NAG Library Function Document

nag_opt_lin_lsq (e04ncc)

1 Purpose

nag_opt_lin_lsq (e04ncc) solves linearly constrained linear least squares problems and convex quadratic programming problems. It is not intended for large sparse problems.

2 Specification

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

```
void nag_opt_lin_lsq (Integer m, Integer n, Integer nclin, const double a[],
     Integer tda, const double bl[], const double bu[], const double cvec[],
     double b[], double h[], Integer tdh, Integer kx[], double x[],
     double *objf, Nag_E04_Opt *options, Nag_Comm *comm, NagError *fail)
```

3 Description

nag_opt_lin_lsq (e04ncc) is designed to solve a class of quadratic programming problems stated in the following general form:

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

where A is an n_L by n matrix and the objective function $F(x)$ may be specified in a variety of ways depending upon the particular problem to be solved. The available forms for $F(x)$ are listed in Table 1 below, in which the prefixes FP, LP, QP and LS stand for 'feasible point', 'linear programming', 'quadratic programming' and 'least squares' respectively, c is an n element vector, b is an m element vector, and $\|x\|$ denotes the Euclidean length of x.

Problem Type	$f(x)$	**Matrix** H
FP	Not applicable	Not applicable
LP	$c^{\mathrm{T}}x$	Not applicable
QP1	$\frac{1}{2}x^{\mathrm{T}}Hx$	n by n symmetric positive semidefinite
QP2	$c^{\mathrm{T}}x+\frac{1}{2}x^{\mathrm{T}}Hx$	n by n symmetric positive semidefinite
QP3	$\frac{1}{2}x^{\mathrm{T}}H^{\mathrm{T}}Hx$	m by n upper trapezoidal
QP4	$c^{\mathrm{T}}x+\frac{1}{2}x^{\mathrm{T}}H^{\mathrm{T}}Hx$	m by n upper trapezoidal
LS1	$\frac{1}{2}\|b-Hx\|^2$	m by n
LS2	$c^{\mathrm{T}}x+\frac{1}{2}\|b-Hx\|^2$	m by n
LS3	$\frac{1}{2}\|b-Hx\|^2$	m by n upper trapezoidal
LS4	$c^{\mathrm{T}}x+\frac{1}{2}\|b-Hx\|^2$	m by n upper trapezoidal

Table 1

For problems of type LS, H is referred to as the *least squares* matrix, or the *matrix of observations*, and b as the *vector of observations*. The default problem type is LS1, and other objective functions are selected by using the optional argument **options.prob** (see Section 12.2).

When H is upper trapezoidal it will usually be the case that $m=n$, so that H is upper triangular, but full generality has been allowed for in the specification of the problem. The upper trapezoidal form is intended for cases where a previous factorization, such as a QR factorization, has been performed.

The constraints involving A are called the *general* constraints. Note that upper and lower bounds are specified for all the variables and for all the general constraints. An equality constraint can be specified by setting $l_i = u_i$. If certain bounds are not present, the associated elements of l or u can be set to special values that will be treated as $-\infty$ or $+\infty$. (See the description of the optional argument **options.inf_bound** in Section 12.2.

The function $F(x)$ is a quadratic function, whose defining feature is that its second-derivative matrix $\nabla^2 F(x)$ (the *Hessian matrix*) is constant. For the LP case, $\nabla^2 F(x) = 0$; for QP1 and QP2, $\nabla^2 F(x) = H$; and for QP3, QP4 and LS problems, $\nabla^2 F(x) = H^{\mathrm{T}} H$ and the Hessian matrix is positive semidefinite (positive definite if H is full rank), so that $F(x)$ is convex. If H is defined as the zero matrix, nag_opt_lin_lsq (e04ncc) will solve the resulting linear programming problem; however, this can be accomplished more efficiently by using nag_opt_lp (e04mfc).

Problems of type QP3 and QP4 for which H is not in upper trapezoidal form should be solved as problems of type LS1 and LS2 respectively, with $b = 0$.

You must supply an initial estimate of the solution.

If H is of full rank then nag_opt_lin_lsq (e04ncc) will obtain the unique (global) minimum. If H is not of full rank then the solution may still be a global minimum if all active constraints have nonzero Lagrange multipliers. Otherwise the solution obtained will be either a weak minimum (i.e., with a unique optimal objective value, but an infinite set of optimal x), or else the objective function is unbounded below in the feasible region. The last case can only occur when $F(x)$ contains an explicit linear term (as in problems LP, QP2, QP4, LS2 and LS4).

The method used by nag_opt_lin_lsq (e04ncc) is described in detail in Section 11.

4 References

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

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

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

Stoer J (1971) On the numerical solution of constrained least squares problems *SIAM J. Numer. Anal.* **8** 382–411

5 Arguments

1: **m** – Integer *Input*

On entry: m, the number of rows in the matrix H. If the problem is of type FP or LP, **m** is not referenced and is assumed to be zero. The default type is LS1; other problem types can be specified using the optional argument **options.prob**, see Section 12.2.

If the problem is of type QP, **m** will usually be n, the number of variables. However, a value of **m** less than n is appropriate for problem type QP3 or QP4 if H is an upper trapezoidal matrix with m rows. Similarly, **m** may be used to define the dimension of a leading block of nonzeros in the Hessian matrices of QP1 or QP2. In QP cases, m should not be greater than m; if it is, the last $(m - n)$ rows of H are ignored.

If the problem is a least squares problem (in particular, the default type LS1), **m** is also the dimension of the array **b**. Note that all possibilities ($m < n$, $m = n$ and $m > n$) are allowed in this case.

Constraint: **m** > 0 if problem is not FP or LP.

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** ≥ 0.

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

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

On entry: the ith row of **a** must contain the coefficients of the ith general linear constraint (the ith row of A), for $i = 1, 2, \ldots, n_L$. If **nclin** $= 0$ then the array **a** is not referenced.

5: **tda** – Integer *Input*

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

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

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

Constraints:

bl$[j] \leq$ **bu**$[j]$, for $j = 0, 1, \ldots,$ **n** + **nclin** $- 1$;
if **bl**$[j] =$ **bu**$[j] = \beta$, $|\beta| <$ **options.inf_bound**.

8: **cvec**[**n**] – const double *Input*

On entry: the coefficients of the explicit linear term of the objective function when the problem is of type LP, QP2, QP4, LS2 or LS4.

If the problem is of type FP, QP1, QP3, LS1 (the default) or LS3, **cvec** is not referenced and may be **NULL**.

9: **b**[**m**] – double *Input/Output*

On entry: the m elements of the vector of observations.

On exit: the transformed residual vector of equation (10).

b is referenced only in the case of least squares problem types (in particular, the default type LS1. For other problem types, **b** is not referenced and may be **NULL**.

10: **h**[**m** $\times$ **tdh**] – double *Input/Output*

Note: the (i,j)th element of the matrix H is stored in **h**$[(i-1) \times$ **tdh** $+ j - 1]$.

On entry: the array **h** must contain the matrix H as specified in Table 1 (see Section 3).

For problems QP1 and QP2, the first m rows and columns of **h** must contain the leading m by m rows and columns of the symmetric Hessian matrix. Only the diagonal and upper triangular elements of the leading m rows and columns of **h** are referenced. The remaining elements are assumed to be zero and need not be assigned.

For problems QP3, QP4, LS3 and LS4, the first m rows of **h** must contain an m by n upper trapezoidal factor of either the Hessian or the least squares matrix, ordered according to the array **kx** (see below). The factor need not be of full rank, i.e., some of the diagonals may be zero.

However, as a general rule, the larger the dimension of the leading nonsingular sub-matrix of H, the fewer iterations will be required. Elements outside the upper trapezoidal part of the first m rows of H are assumed to be zero and need not be assigned.

If a constrained least squares problem contains a very large number of observations, storage limitations may prevent storage of the entire least squares matrix. In such cases, you should transform the original H into a triangular matrix before the call to nag_opt_lin_lsq (e04ncc) and solve as type LS3 or LS4.

On exit: by default, **h** contains the upper triangular Cholesky factor R of equation (8), with columns ordered as indicated by **kx** (see below). If the optional argument **options.hessian** = Nag_TRUE (see Section 12.2), and the problem is one of the LS or QP types, **h** contains the upper triangular Cholesky factor of the Hessian matrix $\nabla^2 F$, with columns ordered as indicated by **kx** (see below). In either case, this matrix may be used to obtain the variance-covariance matrix or to recover the upper triangular factor of the original least squares matrix.

If the problem is of type FP or LP, **h** is not referenced and may be **NULL**.

11: **tdh** – Integer *Input*

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

Constraint: **tdh** $\geq$ **n**.

