NAG Library Function Document

nag_opt_sparse_convex_qp_solve (e04nqc)

Note: *this function uses* **optional arguments** *to define choices in the problem specification and in the details of the algorithm. If you wish to use* default *settings for all of the optional arguments, you need only read Sections 1 to 10 of this document. If, however, you wish to reset some or all of the settings please refer to Section 11 for a detailed description of the algorithm, to Section 12 for a detailed description of the specification of the optional arguments and to Section 13 for a detailed description of the monitoring information produced by the function.*

1 Purpose

nag_opt_sparse_convex_qp_solve (e04nqc) solves sparse linear programming or convex quadratic programming problems. The initialization function nag_opt_sparse_convex_qp_init (e04npc) **must** have been called before calling nag_opt_sparse_convex_qp_solve (e04nqc).

2 Specification

```
#include <nag.h>
#include <nage04.h>
void nag_opt_sparse_convex_qp_solve (Nag_Start start,
     void (*qphx)(Integer ncolh, const double x[], double hx[],
          Integer nstate),
     Integer m, Integer n, Integer ne, Integer nname, Integer lenc,
     Integer ncolh, Integer iobj, double objadd, const char *prob,
     const double acol[], const Integer inda[], const Integer loca[],
     const double bl[], const double bu[], const double c[],
     const char *names[], const Integer helast[], Integer hs[], double x[],
     double pi[], double rc[], Integer *ns, Integer *ninf, double *sinf,
     double *obj, Nag_E04State *state, Nag_Comm *comm, NagError *fail)
```

Before calling nag_opt_sparse_convex_qp_solve (e04nqc) or one of the option setting functions

nag_opt_sparse_convex_qp_option_set_file (e04nrc)

nag_opt_sparse_convex_qp_option_set_string (e04nsc)

nag_opt_sparse_convex_qp_option_set_integer (e04ntc) or

nag_opt_sparse_convex_qp_option_set_double (e04nuc),

nag_opt_sparse_convex_qp_init (e04npc) **must** be called.

The specification for nag_opt_sparse_convex_qp_init (e04npc) is:

```
#include <nag.h>
#include <nage04.h>
void nag_opt_sparse_convex_qp_init (Nag_E04State *state, NagError *fail)
```

After calling nag_opt_sparse_convex_qp_solve (e04nqc) you can call one or both of the functions

nag_opt_sparse_convex_qp_option_get_integer (e04nxc) or

nag_opt_sparse_convex_qp_option_get_double (e04nyc)

to obtain the current value of an optional argument.

3 Description

nag_opt_sparse_convex_qp_solve (e04nqc) is designed to solve large-scale *linear* or *quadratic programming problems* of the form:

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

where x is an n-vector of variables, l and u are constant lower and upper bounds, A is an m by n sparse matrix and $f(x)$ is a linear or quadratic objective function that may be specified in a variety of ways, depending upon the particular problem being solved. The optional argument **Maximize** may be used to specify a problem in which $f(x)$ is maximized instead of minimized.

Upper and lower bounds are specified for all variables and constraints. This form allows full generality in specifying various types of constraint. In particular, the jth constraint may be defined as an equality by setting $l_j = u_j$. If certain bounds are not present, the associated elements of l or u may be set to special values that are treated as $-\infty$ or $+\infty$.

The possible forms for the function $f(x)$ are summarised in Table 1. The most general form for $f(x)$ is

$$f(x) = q + c^{\mathrm{T}}x + \frac{1}{2}x^{\mathrm{T}}Hx = q + \sum_{j=1}^{n} c_j x_j + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} x_i H_{ij} x_j$$

where q is a constant, c is a constant n-vector and H is a constant symmetric n by n matrix with elements $\{H_{ij}\}$. In this form, f is a quadratic function of x and (1) is known as a *quadratic program* (QP). nag_opt_sparse_convex_qp_solve (e04nqc) is suitable for all *convex* quadratic programs. The defining feature of a *convex* QP is that the matrix H must be *positive semidefinite*, i.e., it must satisfy $x^{\mathrm{T}}Hx \geq 0$ for all x. If not, $f(x)$ is nonconvex and nag_opt_sparse_convex_qp_solve (e04nqc) will terminate with the error indicator **fail.code** = NE_HESS_INDEF. If $f(x)$ is nonconvex it may be more appropriate to call nag_opt_sparse_nlp_solve (e04vhc) instead.

