** will be called with **comm** \rightarrow **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 $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{n}$.

fjac – double *

Contains elements of the Jacobian of F at the initial point x_0 ($\frac{\partial f_i}{\partial x_j}$ is held at location **fjac**[($i-1$) \times **st** \rightarrow **tdfjac** + $j-1$], for $i = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{m}$; $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{n}$).

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 ($\frac{\partial c_i}{\partial x_j}$ is held at location **conjac**[($i - 1$) \times **st** \rightarrow **n** + $j - 1$], for $i = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{ncnlin}$; $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{n}$).

In this case the details of any derivative check performed by nag_opt_nlin_lsq (e04unc) 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 (e04unc). This will be the same value as in **verify_grad**.

g_error – Integer

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

obj_start – Integer

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

f_sim – Nag_SimSt

The result of a simple derivative check of the objective gradient, **gprint** \rightarrow **g_chk.type** = **NagSimpleCheck**, 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 – Nag_Boolean

If **Nag_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 = NagSimpleCheck**, will be held in this substructure in members:

n_elements – Integer

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

correct – Nag_Boolean

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

max_error – double

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

max_constraint – Integer

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

f_comp – Nag_CompSt *

The results of a requested component derivative check of the Jacobian of the objective function subfunctions, $\mathbf{st} \rightarrow \mathbf{gprint.g_chk.type} = \mathbf{NagCheckObj}$ or $\mathbf{NagCheckObjCon}$, will be held in the array of $\mathbf{st} \rightarrow \mathbf{m} \times \mathbf{st} \rightarrow \mathbf{n}$ substructures of type **Nag_CompSt** pointed to by **f_comp**. The element $\mathbf{st} \rightarrow \mathbf{gprint.f_comp}[(i-1) \times \mathbf{st} \rightarrow \mathbf{n} + j - 1]$ will hold the details of the component derivative check for Jacobian element i, j , for $i = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{ncnlin}$; $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{n}$. 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.* (1983a).)

correct – Nag_Boolean

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

hopt – double

The optimal finite difference interval. This is $\mathbf{dx}[i]$ in the default derivative checking printout (see Section Itns).

gdif – 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, $\mathbf{st} \rightarrow \mathbf{gprint.g_chk.type} = \mathbf{NagCheckCon}$ or $\mathbf{NagCheckObjCon}$, will be held in the array of $\mathbf{st} \rightarrow \mathbf{ncnlin} \times \mathbf{st} \rightarrow \mathbf{n}$ substructures of type **Nag_CompSt** pointed to by **c_comp**. The element $\mathbf{st} \rightarrow \mathbf{gprint.f_comp}[(i-1) \times \mathbf{st} \rightarrow \mathbf{n} + j - 1]$ will hold the details of the component derivative check for Jacobian element i, j , for $i = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{ncnlin}$; $j = 1, 2, \dots, \mathbf{st} \rightarrow \mathbf{n}$. 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.* (1983a).)

The members of **c_comp** are as for **f_comp**.

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

g_prt – Nag_Boolean

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

it_maj_prt – Nag_Boolean

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

sol_sqp_prt – Nag_Boolean

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

it_prt – Nag_Boolean

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

new_lm – Nag_Boolean

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

sol_prt – Nag_Boolean

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

user – double *
iuser – Integer *
p – Pointer

Pointers for communication of user information. If used they must be allocated memory by the user either before entry to `nag_opt_nlin_lsq` (e04unc) or during a call to **objfun**, **confun** or **print_fun**. The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.

12 Example 2 (EX2)

Example 2 (EX2) solves the same problem as Example 1 (EX1), described in Section 9, but shows the use of the optional parameter, **print_fun**, which allows the user to customize the output from `nag_opt_nlin_lsq` (e04unc). 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** = **NagTrue**, major iteration printout if **comm** → **it_maj_prt** = **NagTrue** and final solution printout if **comm** → **sol_sqp_prt** = **NagTrue**. 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` (e04unc), 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.

See Section 9 for the example program.