12: **kx**[**n**] – Integer *Input/Output*

On entry: for problems of type QP3, QP4, LS3 or LS4 the array **kx** must specify the order of the columns of the matrix H with respect to the ordering of **x**. Thus if column j of H is the column associated with the variable x_i then **kx**$[j-1] = i$.

If the problem is of any other type then the array **kx** need not be initialized.

Constraints:

$1 \leq$ **kx**$[i] \leq$ **n**, for $i = 0, 1, \dots,$ **n** $- 1$;
if $i \neq j$, **kx**$[i] \neq$ **kx**$[j]$.

On exit: defines the order of the columns of H with respect to the ordering of **x**, as described above.

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

On entry: an initial estimate of the solution.

On exit: the point at which nag_opt_lin_lsq (e04ncc) terminated. If **fail.code** = NE_NOERROR, NW_SOLN_NOT_UNIQUE or NW_NOT_FEASIBLE, **x** contains an estimate of the solution.

14: **objf** – double * *Output*

On exit: the value of the objective function at x if x is feasible, or the sum of infeasibilities at x otherwise. If the problem is of type FP and x is feasible, **objf** is set to zero.

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

On entry/exit: a pointer to a structure of type Nag_E04_Opt whose members are optional arguments for nag_opt_lin_lsq (e04ncc). These structure members offer the means of adjusting some of the argument values of the algorithm and on output will supply further details of the results. A description of the members of **options** is given below in Section 12. Some of the results returned in **options** can be used by nag_opt_lin_lsq (e04ncc) to perform a 'warm start' (see the member **options.start** in Section 12.2).

If any of these optional arguments are required then the structure **options** should be declared and initialized by a call to nag_opt_init (e04xxc) and supplied as an argument to nag_opt_lin_lsq (e04ncc). However, if the optional arguments are not required the NAG defined null pointer, `E04_DEFAULT`, can be used in the function call.

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

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

On entry/exit: structure containing pointers for communication with an optional user-defined printing function; see Section 12.3.1 for details. If you do not need to make use of this communication feature the null pointer `NAGCOMM_NULL` may be used in the call to nag_opt_lin_lsq (e04ncc); **comm** will then be declared internally for use in calls to user-supplied functions.

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

The NAG error argument (see Section 3.6 in 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 with the structure member **options.print_level** (see Section 12.2). The default, **options.print_level** $=$ Nag_Soln_Iter provides a single line of output at each iteration and the final result. This section describes the default printout produced by nag_opt_lin_lsq (e04ncc).

The convention for numbering the constraints in the iteration results is that indices 1 to n refer to the bounds on the variables, and indices $n+1$ to $n+n_L$ refer to the general constraints.

The following line of output is produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Itn`	is the iteration count.
`Step`	is the step taken along the computed search direction. If a constraint is added during the current iteration, `Step` will be the step to the nearest constraint. During the optimality phase, the step can be greater than 1.0 only if the factor R_z is singular (see Section 11.3).
`Ninf`	is the number of violated constraints (infeasibilities). This will be zero during the optimality phase.
`Sinf/Objective`	is the value of the current objective function. If x is not feasible, `Sinf` gives a weighted sum of the magnitudes of constraint violations. If x is feasible, `Objective` is the value of the objective function. The output line for the final iteration of the feasibility phase (i.e., the first iteration for which `Ninf` is zero) will give the value of the true objective at the first feasible point. During the optimality phase, the value of the objective function will be non-increasing. During the feasibility phase, the number of constraint infeasibilities will not increase until either a feasible point is found, or the optimality of the multipliers implies that no feasible point exists. Once optimal multipliers are obtained, the number of infeasibilities can increase, but the sum of infeasibilities will either remain constant or be reduced until the minimum sum of infeasibilities is found.
`Norm Gz`	$\left\lVert Z_1^{\mathrm{T}} g_{\mathrm{FR}} \right\rVert$, the Euclidean norm of the reduced gradient with respect to Z_1 (see Section 11.3). During the optimality phase, this norm will be approximately zero after a unit step.

The printout of the final result consists of:

`Varbl`	gives the name (`V`) and index j, for $j=1,2,\ldots,n$ of the variable.
`State`	gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than the optional argument **options.ftol** (default value $\sqrt{\epsilon}$, where ϵ is the ***machine precision***; see Section 12.2), `State` will be `++` or `--` respectively. A key is sometimes printed before `State` to give some additional information about the state of a variable.

	`A`	*Alternative optimum possible*. The variable is active at one of its bounds, but its Lagrange Multiplier is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change to the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them could encounter a bound immediately. In either case, the values of the Lagrange multipliers might also change.
	`D`	*Degenerate*. The variable is free, but it is equal to (or very close to) one of its bounds.
	`I`	*Infeasible*. The variable is currently violating one of its bounds by more than **options.ftol**.

`Value`	is the value of the variable at the final iteration.
`Lower bound`	is the lower bound specified for variable j. (`None` indicates that **bl**$[j-1] \leq -$**options.inf_bound**, where **options.inf_bound** is the optional argument.)
`Upper bound`	is the upper bound specified for variable j. (`None` indicates that **bu**$[j-1] \geq$ **options.inf_bound**, where **options.inf_bound** is the optional argument.)
`Lagr mult`	is the value of the Lagrange multiplier for the associated bound. This will be zero if `State` is `FR` unless **bl**$[j-1] \leq -$**options.inf_bound** and **bu**$[j-1] \geq$ **options.inf_bound**, in which case the entry will be blank. If x is optimal, the multiplier should be non-negative if `State` is `LL`, and non-positive if `State` is `UL`.
`Residual`	is the difference between the variable `Value` and the nearer of its (finite) bounds **bl**$[j-1]$ and **bu**$[j-1]$. A blank entry indicates that the associated variable is not bounded (i.e., **bl**$[j-1] \leq -$**options.inf_bound** and **bu**$[j-1] \geq$ **options.inf_bound**).

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

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

Note that movement off a constraint (as opposed to a variable moving away from its bound) can be interpreted as allowing the entry in the `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 arguments must satisfy **tda** $\geq$ **n**.

On entry, **tdh** $= \langle value \rangle$ while **n** $= \langle value \rangle$. These arguments must satisfy **tdh** $\geq$ **n**.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_ARRAY_CONS

The contents of array **kx** are not valid. Constraint: must contain a permutation of integers $1, 2, \ldots,$ **n**.

NE_B_NULL

options.prob $= \langle value \rangle$ but argument **b** $=$ **NULL**.

NE_BAD_PARAM

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

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

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

NE_BOUND

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

NE_BOUND_LCON

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

NE_CVEC_NULL

options.**prob** = ⟨*value*⟩ but argument **cvec** = **NULL**.

NE_CYCLING

The algorithm could be cycling, since a total of 50 changes were made to the working set without altering x. Check the detailed iteration printout for a repeated pattern of constraint deletions and additions.

If a sequence of constraint changes is being repeated, the iterates are probably cycling. (nag_opt_lin_lsq (e04ncc) does not contain a method that is guaranteed to avoid cycling; such a method would be combinatorial in nature.) Cycling may occur in two circumstances: at a constrained stationary point where there are some small or zero Lagrange multipliers; or at a point (usually a vertex) where the constraints that are satisfied exactly are nearly linearly dependent. In the latter case, you have the option of identifying the offending dependent constraints and removing them from the problem, or restarting the run with a larger value of the optional argument **options**.**ftol** (default value $= \sqrt{\epsilon}$, where ϵ is the ***machine precision***; see Section 12.2). If this error exit occurs but no suspicious pattern of constraint changes can be observed, it may be worthwhile to restart with the final x (with optional argument **options**.**start** = Nag_Cold or Nag_Warm).

NE_H_NULL_QP

options.**prob** = ⟨*value*⟩ but argument **h** = **NULL**. This problem type requires an array to be supplied in argument **h**.

NE_INT_ARG_LT

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

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

On entry, **nclin** = ⟨*value*⟩.
Constraint: **nclin** ≥ 0.

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 contact NAG for assistance.

NE_INVALID_INT_RANGE_1

Value ⟨*value*⟩ given to **options**.**fmax_iter** is not valid. Correct range is **options**.**fmax_iter** ≥ 0.

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

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

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

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

NE_INVALID_REAL_RANGE_F

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

NE_INVALID_REAL_RANGE_FF

Value $\langle value\rangle$ given to **options**.**crash_tol** is not valid. Correct range is $0.0 \leq$ **options**.**crash_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

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

NE_UNBOUNDED

Solution appears to be unbounded.

This error indicator implies that a step as large as optional argument **options**.**inf_step** (default value 10^{20}; see Section 12.2) would have to be taken in order to continue the algorithm. This situation can occur only when H is singular, there is an explicit linear term, and at least one variable has no upper or lower bound.