Problem type	Objective function $f(x)$	Hessian matrix H
FP	Not applicable	$q = c = H = 0$
LP	$q + c^{\mathrm{T}}x$	$H = 0$
QP	$q + c^{\mathrm{T}}x + \frac{1}{2}x^{\mathrm{T}}Hx$	Symmetric positive semidefinite

Table 1
Choices for the objective function $f(x)$

If $H = 0$, then $f(x) = q + c^{\mathrm{T}}x$ and the problem is known as a *linear program* (LP). In this case, rather than defining an H with zero elements, you can define H to have no columns by setting **ncolh** $= 0$ (see Section 5).

If $H = 0$, $q = 0$, and $c = 0$, there is no objective function and the problem is a *feasible point problem* (FP), which is equivalent to finding a point that satisfies the constraints on x. In the situation where no feasible point exists, several options are available for finding a point that minimizes the constraint violations (see the description of the optional argument **Elastic Mode**).

nag_opt_sparse_convex_qp_solve (e04nqc) is suitable for large LPs and QPs in which the matrix A is *sparse*, i.e., when the number of zero elements is sufficiently large that it is worthwhile using algorithms which avoid computations and storage involving zero elements. The matrix A is input to nag_opt_sparse_convex_qp_solve (e04nqc) by means of the three array arguments **acol**, **inda** and **loca**. This allows you to specify the pattern of nonzero elements in A.

nag_opt_sparse_convex_qp_solve (e04nqc) exploits structure in H by requiring H to be defined *implicitly* in a function that computes the product Hx for any given vector x. In many cases, the product Hx can be computed very efficiently for any given x, e.g., H may be a sparse matrix, or a sum of matrices of rank-one.

For problems in which A can be treated as a *dense* matrix, it is usually more efficient to use nag_opt_lp (e04mfc), nag_opt_lin_lsq (e04ncc) or nag_opt_qp (e04nfc).

There is considerable flexibility allowed in the definition of $f(x)$ in Table 1. The vector c defining the linear term $c^{\mathrm{T}}x$ can be input in three ways: as a sparse row of A; as an explicit dense vector c; or as both a sparse row and an explicit vector (in which case, $c^{\mathrm{T}}x$ will be the sum of two linear terms). When stored in A, c is the **iobj**th row of A, which is known as the *objective row*. The objective row must always be a *free* row of A in the sense that its lower and upper bounds must be $-\infty$ and $+\infty$. Storing c as part of A is recommended if c is a sparse vector. Storing c as an explicit vector is recommended for a sequence of problems, each with a different objective (see arguments **c** and **lenc**).

The upper and lower bounds on the m elements of Ax are said to define the *general constraints* of the problem. Internally, nag_opt_sparse_convex_qp_solve (e04nqc) converts the general constraints to equalities by introducing a set of *slack variables* s, where $s = (s_1, s_2, \ldots, s_m)^{\mathrm{T}}$. For example, the linear constraint $5 \leq 2x_1 + 3x_2 \leq +\infty$ is replaced by $2x_1 + 3x_2 - s_1 = 0$, together with the bounded slack $5 \leq s_1 \leq +\infty$. The problem defined by (1) can therefore be re-written in the following equivalent form:

$$\underset{x \in R^n, s \in R^m}{\text{minimize}} f(x) \quad \text{subject to } Ax - s = 0, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u.$$

Since the slack variables s are subject to the same upper and lower bounds as the elements of Ax, the bounds on x and Ax can simply be thought of as bounds on the combined vector (x, s). (In order to indicate their special role in QP problems, the original variables x are sometimes known as 'column variables', and the slack variables s are known as 'row variables'.)

Each LP or QP problem is solved using a two-phase iterative procedure (in which the general constraints are satisfied throughout): a *feasibility phase* (*Phase 1*), in which the sum of infeasibilities with respect to the bounds on x and s is minimized to find a feasible point that satisfies all constraints within a specified feasibility tolerance; and an *optimality phase* (*Phase 2*), in which $f(x)$ is minimized (or maximized) by constructing a sequence of iterates that lies within the feasible region.

Phase 1 involves solving a linear program of the form

Phase 1

$$\underset{x,s,v,w}{\text{minimize}} \sum_{j=1}^{n+m} (v_j + w_j)$$

$$\text{subject to } Ax - s = 0, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} - v + w \leq u, \quad v \geq 0, \quad w \geq 0$$

which is equivalent to minimizing the sum of the constraint violations. If the constraints are feasible (i.e., at least one feasible point exists), eventually a point will be found at which both v and w are zero. Then the associated value of (x, s) satisfies the original constraints and is used as the starting point for the Phase 2 iterations for minimizing $f(x)$.