NE_WARM_START

options.**start** = Nag_Warm but pointer **options**.**state** = **NULL**.

NE_WRITE_ERROR

Error occurred when writing to file $\langle string\rangle$.

NW_NOT_FEASIBLE

No feasible point was found for the linear constraints.

It was not possible to satisfy all the constraints to within the feasibility tolerance. In this case, the constraint violations at the final x will reveal a value of the tolerance for which a feasible point will exist – for example, if the feasibility tolerance for each violated constraint exceeds its `Residual` (see Section 5.1) at the final point. The modified problem (with an altered value of the optional feasibility tolerance, **options**.**ftol**) may then be solved using optional argument **options**.**start** = Nag_Warm (see Section 12.2). You should check that there are no constraint redundancies. If the data for the constraints are accurate only to the absolute precision σ, you should ensure that the value of **options**.**ftol** is *greater* than σ. For example, if all elements of A are of order unity and are accurate only to three decimal places, **options**.**ftol** should be at least 10^{-3}.

NW_OVERFLOW_WARN

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

If overflow occurs preceded by this warning then serious ill conditioning has probably occurred in the working set when adding a constraint. It may be possible to avoid the difficulty by increasing the magnitude of the optional argument **options**.**ftol** and re-running the program. If the message recurs even after this change, the offending linearly dependent constraint j must be removed from the problem.

NW_SOLN_NOT_UNIQUE

Optimal solution is not unique.

The point in **x** is a weak local minimum, i.e., the projected gradient is negligible, the Lagrange multipliers are optimal, but either R_z (see Section 11.3) is singular or there is a small multiplier. This means that x is not unique.

NW_TOO_MANY_ITER

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

The limiting number of iterations (determined by the optional arguments **options**.**max_iter** and **options**.**fmax_iter**, see Section 12.2) was reached before normal termination occurred. If the method appears to be making progress (e.g., the objective function is being satisfactorily reduced), either increase the iteration limits or, alternatively, rerun nag_opt_lin_lsq (e04ncc) using the optional argument **options**.**start** = Nag_Warm to specify the initial working set. If the iteration limit is already large, but some of the constraints could be nearly linearly dependent, check the extended iteration printout (see Section 12.3) for a repeated pattern of constraints entering and leaving the working set. (Near-dependencies are often indicated by wide variations in size in the diagonal elements of the matrix T (see Section 11.2), which will be printed if optional argument **options**.**print_level** = Nag_Soln_Iter_Full (default value **options**.**print_level** = Nag_Soln_Iter; see Section 12.2.) In this case, the algorithm could be cycling (see the comments below for **fail**.**code** = NE_CYCLING).

7 Accuracy

nag_opt_lin_lsq (e04ncc) implements a numerically stable active set strategy and returns solutions that are as accurate as the condition of the problem warrants on the machine.

8 Parallelism and Performance

Not applicable.

9 Further Comments

9.1 Termination Criteria

nag_opt_lin_lsq (e04ncc) exits with **fail**.**code** = NE_NOERROR if x is a strong local minimizer, i.e., the reduced gradient is negligible, the Lagrange multipliers are optimal (see Section 5.1) and R_z (see Section 11.3) is nonsingular.

9.2 Scaling

Sensible scaling of the problem is likely to reduce the number of iterations required and make the problem less sensitive to perturbations in the data, thus improving the condition of the problem. In the absence of better information it is usually sensible to make the Euclidean lengths of each constraint of comparable magnitude. See the e04 Chapter Introduction and Gill *et al.* (1981) for further information and advice.

10 Example

To minimize the quadratic function $c^{\mathrm{T}}x + \frac{1}{2}x^{\mathrm{T}}Hx$, where

$$c = (-4.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -0.3)^{\mathrm{T}},$$

$$H = \begin{pmatrix} 2 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

subject to the bounds

$$\begin{array}{c} -2 \leq x_1 \leq 2 \\ -2 \leq x_2 \leq 2 \\ -2 \leq x_3 \leq 2 \\ -2 \leq x_4 \leq 2 \\ -2 \leq x_5 \leq 2 \\ -2 \leq x_6 \leq 2 \\ -2 \leq x_7 \leq 2 \\ -2 \leq x_8 \leq 2 \\ -2 \leq x_9 \leq 2 \end{array}$$

and to the general constraints

$$\begin{array}{r} -2.0 \leq x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + 4x_9 \leq 1.5 \\ -2.0 \leq x_1 + 2x_2 + 3x_3 + 4x_4 - 2x_5 + x_6 + x_7 + x_8 + x_9 \leq 1.5 \\ -2.0 \leq x_1 - x_2 + x_3 - x_4 + x_5 + x_6 + x_7 + x_8 + x_9 \leq 4.0 \end{array}$$

The initial point, which is feasible, is

$$x_0 = (0, 0, 0, 0, 0, 0, 0, 0, 0)^{\mathrm{T}},$$

and $F(x_0) = 0$.

The optimal solution (to five figures) is

$$x^* = (2.0, -0.23333, -0.26667, -0.3, -0.1, 2.0, 2.0, -1.7777, -0.45555)^{\mathrm{T}},$$

and $F(x^*) = -8.0678$. Three bound constraints and two general constraints are active at the solution. Note that, although the Hessian matrix is positive semidefinite, the point x^* is unique.

This example illustrates the use of the **options** structure. Since the problem is of type QP2 (as described in Section 3) and the default value of the optional argument **options.prob** = Nag_LS1, it is necessary to reset this argument to **options.prob** = Nag_QP2 in order to solve the problem. This is achieved by declaring the **options** structure and initializing it by calling nag_opt_init (e04xxc). Then **options.prob** is assigned directly, before calling nag_opt_lin_lsq (e04ncc). On return from nag_opt_lin_lsq (e04ncc), nag_opt_free (e04xzc) is used to free the memory assigned to the pointers in the options structure. You must **not** use the standard C function `free()` for this purpose.

10.1 Program Text

```
/* nag_opt_lin_lsq (e04ncc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 5, 1998.
 *
 * Mark 6 revised, 2000.
 * Mark 8 revised, 2004.
 *
 */

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

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

int main(void)
{
  Integer     exit_status = 0;
  Integer     i, j, *kx = 0, m, n, nbnd, nclin, tda, tdh;
  Nag_E04_Opt options;
  double      *a = 0, *bl = 0, *bu = 0, *cvec = 0, *h = 0, objf, *x = 0;
  Nag_Comm    comm;
  NagError    fail;

  INIT_FAIL(fail);

  printf("nag_opt_lin_lsq (e04ncc) Example Program Results\n");
  fflush(stdout);

#ifdef _WIN32
  scanf_s(" %*[^\n]"); /* Skip heading in data file */
#else
  scanf(" %*[^\n]"); /* Skip heading in data file */
#endif

  /* Read problem dimensions */
#ifdef _WIN32
  scanf_s(" %*[^\n]");
#else
  scanf(" %*[^\n]");
#endif
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n]", &m, &n, &nclin);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n]", &m, &n, &nclin);
#endif

  if (m > 0 && n > 0 && nclin >= 0)
    {
      nbnd = n + nclin;
      if (!(a = NAG_ALLOC(nclin*n, double)) ||
          !(bl = NAG_ALLOC(nbnd, double)) ||
          !(bu = NAG_ALLOC(nbnd, double)) ||
          !(cvec = NAG_ALLOC(n, double)) ||
          !(h = NAG_ALLOC(m*n, double)) ||
          !(x = NAG_ALLOC(n, double)) ||
          !(kx = NAG_ALLOC(n, Integer)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
      tda = n;
      tdh = n;
    }
  else
    {
      printf("Invalid m or n or nclin.\n");
      exit_status = 1;
      return exit_status;
    }

  /* We solve a QP2 type problem in this example */
  /* Read cvec, h, a, bl, bu and x from data file */

#ifdef _WIN32
  scanf_s(" %*[^\n]");
#else
  scanf(" %*[^\n]");
#endif
  for (i = 0; i < m; ++i)
#ifdef _WIN32
    scanf_s("%lf", &cvec[i]);
#else
    scanf("%lf", &cvec[i]);
#endif

#ifdef _WIN32
  scanf_s(" %*[^\n]");
#else
  scanf(" %*[^\n]");
#endif
  for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
#ifdef _WIN32
      scanf_s("%lf", &H(i, j));
#else
      scanf("%lf", &H(i, j));
#endif

  if (nclin > 0)
    {
#ifdef _WIN32
      scanf_s(" %*[^\n]");
#else
      scanf(" %*[^\n]");
#endif
      for (i = 0; i < nclin; ++i)
        for (j = 0; j < n; ++j)
#ifdef _WIN32
          scanf_s("%lf", &A(i, j));
#else
          scanf("%lf", &A(i, j));
#endif
    }

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

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

  /* Read the initial point x */
```

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

  /* Change the problem type */
  /* nag_opt_init (e04xxc).
   * Initialization function for option setting
   */
  nag_opt_init(&options);
  options.prob = Nag_QP2;

  /* nag_opt_lin_lsq (e04ncc), see above. */
  nag_opt_lin_lsq(m, n, nclin, a, tda, bl, bu, cvec, (double *) 0,
                  h, tdh, kx, x, &objf, &options, &comm, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_opt_lin_lsq (e04ncc).\n%s\n",
              fail.message);
      exit_status = 1;
    }

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

 END:
  NAG_FREE(a);
  NAG_FREE(bl);
  NAG_FREE(bu);
  NAG_FREE(cvec);
  NAG_FREE(h);
  NAG_FREE(x);
  NAG_FREE(kx);

  return exit_status;
}
```

10.2 Program Data

```
nag_opt_lin_lsq (e04ncc) Example Program Data
Values of m, n, nclin
  9   9   3

Objective function vector cvec
 -4.0  -1.0  -1.0  -1.0  -1.0  -1.0  -1.0  -0.1  -0.3

Objective function matrix H
  2.0   1.0   1.0   1.0   1.0   0.0   0.0   0.0   0.0
  1.0   2.0   1.0   1.0   1.0   0.0   0.0   0.0   0.0
  1.0   1.0   2.0   1.0   1.0   0.0   0.0   0.0   0.0
  1.0   1.0   1.0   2.0   1.0   0.0   0.0   0.0   0.0
  1.0   1.0   1.0   1.0   2.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

Linear constraint matrix A
  1.0   1.0   1.0   1.0   1.0   1.0   1.0   1.0   4.0
  1.0   2.0   3.0   4.0  -2.0   1.0   1.0   1.0   1.0
  1.0  -1.0   1.0  -1.0   1.0   1.0   1.0   1.0   1.0

Lower bounds
 -2.0  -2.0  -2.0  -2.0  -2.0  -2.0  -2.0  -2.0  -2.0  -2.0  -2.0  -2.0

Upper bounds
  2.0   2.0   2.0   2.0   2.0   2.0   2.0   2.0   2.0   1.5   1.5   4.0

Initial estimate of x
  0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

10.3 Program Results

```
nag_opt_lin_lsq (e04ncc) Example Program Results

Parameters to e04ncc
--------------------

Linear constraints............   3    Number of variables...........   9
Objective matrix rows.........   9

prob....................   Nag_QP2    start...................  Nag_Cold
ftol....................  1.05e-08    rank_tol................  1.05e-07
crash_tol...............  1.00e-02    hessian..................    Nag_FALSE
inf_bound...............  1.00e+20    inf_step................  1.00e+20
fmax_iter...............        60    max_iter................        60
machine precision.......  1.11e-16
print_level......... Nag_Soln_Iter
outfile.................    stdout

Memory allocation:
state...................       Nag
ax......................       Nag    lambda..................       Nag

Rank of the objective function data matrix = 5


  Itn     Step  Ninf  Sinf/Objective  Norm Gz
   0   0.0e+00    0    0.000000e+00  4.5e+00
   1   7.5e-01    0   -4.375000e+00  5.0e-01
   2   1.0e+00    0   -4.400000e+00  2.8e-17
   3   3.0e-01    0   -4.700000e+00  8.9e-01
   4   1.0e+00    0   -5.100000e+00  2.4e-17
   5   5.4e-01    0   -6.055714e+00  1.7e+00
   6   1.1e-02    0   -6.113326e+00  1.6e+00
   7   1.1e-01    0   -6.215049e+00  1.2e+00
   8   1.0e+00    0   -6.538008e+00  1.8e-17
   9   6.5e-01    0   -7.428704e+00  7.2e-02
  10   1.0e+00    0   -7.429717e+00  1.8e-17
  11   1.0e+00    0   -8.067718e+00  1.8e-17
  12   1.0e+00    0   -8.067778e+00  1.8e-17
Exit from QP problem after 12 iterations.


Varbl State     Value      Lower Bound   Upper Bound    Lagr Mult    Residual
V   1   UL   2.00000e+00  -2.00000e+00   2.00000e+00  -8.0000e-01   0.0000e+00
V   2   FR  -2.33333e-01  -2.00000e+00   2.00000e+00   0.0000e+00   1.7667e+00
V   3   FR  -2.66667e-01  -2.00000e+00   2.00000e+00   0.0000e+00   1.7333e+00
V   4   FR  -3.00000e-01  -2.00000e+00   2.00000e+00   0.0000e+00   1.7000e+00
V   5   FR  -1.00000e-01  -2.00000e+00   2.00000e+00   0.0000e+00   1.9000e+00
V   6   UL   2.00000e+00  -2.00000e+00   2.00000e+00  -9.0000e-01   0.0000e+00
V   7   UL   2.00000e+00  -2.00000e+00   2.00000e+00  -9.0000e-01   0.0000e+00
V   8   FR  -1.77778e+00  -2.00000e+00   2.00000e+00   0.0000e+00   2.2222e-01
V   9   FR  -4.55556e-01  -2.00000e+00   2.00000e+00   0.0000e+00   1.5444e+00
```

```
L Con State      Value       Lower Bound   Upper Bound    Lagr Mult    Residual
L   1   UL   1.50000e+00  -2.00000e+00   1.50000e+00  -6.6667e-02   1.1102e-15
L   2   UL   1.50000e+00  -2.00000e+00   1.50000e+00  -3.3333e-02  -4.4409e-16
L   3   FR   3.93333e+00  -2.00000e+00   4.00000e+00   0.0000e+00   6.6667e-02

Exit after 12 iterations.

Optimal QP solution found.

Final QP objective value =  -8.0677778e+00
```

11 Further Description

This section gives a detailed description of the algorithm used in nag_opt_lin_lsq (e04ncc). This, and possibly the next section, Section 12, may be omitted if the more sophisticated features of the algorithm and software are not currently of interest.

11.1 Overview

nag_opt_lin_lsq (e04ncc) is based on an inertia-controlling method that maintains a Cholesky factorization of the reduced Hessian (see below). The method is based on that of Gill and Murray (1978) and is described in detail by Gill *et al.* (1981). Here we briefly summarise the main features of the method.

nag_opt_lin_lsq (e04ncc) uses essentially the same algorithm as the subroutine LSSOL described in Gill *et al.* (1986). It is based on a two-phase (primal) quadratic programming method with features to exploit the convexity of the objective function due to Gill *et al.* (1984). (In the full-rank case, the method is related to that of Stoer, see Stoer (1971).) nag_opt_lin_lsq (e04ncc) has two phases: 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 two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function. The feasibility phase does *not* perform the standard simplex method (i.e., it does not necessarily find a vertex), except in the LP case when $n_L \leq n$. Once any iterate is feasible, all subsequent iterates remain feasible.

nag_opt_lin_lsq (e04ncc) has been designed to be efficient when used to solve a *sequence* of related problems — for example, within a sequential quadratic programming method for nonlinearly constrained optimization (e.g., nag_opt_nlp (e04ucc)). In particular, you may specify an initial working set (the indices of the constraints believed to be satisfied exactly at the solution); see the discussion of the optional argument **options.start** in Section 12.2.

In general, an iterative process is required to solve a quadratic program. (For simplicity, we shall always consider a typical iteration and avoid reference to the index of the iteration.) Each new iterate $\bar{x}$ is defined by

$$\bar{x} = x + \alpha p, \qquad (2)$$

where the *step length* α is a non-negative scalar, and p is called the *search direction.*

At each point x, a *working set* of constraints is defined to be a linearly independent subset of the constraints that are satisfied 'exactly' (to within the tolerance defined by the optional argument **options.ftol**; see Section 12.2). The working set is the current prediction of the constraints that hold with equality at a solution of (1). The search direction is constructed so that the constraints in the working set remain *unaltered* for any value of the step length. For a bound constraint in the working set, this property is achieved by setting the corresponding element of the search direction to zero. Thus, the associated variable is *fixed*, and specification of the working set induces a partition of x into *fixed* and *free* variables. During a given iteration, the fixed variables are effectively removed from the problem; since the relevant elements of the search direction are zero, the columns of A corresponding to fixed variables may be ignored.

Let n_{W} denote the number of general constraints in the working set and let n_{FX} denote the number of variables fixed at one of their bounds (n_{W} and n_{FX} are the quantities `Lin` and `Bnd` in the extended iteration printout from nag_opt_lin_lsq (e04ncc); see Section 12.3). Similarly, let n_{FR} $(n_{\mathrm{FR}} = n - n_{\mathrm{FX}})$ denote the number of free variables. At every iteration, *the variables are re-ordered so that the last* n_{FX} *variables are fixed,* with all other relevant vectors and matrices ordered accordingly. The order of the variables is indicated by the contents of the array **kx** on exit (see Section 5).