If the constraints are infeasible (i.e., $v \neq 0$ or $w \neq 0$ at the end of Phase 1), no solution exists for (1) and you have the option of either terminating or continuing in so-called *elastic mode* (see the discussion of the optional argument **Elastic Mode**). In elastic mode, a 'relaxed' or 'perturbed' problem is solved in which $f(x)$ is minimized while allowing some of the bounds to become 'elastic', i.e., to change from their specified values. Variables subject to elastic bounds are known as *elastic variables*. An elastic variable is free to violate one or both of its original upper or lower bounds. You are able to assign which bounds will become elastic if elastic mode is ever started (see the argument **helast** in Section 5).

To make the relaxed problem meaningful, nag_opt_sparse_convex_qp_solve (e04nqc) minimizes $f(x)$ while (in some sense) finding the 'smallest' violation of the elastic variables. In the situation where all the variables are elastic, the relaxed problem has the form

Phase 2 (γ)

$$\underset{x,s,v,w}{\text{minimize}} f(x) + \gamma \sum_{j=1}^{n+m} (v_j + w_j)$$

$$\text{subject to } Ax - s = 0, \quad l \leq \begin{pmatrix} x \\ s \end{pmatrix} - v + w \leq u, \quad v \geq 0, \quad w \geq 0,$$

where γ is a non-negative argument known as the *elastic weight* (see the description of the optional argument **Elastic Weight**), and $f(x) + \gamma \sum_j (v_j + w_j)$ is called the *composite objective*. In the more general situation where only a subset of the bounds are elastic, the v's and w's for the non-elastic bounds are fixed at zero.

The elastic weight can be chosen to make the composite objective behave like the original objective $f(x)$, the sum of infeasibilities, or anything in-between. If $\gamma = 0$, nag_opt_sparse_convex_qp_solve (e04nqc) will attempt to minimize f subject to the (true) upper and lower bounds on the non-elastic variables (and declare the problem infeasible if the non-elastic variables cannot be made feasible).

At the other extreme, choosing γ sufficiently large will have the effect of minimizing the sum of the violations of the elastic variables subject to the original constraints on the non-elastic variables. Choosing a large value of the elastic weight is useful for defining a 'least-infeasible' point for an infeasible problem.

In Phase 1 and elastic mode, all calculations involving v and w are done implicitly in the sense that an elastic variable x_j is allowed to violate its lower bound (say) and an explicit value of v can be recovered as $v_j = l_j - x_j$.

A constraint is said to be *active* or *binding* at x if the associated element of either x or Ax is equal to one of its upper or lower bounds. Since an active constraint in Ax has its associated slack variable at a bound, the status of both simple and general upper and lower bounds can be conveniently described in terms of the status of the variables (x, s). A variable is said to be *nonbasic* if it is temporarily fixed at its upper or lower bound. It follows that regarding a general constraint as being *active* is equivalent to thinking of its associated slack as being *nonbasic*.

At each iteration of an active-set method, the constraints $Ax - s = 0$ are (conceptually) partitioned into the form

$$Bx_B + Sx_S + Nx_N = 0,$$

where x_N consists of the nonbasic elements of (x, s) and the *basis matrix* B is square and nonsingular. The elements of x_B and x_S are called the *basic* and *superbasic* variables respectively; with x_N they are a permutation of the elements of x and s. At a QP solution, the basic and superbasic variables will lie somewhere between their upper or lower bounds, while the nonbasic variables will be equal to one of their bounds. At each iteration, x_S is regarded as a set of independent variables that are free to move in any desired direction, namely one that will improve the value of the objective function (or sum of infeasibilities). The basic variables are then adjusted in order to ensure that (x, s) continues to satisfy $Ax - s = 0$. The number of superbasic variables (n_S say) therefore indicates the number of degrees of freedom remaining after the constraints have been satisfied. In broad terms, n_S is a measure of *how nonlinear* the problem is. In particular, n_S will always be zero for FP and LP problems.

If it appears that no improvement can be made with the current definition of B, S and N, a nonbasic variable is selected to be added to S, and the process is repeated with the value of n_S increased by one. At all stages, if a basic or superbasic variable encounters one of its bounds, the variable is made nonbasic and the value of n_S is decreased by one.