11.2 Definition of the Search Direction

Let A_{FR} denote the n_{W} by n_{FR} sub-matrix of general constraints in the working set corresponding to the free variables, and let p_{FR} denote the search direction with respect to the free variables only. The general constraints in the working set will be unaltered by any move along p if

$$A_{\mathrm{FR}} p_{\mathrm{FR}} = 0. \tag{3}$$

In order to compute p_{FR}, the *TQ factorization* of A_{FR} is used:

$$A_{\mathrm{FR}} Q_{\mathrm{FR}} = \begin{pmatrix} 0 & T \end{pmatrix} \tag{4}$$

where T is a nonsingular n_{W} by n_{W} reverse-triangular matrix (i.e., $t_{ij} = 0$ if $i + j < n_{\mathrm{W}}$), and the nonsingular n_{FR} by n_{FR} matrix Q_{FR} is the product of orthogonal transformations (see Gill *et al.* (1984)). If the columns of Q_{FR} are partitioned so that

$$Q_{\mathrm{FR}} = \begin{pmatrix} Z & Y \end{pmatrix}, \tag{5}$$

where Y is n_{FR} by n_{W}, then the n_Z $(n_Z = n_{\mathrm{FR}} - n_{\mathrm{W}})$ columns of Z form a basis for the null space of A_{FR}. Let n_R be an integer such that $0 \le n_R \le n_Z$, and let Z_1 denote a matrix whose n_R columns are a subset of the columns of Z. (The integer n_R is the quantity `Zr` in the extended iteration printout from nag_opt_lin_lsq (e04ncc); see Section 12.3. In many cases, Z_1 will include *all* the columns of Z.) The direction p_{FR} will satisfy (3) if

$$p_{\mathrm{FR}} = Z_1 p_Z \tag{6}$$

where p_Z is any n_R-vector.

11.3 The Main Iteration

Let Q denote the n by n matrix

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

where I_{FX} is the identity matrix of order n_{FX}. Let R denote an n by n upper triangular matrix (the *Cholesky factor*) such that

$$Q^{\mathrm{T}} \tilde{\nabla^2} F Q \equiv H_Q = R^{\mathrm{T}} R, \tag{8}$$

and let the matrix of the first n_Z rows and columns of R be denoted by R_Z. (The matrix $\tilde{\nabla^2} F$ in (8) is the Hessian with its rows and columns permuted so that the free variables come first.)

The definition of p_Z in (6) depends on whether or not the matrix R_Z is singular at x. In the nonsingular case, p_Z satisfies the equations

$$R_Z^{\mathrm{T}} R_Z p_Z = -g_Z \tag{9}$$

where g_Z denotes the vector $Z^{\mathrm{T}} g_{\mathrm{FR}}$ and g denotes the objective gradient. (The norm of g_{FR} is the printed quantity `Norm Gf`; see Section 12.3.) When p_Z is defined by (9), $x + p$ is the minimizer of the objective function subject to the constraints (bounds and general) in the working set treated as equalities. In general, a vector f_Z is available such that $R_Z^{\mathrm{T}} f_Z = -g_Z$, which allows p_Z to be computed from a single back-substitution $R_Z p_Z = f_Z$. For example, when solving problem LS1, f_Z comprises the first n_Z elements of the *transformed residual vector*

$$f = P(b - Hx) \tag{10}$$

which is recurred from one iteration to the next, where P is an orthogonal matrix.

In the singular case, p_Z is defined such that

$$R_Z p_Z = 0 \quad \text{and} \quad g_Z^{\mathrm{T}} p_Z < 0. \tag{11}$$

This vector has the property that the objective function is linear along p and may be reduced by any step of the form $x + \alpha p$, where $\alpha > 0$.

The vector $Z^{\mathrm{T}} g_{\mathrm{FR}}$ is known as the *projected gradient* at x. If the projected gradient is zero, x is a constrained stationary point in the subspace defined by Z. During the feasibility phase, the projected gradient will usually be zero only at a vertex (although it may be zero at non-vertices in the presence of constraint dependencies). During the optimality phase, a zero projected gradient implies that x minimizes the quadratic objective when the constraints in the working set are treated as equalities. At a constrained stationary point, Lagrange multipliers λ_A and λ_B for the general and bound constraints are defined from the equations

$$A_{\mathrm{FR}}^{\mathrm{T}} \lambda_A = g_{\mathrm{FR}} \quad \text{and} \quad \lambda_B = g_{\mathrm{FX}} - A_{\mathrm{FX}}^{\mathrm{T}} \lambda_A. \tag{12}$$

Given a positive constant δ of the order of the ***machine precision***, the Lagrange multiplier λ_j corresponding to an inequality constraint in the working set is said to be *optimal* if $\lambda_j \leq \delta$ when the associated constraint is at its *upper bound*, or if $\lambda_j \geq -\delta$ when the associated constraint is at its *lower bound*. If a multiplier is non-optimal, the objective function (either the true objective or the sum of infeasibilities) can be reduced by deleting the corresponding constraint (with index `Jdel`; see Section 12.3) from the working set.

If optimal multipliers occur during the feasibility phase and the sum of infeasibilities is nonzero, there is no feasible point, and nag_opt_lin_lsq (e04ncc) will continue until the minimum value of the sum of infeasibilities has been found. At this point, the Lagrange multiplier λ_j corresponding to an inequality constraint in the working set will be such that $-(1 + \delta) \leq \lambda_j \leq \delta$ when the associated constraint is at its *upper bound*, and $-\delta \leq \lambda_j \leq (1 + \delta)$ when the associated constraint is at its *lower bound*. Lagrange multipliers for equality constraints will satisfy $|\lambda_j| \leq 1 + \delta$.

The choice of step length is based on remaining feasible with respect to the satisfied constraints. If R_Z is nonsingular and $x + p$ is feasible, α will be taken as unity. In this case, the projected gradient at $\bar{x}$ will be zero, and Lagrange multipliers are computed. Otherwise, α is set to α_M, the step to the 'nearest' constraint (with index `Jadd`; see Section 12.3), which is added to the working set at the next iteration.

If H is not input as a triangular matrix, it is overwritten by a triangular matrix R satisfying (8) obtained using the Cholesky factorization in the QP case, or the QR factorization in the LS case. Column interchanges are used in both cases, and an estimate is made of the rank of the triangular factor. Thereafter, the dependent rows of R are eliminated from the problem.

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

$$f = Pb - \begin{pmatrix} R \\ 0 \end{pmatrix} Q^{\mathrm{T}} x, (b \equiv 0 \text{ for the QP case }),$$

and

$$Q^{\mathrm{T}} g = Q^{\mathrm{T}} c - R^{\mathrm{T}} f.$$

Note that the triangular factor R associated with the Hessian of the original problem is updated during both the optimality *and* the feasibility phases.

The treatment of the singular case depends critically on the following feature of the matrix updating schemes used in nag_opt_lin_lsq (e04ncc): if a given factor R_Z is nonsingular, it can become singular during subsequent iterations only when a constraint leaves the working set, in which case only its last diagonal element can become zero. This property implies that a vector satisfying (11) may be found using the single back-substitution $\bar{R}_Z p_Z = e_Z$, where $\bar{R}_Z$ is the matrix R_Z with a unit last diagonal, and e_Z is a vector of all zeros except in the last position. If the Hessian matrix $\nabla^2 F$ is singular, the matrix R (and hence R_Z) may be singular at the start of the optimality phase. However, R_Z will be nonsingular if enough constraints are included in the initial working set. (The matrix with no rows and columns is positive definite by definition, corresponding to the case when A_{FR} contains n_{FR} constraints.) The idea is to include as many general constraints as necessary to ensure a nonsingular R_Z.

At the beginning of each phase, an upper triangular matrix R_1 is determined that is the largest nonsingular leading sub-matrix of R_Z. The use of interchanges during the factorization of H tends to maximize the dimension of R_1. (The rank of R_1 is estimated using the optional argument **options.rank_tol**; see Section 12.2.) Let Z_1 denote the columns of Z corresponding to R_1, and let Z be partitioned as $Z = \begin{pmatrix} Z_1 & Z_2 \end{pmatrix}$. A working set for which Z_1 defines the null space can be obtained by including *the rows of* Z_2^{T} as 'artificial constraints'. Minimization of the objective function then proceeds within the subspace defined by Z_1.