Associated with each of the m equality constraints $Ax - s = 0$ is a *dual variable* π_i. Similarly, each variable in (x, s) has an associated *reduced gradient* d_j (also known as a *reduced cost*). The reduced gradients for the variables x are the quantities $g - A^{\mathrm{T}}\pi$, where g is the gradient of the QP objective function, and the reduced gradients for the slack variables s are the dual variables π. The QP subproblem is optimal if $d_j \geq 0$ for all nonbasic variables at their lower bounds, $d_j \leq 0$ for all nonbasic variables at their upper bounds and $d_j = 0$ for all superbasic variables. In practice, an *approximate* QP solution is

found by slightly relaxing these conditions on d_j (see the description of the optional argument **Optimality Tolerance**).

The process of computing and comparing reduced gradients is known as *pricing* (a term first introduced in the context of the simplex method for linear programming). To 'price' a nonbasic variable x_j means that the reduced gradient d_j associated with the relevant active upper or lower bound on x_j is computed via the formula $d_j = g_j - a_j^{\mathrm{T}}\pi$, where a_j is the jth column of $\begin{pmatrix} A & -I \end{pmatrix}$. (The variable selected by such a process and the corresponding value of d_j (i.e., its reduced gradient) are the quantities `+SBS` and `dj` in the monitoring file output; see Section 9.1.) If A has significantly more columns than rows (i.e., $n \gg m$), pricing can be computationally expensive. In this case, a strategy known as *partial pricing* can be used to compute and compare only a subset of the d_js.

nag_opt_sparse_convex_qp_solve (e04nqc) is based on SQOPT, which is part of the SNOPT package described in Gill *et al.* (2005a). It uses stable numerical methods throughout and includes a reliable basis package (for maintaining sparse LU factors of the basis matrix B), a practical anti-degeneracy procedure, efficient handling of linear constraints and bounds on the variables (by an active-set strategy), as well as automatic scaling of the constraints. Further details can be found in Section 11.

4 References

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

Gill P E and Murray W (1978) Numerically stable methods for quadratic programming *Math. Programming* **14** 349–372

Gill P E, Murray W and Saunders M A (1995) User's guide for QPOPT 1.0: a Fortran package for quadratic programming *Report SOL 95-4* Department of Operations Research, Stanford University

Gill P E, Murray W and Saunders M A (2005a) Users' guide for SQOPT 7: a Fortran package for large-scale linear and quadratic programming *Report NA 05-1* Department of Mathematics, University of California, San Diego http://www.ccom.ucsd.edu/~peg/papers/sqdoc7.pdf

Gill P E, Murray W and Saunders M A (2005b) Users' guide for SNOPT 7.1: a Fortran package for large-scale linear nonlinear programming *Report NA 05-2* Department of Mathematics, University of California, San Diego http://www.ccom.ucsd.edu/~peg/papers/sndoc7.pdf

Gill P E, Murray W, Saunders M A and Wright M H (1987) Maintaining LU factors of a general sparse matrix *Linear Algebra and its Applics.* **88/89** 239–270

Gill P E, Murray W, Saunders M A and Wright M H (1989) A practical anti-cycling procedure for linearly constrained optimization *Math. Programming* **45** 437–474

Gill P E, Murray W, Saunders M A and Wright M H (1991) Inertia-controlling methods for general quadratic programming *SIAM Rev.* **33** 1–36

Hall J A J and McKinnon K I M (1996) The simplest examples where the simplex method cycles and conditions where EXPAND fails to prevent cycling *Report MS* 96–100 Department of Mathematics and Statistics, University of Edinburgh

5 Arguments

The first n entries of the arguments **bl**, **bu**, **hs** and **x** refer to the variables x. The last m entries refer to the slacks s.

1: **start** – Nag_Start *Input*

On entry: indicates how a starting basis (and certain other items) will be obtained.

start = Nag_Cold
Requests that an internal Crash procedure be used to choose an initial basis, unless a Basis file is provided via optional arguments **Old Basis File**, **Insert File** or **Load File**.

start = Nag_BasisFile
Is the same as **start** = Nag_Cold but is more meaningful when a Basis file is given.

start = Nag_Warm
Means that a basis is already defined in **hs** and a start point is already defined in **x** (probably from an earlier call).

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

2: **qphx** – function, supplied by the user *External Function*

For QP problems, you must supply a version of **qphx** to compute the matrix product Hx for a given vector x. If H has rows and columns of zeros, it is most efficient to order x so that the nonlinear variables appear first. For example, if $x = (y, z)^{\mathrm{T}}$ and only y enters the objective quadratically then

$$Hx = \begin{pmatrix} H_1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} H_1 y \\ 0 \end{pmatrix}. \tag{2}$$

In this case, **ncolh** should be the dimension of y, and **qphx** should compute $H_1 y$. For FP and LP problems, **qphx** will never be called by nag_opt_sparse_convex_qp_solve (e04nqc) and hence **qphx** may be specified as **NULLFN**.

The specification of **qphx** is:

```
void qphx (Integer ncolh, const double x[], double hx[],
     Integer nstate)
```

1: **ncolh** – Integer *Input*

On entry: this is the same argument **ncolh** as supplied to nag_opt_sparse_convex_qp_solve (e04nqc).

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

On entry: the first **ncolh** elements of the vector x.

3: **hx**[**ncolh**] – double *Output*

On exit: the product Hx. If **ncolh** is less than the input argument **n**, Hx is really the product $H_1 y$ in (2).

4: **nstate** – Integer *Input*

On entry: allows you to save computation time if certain data must be read or calculated only once. To preserve this data for a subsequent calculation place it in **comm**.

nstate = 1
nag_opt_sparse_convex_qp_solve (e04nqc) is calling **qphx** for the first time.

nstate = 0
There is nothing special about the current call of **qphx**.

nstate $\geq$ 2
nag_opt_sparse_convex_qp_solve (e04nqc) is calling **qphx** for the last time. This argument setting allows you to perform some additional computation on the final solution.

nstate = 2
The current x is optimal.

nstate = 3
The problem appears to be infeasible.

nstate = 4
The problem appears to be unbounded.

nstate = 5
The iterations limit was reached.

5: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_opt_sparse_convex_qp_solve (e04nqc) you may allocate memory and initialize these pointers with various quantities for use by **qphx** when called from nag_opt_sparse_convex_qp_solve (e04nqc) (see Section 3.2.1.1 in the Essential Introduction).

3: **m** – Integer *Input*

On entry: m, the number of general linear constraints (or slacks). This is the number of rows in the linear constraint matrix A, including the free row (if any; see **iobj**). Note that A must have at least one row. If your problem has no constraints, or only upper or lower bounds on the variables, then you must include a dummy row with sufficiently wide upper and lower bounds (see also **acol**, **inda** and **loca**).

Constraint: $\mathbf{m} \geq 1$.

4: **n** – Integer *Input*

On entry: n, the number of variables (excluding slacks). This is the number of columns in the linear constraint matrix A.

Constraint: $\mathbf{n} \geq 1$.

5: **ne** – Integer *Input*

On entry: the number of nonzero elements in A.

Constraint: $1 \leq \mathbf{ne} \leq \mathbf{n} \times \mathbf{m}$.

6: **nname** – Integer *Input*

On entry: the number of column (i.e., variable) and row names supplied in the array **names**.

nname $= 1$
There are no names. Default names will be used in the printed output.

nname $= \mathbf{n} + \mathbf{m}$
All names must be supplied.

Constraint: **nname** $= 1$ or $\mathbf{n} + \mathbf{m}$.

7: **lenc** – Integer *Input*

On entry: the number of elements in the constant objective vector c.

If **lenc** > 0, the first **lenc** elements of x belong to variables corresponding to the constant objective term c.

Constraint: $0 \leq \mathbf{lenc} \leq \mathbf{n}$.

8: **ncolh** – Integer *Input*

On entry: n_H, the number of leading nonzero columns of the Hessian matrix H. For FP and LP problems, **ncolh** must be set to zero.

The first **ncolh** elements of x belong to variables corresponding to the nonzero block of the QP Hessian.

Constraint: $0 \leq \mathbf{ncolh} \leq \mathbf{n}$.

9: **iobj** – Integer *Input*

On entry: if **iobj** > 0, row **iobj** of A is a free row containing the nonzero elements of the vector c appearing in the linear objective term $c^{\mathrm{T}}x$.

If **iobj** $= 0$, there is no free row, and the linear objective vector should be supplied in array **c**.

Constraint: $0 \leq$ **iobj** $\leq$ **m**.

10: **objadd** – double *Input*

On entry: the constant q, to be added to the objective for printing purposes. Typically **objadd** $= 0.0$.