The artificially augmented working set is given by

$$\bar{A}_{\mathrm{FR}} = \begin{pmatrix} A_{\mathrm{FR}} \\ Z_2^{\mathrm{T}} \end{pmatrix}, \tag{13}$$

so that p_{FR} will satisfy $A_{\mathrm{FR}} p_{\mathrm{FR}} = 0$ and $Z_2^{\mathrm{T}} p_{\mathrm{FR}} = 0$. By definition of the TQ factorization, $\bar{A}_{\mathrm{FR}}$ *automatically* satisfies the following:

$$\bar{A}_{\mathrm{FR}} Q_{\mathrm{FR}} = \begin{pmatrix} A_{\mathrm{FR}} \\ Z_2^{\mathrm{T}} \end{pmatrix} Q_{\mathrm{FR}} = \begin{pmatrix} A_{\mathrm{FR}} \\ Z_2^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} Z_1 & Z_2 & Y \end{pmatrix} = \begin{pmatrix} 0 & \bar{T} \end{pmatrix},$$

where

$$\bar{T} = \begin{pmatrix} 0 & T \\ I & 0 \end{pmatrix},$$

and hence the TQ factorization of (13) requires no additional work.

The matrix Z_2 need not be kept fixed, since its role is purely to define an appropriate null space; the TQ factorization can therefore be updated in the normal fashion as the iterations proceed. No work is required to 'delete' the artificial constraints associated with Z_2 when $Z_1^{\mathrm{T}} g_{\mathrm{FR}} = 0$, since this simply involves repartitioning Q_{FR}. When deciding which constraint to delete, the 'artificial' multiplier vector associated with the rows of Z_2^{T} is equal to $Z_2^{\mathrm{T}} g_{\mathrm{FR}}$, and the multipliers corresponding to the rows of the 'true' working set are the multipliers that would be obtained if the temporary constraints were not present.

The number of columns in Z_2 and Z_1, the Euclidean norm of $Z_1^{\mathrm{T}} g_{\mathrm{FR}}$, and the condition estimator of R_1 appear in the extended iteration printout as `Art`, `Zr`, `Norm Gz` and `Cond Rz` respectively (see Section 12.3).

Although the algorithm of nag_opt_lin_lsq (e04ncc) does not perform simplex steps in general, there is one exception: a linear program with fewer general constraints than variables (i.e., $n_L \leq n$). (Use of the simplex method in this situation leads to savings in storage.) At the starting point, the 'natural' working set (the set of constraints exactly or nearly satisfied at the starting point) is augmented with a suitable number of 'temporary' bounds, each of which has the effect of temporarily fixing a variable at its current value. In subsequent iterations, a temporary bound is treated as a standard constraint until it is deleted from the working set, in which case it is never added again.

One of the most important features of nag_opt_lin_lsq (e04ncc) is its control of the conditioning of the working set, whose nearness to linear dependence is estimated by the ratio of the largest to smallest diagonals of the TQ factor T (the printed value `Cond T`; see Section 12.3). In constructing the initial working set, constraints are excluded that would result in a large value of `Cond T`. Thereafter, nag_opt_lin_lsq (e04ncc) allows constraints to be violated by as much as a user-specified feasibility tolerance (see **options.ftol**, Section 12.2) in order to provide, whenever possible, a *choice* of constraints to be added to the working set at a given iteration. Let α_M denote the maximum step at which $x + \alpha_M p$ does not violate any constraint by more than its feasibility tolerance. All constraints at distance $\alpha(\alpha \leq \alpha_M)$ along p from the current point are then viewed as acceptable candidates for inclusion in the working set. The constraint whose normal makes the largest angle with the search direction is added to the working set. In order to ensure that the new iterate satisfies the constraints in the working set as accurately as possible, the step taken is the exact distance to the newly added constraint. As a consequence, negative steps are occasionally permitted, since the current iterate may violate the constraint to be added by as much as the feasibility tolerance.

12 Optional Arguments

A number of optional input and output arguments to nag_opt_lin_lsq (e04ncc) are available through the structure argument **options**, type Nag_E04_Opt. An argument may be selected by assigning an appropriate value to the relevant structure member; those arguments not selected will be assigned default values. If no use is to be made of any of the optional arguments you should use the NAG defined null pointer, `E04_DEFAULT`, in place of **options** when calling nag_opt_lin_lsq (e04ncc); the default settings will then be used for all arguments.

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

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

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

12.1 Optional Argument Checklist and Default Values

For easy reference, the following list shows the members of **options** which are valid for nag_opt_lin_lsq (e04ncc) together with their default values where relevant. The number ϵ is a generic notation for ***machine precision*** (see nag_machine_precision (X02AJC)).

`Nag_ProblemType prob`	**Nag_LS1**
`Nag_Start start`	Nag_Cold
`Boolean list`	Nag_TRUE
`Nag_PrintType print_level`	**Nag_Soln_Iter**
`char outfile[80]`	`stdout`
`void (*print_fun)()`	**NULL**
`Integer fmax_iter`	$\max(50, 5(\mathbf{n} + \mathbf{nclin}))$
`Integer max_iter`	$\max(50, 5(\mathbf{n} + \mathbf{nclin}))$
`double crash_tol`	0.01
`double ftol`	$\sqrt{\epsilon}$
`double inf_bound`	10^{20}
`double inf_step`	$\max(\mathbf{options.inf\_bound}, 10^{20})$
`double rank_tol`	100ϵ or $10\sqrt{\epsilon}$
`Integer *state`	size $\mathbf{n} + \mathbf{nclin}$
`double *ax`	size **nclin**
`double *lambda`	size $\mathbf{n} + \mathbf{nclin}$
`Boolean hessian`	Nag_FALSE
`Integer iter`	

12.2 Description of the Optional Arguments

prob – Nag_ProblemType Default $=$ Nag_LS1

On entry: specifies the type of objective function to be minimized during the optimality phase. The following are the ten possible values of **options.prob** and the size of the arrays **h**, **kx**, **b** and **cvec** that are required to define the objective function:

Nag_FP	**h**, **b** and **cvec** not referenced;
Nag_LP	**h** and **b** not referenced, **cvec** of size **n**;
Nag_QP1	**h** of size $\mathbf{m} \times \mathbf{tdh}$, symmetric, **b** and **cvec** not referenced;
Nag_QP2	**h** of size $\mathbf{m} \times \mathbf{tdh}$, symmetric, **b** not referenced, **cvec** of size **n**;

Nag_QP3	**h** of size **m** $\times$ **tdh**, upper trapezoidal, **b** and **cvec** not referenced;
Nag_QP4	**h** of size **m** $\times$ **tdh**, upper trapezoidal, **b** not referenced, **cvec** of size **n**.
Nag_LS1	**h** of size **m** $\times$ **tdh**, **b** of size **m**, **cvec** not referenced;
Nag_LS2	**h** of size **m** $\times$ **tdh**, **b** of size **m**, **cvec** of size **n**;
Nag_LS3	**h** of size **m** $\times$ **tdh**, upper trapezoidal, **b** of size **m**, **cvec** not referenced;
Nag_LS4	**h** of size **m** $\times$ **tdh**, upper trapezoidal, **b** of size **m**, **cvec** of size **n**.

The array **kx** of size **n** must be supplied for all problem types but need only be initialized for types Nag_QP3, Nag_QP4, Nag_LS3 and Nag_LS4. If $H = 0$, i.e., the objective function is purely linear, the efficiency of nag_opt_lin_lsq (e04ncc) may be increased by specifying **options.prob** = Nag_LP.

Constraint: **options.prob** = Nag_FP, Nag_LP, Nag_QP1, Nag_QP2, Nag_QP3, Nag_QP4, Nag_LS1, Nag_LS2, Nag_LS3 or Nag_LS4.

start – Nag_Start Default = Nag_Cold

On entry: specifies how the initial working set is chosen. With **options.start** = Nag_Cold, nag_opt_lin_lsq (e04ncc) 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 argument **options.crash_tol**; see below).

With **options.start** = Nag_Warm, you must provide a valid definition of every array element of the optional argument **options.state** (see below). nag_opt_lin_lsq (e04ncc) will override your specification of **options.state** if necessary, so that a poor choice of the working set will not cause a fatal error. For instance, any elements of **options.state** which are set to -2, -1 or 4 will be reset to zero, as will any elements which are set to 3 when the corresponding elements of **bl** and **bu** are not equal. A warm start will be advantageous if a good estimate of the initial working set is available – for example, when nag_opt_lin_lsq (e04ncc) is called repeatedly to solve related problems.

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

list – Nag_Boolean Default = Nag_TRUE

On entry: if **options.list** = Nag_TRUE the argument settings in the call to nag_opt_lin_lsq (e04ncc) will be printed.

print_level – Nag_PrintType Default = Nag_Soln_Iter

On entry: the level of results printout produced by nag_opt_lin_lsq (e04ncc). 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 Ax and the constraint status also printed at each iteration.
Nag_Soln_Iter_Full	As Nag_Soln_Iter_Const with the diagonal elements of the matrix T associated with the TQ factorization (see (4) in Section 11.2) of the working set, and the diagonal elements of the upper triangular matrix R printed at each iteration.