11: **prob** – const char * *Input*

On entry: the name for the problem. It is used in the printed solution and in some functions that output Basis files. Only the first eight characters of **prob** are significant.

12: **acol**[**ne**] – const double *Input*

On entry: the nonzero elements of A, ordered by increasing column index. Note that all elements must be assigned a value in the calling program.

13: **inda**[**ne**] – const Integer *Input*

On entry: **inda**$[i-1]$ must contain the row index of the nonzero element stored in **acol**$[i-1]$, for $i = 1, 2, \ldots,$ **ne**. Thus a pair of values (**acol**$[i-1]$, **inda**$[i-1]$) contains a matrix element and its corresponding row index.

If **lenc** > 0, the first **lenc** elements of **acol** and **inda** belong to variables corresponding to the constant objectiver term c.

If the problem has a quadratic objective, the first **ncolh** columns of **acol** and **inda** belong to variables corresponding to the nonzero block of the QP Hessian. Function **qphx** knows about these variables.

Note that the row indices for a column must lie in the range 1 to **m**, and may be supplied in any order.

Constraint: $1 \leq$ **inda**$[i-1] \leq$ **m**, for $i = 1, 2, \ldots,$ **ne**.

14: **loca**[**n** + **1**] – const Integer *Input*

On entry: **loca**$[j-1]$ must contain the value $p+1$, where p is the index in **acol** and **inda** of the start of the jth column, for $j = 1, 2, \ldots,$ **n**. Thus, the entries of column j are held in **acol**$[i-1]$, and their corresponding row indices are in **inda**$[i-1]$, for $i = k, \ldots, l$, where $k =$ **loca**$[j-1]$ and $l =$ **loca**$[j] - 1$. To specify the jth column as empty, set **loca**$[j-1] =$ **loca**$[j]$. Note that the first and last elements of **loca** must be **loca**$[0] = 1$ and **loca**[**n**] = **ne** $+ 1$. If your problem has no constraints, or just bounds on the variables, you may include a dummy 'free' row with a single (zero) element by setting **ne** $= 1$, **acol**$[0] = 0.0$, **inda**$[0] = 1$, **loca**$[0] = 1$, and **loca**$[j-1] = 2$, for $j = 1, 2 \ldots,$ **n**. This row is made 'free' by setting its bounds to be **bl**[**n**] $= -bigbnd$ and **bu**[**n**] $= bigbnd$, where $bigbnd$ is the value of the optional argument **Infinite Bound Size**.

Constraints:

loca$[0] = 1$;
loca$[j-1] \geq 1$, for $j = 2, 3, \ldots,$ **n**;
loca[**n**] = **ne** $+ 1$;
$0 \leq$ **loca**$[j] -$ **loca**$[j-1] \leq$ **m**, for $j = 1, 2, \ldots,$ **n**.

15: **bl**[**n** + **m**] – const double *Input*

On entry: l, the lower bounds for all the variables and general constraints, in the following order. The first **n** elements of **bl** must contain the bounds on the variables x, and the next **m** elements the bounds for the general linear constraints Ax (which, equivalently, are the bounds for the slacks, s)

and the free row (if any). To fix the jth variable, set $\mathbf{bl}[j-1] = \mathbf{bu}[j-1] = \beta$, say, where $|\beta| < bigbnd$. To specify a nonexistent lower bound (i.e., $l_j = -\infty$), set $\mathbf{bl}[j-1] \leq -bigbnd$. Here, $bigbnd$ is the value of the optional argument **Infinite Bound Size**. To specify the jth constraint as an *equality*, set $\mathbf{bl}[\mathbf{n}+j-1] = \mathbf{bu}[\mathbf{n}+j-1] = \beta$, say, where $|\beta| < bigbnd$. Note that the lower bound corresponding to the free row must be set to $-\infty$ and stored in $\mathbf{bl}[\mathbf{n}+\mathbf{iobj}-1]$.

Constraint: if $\mathbf{iobj} > 0$, $\mathbf{bl}[\mathbf{n}+\mathbf{iobj}-1] \leq -bigbnd$

(See also the description for **bu**.)

16: **bu**[**n** + **m**] – const double *Input*

On entry: u, the upper bounds for all the variables and general constraints, in the following order. The first **n** elements of **bu** must contain the bounds on the variables x, and the next **m** elements the bounds for the general linear constraints Ax (which, equivalently, are the bounds for the slacks, s) and the free row (if any). To specify a nonexistent upper bound (i.e., $u_j = +\infty$), set $\mathbf{bu}[j-1] \geq bigbnd$. Note that the upper bound corresponding to the free row must be set to $+\infty$ and stored in $\mathbf{bu}[\mathbf{n}+\mathbf{iobj}-1]$.

Constraints:

if $\mathbf{iobj} > 0$, $\mathbf{bu}[\mathbf{n}+\mathbf{iobj}-1] \geq bigbnd$;
otherwise $\mathbf{bl}[i-1] \leq \mathbf{bu}[i-1]$.

17: **c**[**lenc**] – const double *Input*

On entry: contains the explicit objective vector c (if any). If $\mathbf{lenc} = 0$, then **c** is not referenced and may be **NULL**.

18: **names**[**nname**] – const char * *Input*

On entry: the optional column and row names, respectively.

If $\mathbf{nname} = 1$, **names** is not referenced and the printed output will use default names for the columns and rows.

If $\mathbf{nname} = \mathbf{n} + \mathbf{m}$, the first **n** elements must contain the names for the columns and the next **m** elements must contain the names for the rows. Note that the name for the free row (if any) must be stored in $\mathbf{names}[\mathbf{n}+\mathbf{iobj}-1]$.

Note: that only the first eight characters of the strings in **names** are significant.

19: **helast**[**n** + **m**] – const Integer *Input*

On entry: defines which variables are to be treated as being elastic in elastic mode. The allowed values of **helast** are:

$\mathbf{helast}[j-1]$	**Status in elastic mode**
0	Variable j is non-elastic and cannot be infeasible
1	Variable j can violate its lower bound
2	Variable j can violate its upper bound
3	Variable j can violate either its lower or upper bound

helast need not be assigned if optional argument $\mathbf{Elastic\ Mode} = 0$.

Constraint: if $\mathbf{Elastic\ Mode} \neq 0$, $\mathbf{helast}[j-1] = 0, 1, 2, 3$, for $j = 1, 2, \ldots, \mathbf{n}+\mathbf{m}$.

20: **hs**[**n** + **m**] – Integer *Input/Output*

On entry: if **start** = Nag_Cold or Nag_BasisFile, and a Basis file of some sort is to be input (see the description of the optional arguments **Old Basis File**, **Insert File** or **Load File**), then **hs** and **x** need not be set at all.

If **start** = Nag_Cold or Nag_BasisFile and there is no Basis file, the first **n** elements of **hs** and **x** must specify the initial states and values, respectively, of the variables x. (The slacks s need not

be initialized.) An internal Crash procedure is then used to select an initial basis matrix B. The initial basis matrix will be triangular (neglecting certain small elements in each column). It is chosen from various rows and columns of $\begin{pmatrix} A & -I \end{pmatrix}$. Possible values for $\mathbf{hs}[j-1]$ are as follows:

$\mathbf{hs}[j-1]$	**State of** $\mathbf{x}[j-1]$ **during Crash procedure**
0 or 1	Eligible for the basis
2	Ignored
3	Eligible for the basis (given preference over 0 or 1)
4 or 5	Ignored

If nothing special is known about the problem, or there is no wish to provide special information, you may set $\mathbf{hs}[j-1] = 0$ and $\mathbf{x}[j-1] = 0.0$, for $j = 1, 2, \ldots, \mathbf{n}$. All variables will then be eligible for the initial basis. Less trivially, to say that the jth variable will probably be equal to one of its bounds, set $\mathbf{hs}[j-1] = 4$ and $\mathbf{x}[j-1] = \mathbf{bl}[j-1]$ or $\mathbf{hs}[j-1] = 5$ and $\mathbf{x}[j-1] = \mathbf{bu}[j-1]$ as appropriate.

Following the Crash procedure, variables for which $\mathbf{hs}[j-1] = 2$ are made superbasic. Other variables not selected for the basis are then made nonbasic at the value $\mathbf{x}[j-1]$ if $\mathbf{bl}[j-1] \leq \mathbf{x}[j-1] \leq \mathbf{bu}[j-1]$, or at the value $\mathbf{bl}[j-1]$ or $\mathbf{bu}[j-1]$ closest to $\mathbf{x}[j-1]$.

If **start** = Nag_Warm, **hs** and **x** must specify the initial states and values, respectively, of the variables and slacks (x, s). If nag_opt_sparse_convex_qp_solve (e04nqc) has been called previously with the same values of **n** and **m**, **hs** already contains satisfactory information.