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

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

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

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

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

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

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

See Section 12.3.1 below for further details.

fmax_iter – Integer Default $= \max(50, 5(\mathbf{n} + \mathbf{nclin}))$
max_iter – Integer Default $= \max(50, 5(\mathbf{n} + \mathbf{nclin}))$

On entry: **options.fmax_iter** and **options.max_iter** specify the maximum number of iterations allowed in the feasibility and optimality phase, respectively.

If you wish to check that a call to nag_opt_lin_lsq (e04ncc) is correct before attempting to solve the problem in full then **options.fmax_iter** may be set to 0. No iterations will then be performed but all initialization prior to the first iteration will be done and a listing of argument settings will be output, if optional argument **options.list** = Nag_TRUE (the default setting).

Constraints:

options.fmax_iter ≥ 0;
options.max_iter ≥ 0.

crash_tol – double Default = 0.01

On entry: **options.crash_tol** is used when optional argument **options.start** = Nag_Cold (the default) and nag_opt_lin_lsq (e04ncc) selects an initial working set. The initial working set will include (if possible) bounds or general inequality constraints that lie within **options.crash_tol** of their bounds. In particular, a constraint of the form $a_j^{\mathrm{T}} x \geq l$ will be included in the initial working set if $\left|a_j^{\mathrm{T}} x - l\right| \leq$ **options.crash_tol** $\times (1 + |l|)$.

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

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

On entry: defines the maximum acceptable *absolute* violation in each constraint at a 'feasible' point. For example, if the variables and the coefficients in the general constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify **options.ftol** as 10^{-6}.

nag_opt_lin_lsq (e04ncc) attempts to find a feasible solution before optimizing the objective function. If the sum of infeasibilities cannot be reduced to zero, nag_opt_lin_lsq (e04ncc) finds the minimum value of the sum. Let `Sinf` be the corresponding sum of infeasibilities. If `Sinf` is quite small, it may be appropriate to raise **options.ftol** by a factor of 10 or 100. Otherwise, some error in the data should be suspected.

Note that a 'feasible solution' is a solution that satisfies the current constraints to within the feasibility tolerance **options.ftol**.

Constraint: **options.ftol** > 0.0.

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

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

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

inf_step – double Default $= \max(\textbf{options.inf\_bound}, 10^{20})$

On entry: specifies the magnitude of the change in variables that will be considered a step to an unbounded solution. (Note that an unbounded solution can occur only when the Hessian is singular and the objective contains an explicit linear term.) If the change in x during an iteration would exceed the value of **options.inf_step**, the objective function is considered to be unbounded below in the feasible region.

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

rank_tol – double Default $= 100\epsilon$ or $10\sqrt{\epsilon}$

The default value is 100ϵ for problem types QP1, LS1 and LS3 but is $10\sqrt{\epsilon}$ for other QP and LS problem types. This option does not apply to FP or LP problem types.

On entry: **options.rank_tol** enables you to control the estimate of the triangular factor R_1 (see Section 11.3). If ρ_i denotes the function $\rho_i = \max\{|R_{11}|, |R_{22}|, \dots, |R_{ii}|\}$, the rank of R is defined to be smallest index i such that $|R_{i+1,i+1}| \leq$ **options.rank_tol** $\times |\rho_{i+1}|$.

Constraint: $0.0 <$ **options.rank_tol** < 1.0.

state – Integer * Default memory $= \mathbf{n} + \mathbf{nclin}$

On entry: **options.state** need not be set if the default option of **options.start** = Nag_Cold is used as $\mathbf{n} + \mathbf{nclin}$ values of memory will be automatically allocated by nag_opt_lin_lsq (e04ncc).

If the option **options.start** = Nag_Warm has been chosen, **options.state** must point to a minimum of $\mathbf{n} + \mathbf{nclin}$ elements of memory. This memory will already be available if the **options** structure has been used in a previous call to nag_opt_lin_lsq (e04ncc) from the calling program, with **options.start** = Nag_Cold and the same values of **n** and **nclin**. If a previous call has not been made sufficient memory must be allocated to **options.state** by you.

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

options.state$[j]$	**Meaning**
0	The constraint should *not* be in the initial working set.
1	The constraint should be in the initial working set at its lower bound.
2	The constraint should be in the initial working set at its upper bound.
3	The constraint should be in the initial working set as an equality. This value should 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 **bu** and **bl** are not equal. If nag_opt_lin_lsq (e04ncc) has been called previously with the same values of **n** and **nclin**, **options.state** already contains satisfactory information. (See also the description of the optional argument **options.start**.) The function also adjusts (if necessary) the values supplied in **x** to be consistent with the values supplied in **options.state**.

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

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

options.state$[j]$	**Meaning**
-2	The constraint violates its lower bound by more than the feasibility tolerance.
-1	The constraint violates its upper bound by more than the feasibility tolerance.
0	The constraint is satisfied to within the feasibility tolerance, but is not in the working set.
1	This inequality constraint is included in the working set at its lower bound.
2	This inequality constraint is included in the working set at its upper bound.
3	This constraint is included in the working set as an equality. This value of **options.state** can occur only when **bl**$[j]$ = **bu**$[j]$.
4	This corresponds to optimality being declared with **x**$[j]$ being temporarily fixed at its current value. This value of **options.state** can only occur when **fail.code** = NW_SOLN_NOT_UNIQUE.

ax – double * Default memory = **nclin**

On entry: **nclin** values of memory will be automatically allocated by nag_opt_lin_lsq (e04ncc) and this is the recommended method of use of **options.ax**. However you may supply memory from the calling program.

On exit: if **nclin** > 0, **options.ax** points to the final values of the linear constraints Ax.

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

On entry: **n** + **nclin** values of memory will be automatically allocated by nag_opt_lin_lsq (e04ncc) and this is the recommended method of use of **options.lambda**. However you may supply memory from the calling program.

On exit: the values of the Lagrange multipliers for each constraint with respect to the current working set. The first n elements contain the multipliers for the bound constraints on the variables, and the next n_L elements contain the multipliers for the general linear constraints (if any). If **options.state**$[j-1] = 0$ (i.e., constraint j is not in the working set), **options.lambda**$[j-1]$ is zero. If x is optimal, **options.lambda**$[j-1]$ should be non-negative if **options.state**$[j-1] = 1$, non-positive if **options.state**$[j-1] = 2$ and zero if **options.state**$[j-1] = 4$.

hessian – Nag_Boolean Default = Nag_FALSE

On entry: controls the contents of the argument **h** on return from nag_opt_lin_lsq (e04ncc). nag_opt_lin_lsq (e04ncc) works exclusively with the transformed and reordered matrix H_Q (8), and hence extra computation is required to form the Hessian itself. If the optional argument **options.hessian** = Nag_FALSE, **h** contains the Cholesky factor of the matrix H_Q with columns ordered as indicated by **kx** (see Section 5). If **options.hessian** = Nag_TRUE, **h** contains the Cholesky factor of the Hessian matrix $\nabla^2 F$, with columns ordered as indicated by **kx**.

iter – Integer

On exit: the total number of iterations performed in the feasibility phase and (if appropriate) the optimality phase.

12.3 Description of Printed Output

The level of printed output can be controlled with the structure members **options.list** and **options.print_level** (see Section 12.2). If **options.list** = Nag_TRUE then the argument values to nag_opt_lin_lsq (e04ncc) are listed, whereas the printout of results is governed by the value of **options.print_level**. The default of **options.print_level** = Nag_Soln_Iter 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_lin_lsq (e04ncc).

To aid interpretation of the printed results, the following convention is used for numbering the constraints: indices 1 to n refer to the bounds on the variables, and indices $n+1$ to $n+n_L$ refer to the general constraints.

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

`Itn`	is the iteration count.
`Step`	is the step taken along the computed search direction. If a constraint is added during the current iteration, `Step` will be the step to the nearest constraint. During the optimality phase, the step can be greater than 1.0 only if the factor R_Z is singular (see Section 11.3).
`Ninf`	is the number of violated constraints (infeasibilities). This will be zero during the optimality phase.
`Sinf/Objective`	is the value of the current objective function. If x is not feasible, `Sinf` gives a weighted sum of the magnitudes of constraint violations. If x is feasible, `Objective` is the value of the objective function. The output line for the final iteration of the feasibility phase (i.e., the first iteration for which `Ninf` is zero) will give the value of the true objective at the first feasible point.

During the optimality phase, the value of the objective function will be non-increasing. During the feasibility phase, the number of constraint infeasibilities will not increase until either a feasible point is found, or the optimality of the multipliers implies that no feasible point exists. Once optimal multipliers are obtained, the number of infeasibilities can increase, but the sum of infeasibilities will either remain constant or be reduced until the minimum sum of infeasibilities is found.

`Norm Gz`	$\lVert Z_1^{\mathrm{T}} g_{\mathrm{FR}} \rVert$, the Euclidean norm of the reduced gradient with respect to Z_1 (see Section 11.3). During the optimality phase, this norm will be approximately zero after a unit step.

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

`Jdel`	is the index of the constraint deleted from the working set, along with the designation `L` (lower bound), `U` (upper bound), `E` (equality), `F` (temporarily fixed variable) or `A` (artificial constraint). If `Jdel` is zero, no constraint was deleted.
`Jadd`	is the index of the constraint added to the working set, along with a designation as for `Jdel`. If `Jadd` is zero, no constraint was added.
`Bnd`	is the number of simple bound constraints in the current working set.
`Lin`	is the number of general linear constraints in the current working set.
`Art`	is the number of artificial constraints in the working set, i.e., the number of columns of Z_2 (see Section 11.3).
`Zr`	is the number of columns of Z_1 (see Section 11.2). `Zr` is the dimension of the subspace in which the objective function is currently being minimized. The value of `Zr` is the number of variables minus the number of constraints in the working set; i.e., $\mathtt{Zr} = n - (\mathtt{Bnd} + \mathtt{Lin} + \mathtt{Art})$.

The value of n_Z, the number of columns of Z (see Section 11) can be calculated as $n_Z = n - (\mathtt{Bnd} + \mathtt{Lin})$. A zero value of n_Z implies that x lies at a vertex of the feasible region.

`Norm Gf`	is the Euclidean norm of the gradient function with respect to the free variables, i.e., variables not currently held at a bound.
`Cond T`	is a lower bound on the condition number of the working set.

`Cond Rz` is a lower bound on the condition number of the triangular factor R_1 (the first `Zr` rows and columns of the factor R_Z).

When **options.print_level** = Nag_Soln_Iter_Const or Nag_Soln_Iter_Full more detailed results are given at each iteration. For the setting **options.print_level** = Nag_Soln_Iter_Const additional values output are:

`Value of x` is the value of x currently held in **x**.

`State` is the current value of **options.state** associated with x.

`Value of Ax` is the value of Ax currently held in **options.ax**.

`State` is the current value of **options.state** associated with Ax.

Also printed are the Lagrange Multipliers for the bound constraints, linear constraints and artificial constraints.

If **options.print_level** = Nag_Soln_Iter_Full then the diagonals of T and R are also output at each iteration.

When **options.print_level** = Nag_Soln, Nag_Soln_Iter, Nag_Soln_Iter_Long, Nag_Soln_Iter_Const or Nag_Soln_Iter_Full the final printout from nag_opt_lin_lsq (e04ncc) includes a listing of the status of every variable and constraint. The following describes the printout for each variable.

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

`State` gives the state of the variable (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than the optional argument **options.ftol** (default value $\sqrt{\epsilon}$, where ϵ is the ***machine precision***; see Section 12.2), `State` will be `++` or `--` respectively.

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

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

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

`I` *Infeasible*. The variable is currently violating one of its bounds by more than **options.ftol**.

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

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

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

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

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

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

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

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

If **options**.**print_level** = Nag_NoPrint then printout will be suppressed; you can print the final solution when nag_opt_lin_lsq (e04ncc) returns to the calling program.

12.3.1 Output of results via a user-defined printing function

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

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

The rest of this section can be skipped if you wish to use the default printing facilities.

When a user-defined function is assigned to **options**.**print_fun** this will be called in preference to the internal print function of nag_opt_lin_lsq (e04ncc). Calls to the user-defined function are again controlled by means of the **options**.**print_level** member. Information is provided through **st** and **comm**, the two structure arguments to **options**.**print_fun**.

If **comm**→**it_prt** = Nag_TRUE then the results from the last iteration of nag_opt_lin_lsq (e04ncc) are provided through **st**. Note that **options**.**print_fun** will be called with **comm**→**it_prt** = Nag_TRUE only if **options**.**print_level** = Nag_Iter, Nag_Iter_Long, 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.

iter – Integer

The iteration count.

jdel – Integer

Index of constraint deleted from the working set.

jadd – Integer

Index of constraint added to the working set.

step – double

The step taken along the computed search direction.

ninf – Integer

The number of violated constraints (infeasibilities).

f – double

The current value of the objective function if **st**→**ninf** = 0; otherwise, **st**→**f** is a weighted sum of the magnitudes of constraint violations.

bnd – Integer

Number of bound constraints in the working set.

lin – Integer

Number of general linear constraints in the working set.

nart – Integer

Number of artificial constraints in the working set (see Section 11.3).

nrank – Integer

The rank of the upper triangular matrix R (see Section 11.3).

nrz – Integer

Number of columns of Z_1 (see Section 11.2).

norm_gz – double

Euclidean norm of the reduced gradient, $\left\|Z_1^{\mathrm{T}} g_{\mathrm{FR}}\right\|$ (see Section 11.3).

norm_gf – double

Euclidean norm of the gradient function with respect to the free variables.

cond_t – double

A lower bound on the condition number of the working set.

cond_r – double

A lower bound on the condition number of the triangular factor R_1 (see Section 11.3).

x – double *

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

ax – double *

If **st**→**nclin** > 0, the **st**→**nclin** components of the linear constraints Ax.

state – Integer *

options.**state** contains the status of the **st**→**n** variables and **st**→**nclin** general linear constraints. See Section 12.2 for a description of the possible status values.

diagt – double *

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

diagr – double *

If **st**→**nrank** > 0, the first **st**→**nrank** elements of the diagonal of the upper triangular matrix R.

If **comm**→**new_lm** = Nag_TRUE then the Lagrange multipliers have been updated and the following members of **st** are set:

bnd – Integer

The number of bound constraints in the working set.

kx – Integer *
bclambda – double *

Indices of the bound constraints in the working set, with associated multipliers. **st**→**kx**$[i]$ is the index of the constraint with multiplier **st**→**bclambda**$[i]$, for $i = 0, 1, \ldots,$ **st**→**bnd** $- 1$.

lin – Integer

The number of linear constraints in the working set.

kactive – Integer *
lambda – double *

Indices of the linear constraints in the working set, with associated multipliers. **st**→**kactive**$[i]$ is the index of the constraint with multiplier **st**→**lambda**[**st**→**bnd** $+ i]$, for $i = 0, 1, \ldots,$ **st**→**lin** $- 1$.

nart – Integer

The number of artificial constraints in the working set (see Section 11.3).

gq – double *

st→**gq**$[i]$, for $i = 0, 1, \ldots,$ **st**→**nart** $- 1$, hold the multipliers for the artificial constraints.

If **comm**→**sol_prt** = Nag_TRUE then the final result from nag_opt_lin_lsq (e04ncc) is available and the following members of **st** are set:

n – Integer

The number of variables.

nclin – Integer

The number of linear constraints.

iter – Integer

The iteration count.

x – double *

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

feasible – Nag_Boolean

Will be Nag_TRUE if the final point is feasible.

f – double

The final value of the objective function if **st**→**feasible** is Nag_TRUE; otherwise, the sum of infeasibilities. If the problem is of type FP and x is feasible then **st**→**f** is set to zero.

ax – double *

If **st**→**nclin** > 0, the **st**→**nclin** components of the final linear constraint activities, Ax.

state – Integer *

Contains the final status of the **st**→**n** variables and **st**→**nclin** general linear constraints. See Section 12.2 for a description of the possible status values.

lambda – double *

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

bl – double *

Contains the **st**→**n** + **st**→**nclin** lower bounds.

bu – double *

Contains the **st**→**n** + **st**→**nclin** upper bounds.

endstate – Nag_EndState

The state of termination of nag_opt_lin_lsq (e04ncc). Possible values of **st**→**endstate** and their correspondence to the exit value of **fail** are:

Value of **st**→**endstate**	Value of **fail**
Nag_Feasible or Nag_Optimal	NE_NOERROR
Nag_Weakmin	NW_SOLN_NOT_UNIQUE
Nag_Unbounded	NE_UNBOUNDED
Nag_Infeasible	NW_NOT_FEASIBLE

Nag_Too_Many_Iter	NW_TOO_MANY_ITER
Nag_Cycling	NE_CYCLING

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

it_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with the result of the current iteration.

sol_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with the final result.

new_lm – Nag_Boolean

Will be Nag_TRUE when the Lagrange multipliers have been updated.

user – double *
iuser – Integer *
p – Pointer

Pointers for communication of user information. If used they must be allocated memory either before entry to nag_opt_lin_lsq (e04ncc) or during a call to **options.print_fun**. The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.