Constraints:

if **start** = Nag_Cold or Nag_BasisFile, $0 \leq \mathbf{hs}[j-1] \leq 5$, for $j = 1, 2, \ldots, \mathbf{n}$;
if **start** = Nag_Warm, $0 \leq \mathbf{hs}[j-1] \leq 3$, for $j = 1, 2, \ldots, \mathbf{n} + \mathbf{m}$.

On exit: the final states of the variables and slacks (x, s). The significance of each possible value of $\mathbf{hs}[j-1]$ is as follows:

$\mathbf{hs}[j-1]$	**State of variable** j	**Normal value of** $\mathbf{x}[j-1]$
0	Nonbasic	$\mathbf{bl}[j-1]$
1	Nonbasic	$\mathbf{bu}[j-1]$
2	Superbasic	Between $\mathbf{bl}[j-1]$ and $\mathbf{bu}[j-1]$
3	Basic	Between $\mathbf{bl}[j-1]$ and $\mathbf{bu}[j-1]$

If **ninf** = 0, basic and superbasic variables may be outside their bounds by as much as the value of the optional argument **Feasibility Tolerance**. Note that unless the optional argument **Scale Option** = 0 is specified, the optional argument **Feasibility Tolerance** applies to the variables of the scaled problem. In this case, the variables of the original problem may be as much as 0.1 outside their bounds, but this is unlikely unless the problem is very badly scaled.

Very occasionally some nonbasic variables may be outside their bounds by as much as the optional argument **Feasibility Tolerance**, and there may be some nonbasic variables for which $\mathbf{x}[j-1]$ lies strictly between its bounds.

If **ninf** > 0, some basic and superbasic variables may be outside their bounds by an arbitrary amount (bounded by **sinf** if **Scale Option** = 0).

21: $\mathbf{x}[\mathbf{n} + \mathbf{m}]$ – double *Input/Output*

On entry: the initial values of the variables x, and, if **start** = Nag_Warm, the slacks s, i.e., (x, s). (See the description for argument **hs**.)

On exit: the final values of the variables and slacks (x, s).

22: **pi**[**m**] – double *Output*

On exit: contains the dual variables π (a set of Lagrange multipliers (shadow prices) for the general constraints).

23: **rc**[**n** + **m**] – double *Output*

On exit: contains the reduced costs, $g - \begin{pmatrix} A & -I \end{pmatrix}^{\mathrm{T}}\pi$. The vector g is the gradient of the objective if **x** is feasible, otherwise it is the gradient of the Phase 1 objective. In the former case, $g(i) = 0$, for $i = \mathbf{n} + 1 : \mathbf{m}$, hence $\mathbf{rc}[\mathbf{n} + 1 : \mathbf{m} - 1] = \pi$.

24: **ns** – Integer * *Input/Output*

On entry: n_S, the number of superbasics. For QP problems, **ns** need not be specified if **start** = Nag_Cold, but must retain its value from a previous call when **start** = Nag_Warm. For FP and LP problems, **ns** need not be initialized.

On exit: the final number of superbasics. This will be zero for FP and LP problems.

25: **ninf** – Integer * *Output*

On exit: the number of infeasibilities.

26: **sinf** – double * *Output*

On exit: the sum of the scaled infeasibilities. This will be zero if **ninf** = 0, and is most meaningful when **Scale Option** = 0.

27: **obj** – double * *Output*

On exit: the value of the objective function.

If **ninf** = 0, **obj** includes the quadratic objective term $\frac{1}{2}x^{\mathrm{T}}Hx$ (if any).

If **ninf** > 0, **obj** is just the linear objective term $c^{\mathrm{T}}x$ (if any).

For FP problems, **obj** is set to zero.

Note that **obj** does not include contributions from the constant term **objadd** or the objective row, if any.

28: **state** – Nag_E04State * *Communication Structure*

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

29: **comm** – Nag_Comm *

The NAG communication argument (see Section 3.2.1.1 in the Essential Introduction).

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

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

nag_opt_sparse_convex_qp_solve (e04nqc) returns with **fail.code** = NE_NOERROR if the reduced gradient (`rgNorm`; see Section 9.1) is negligible, the Lagrange multipliers (`Lagr Mult`; see Section 9.1) are optimal, x satisfies the constraints to the accuracy requested by the value of the optional argument **Feasibility Tolerance** and the reduced Hessian factor R (see Section 11.2) is nonsingular.

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_ALLOC_INSUFFICIENT

Internal memory allocation was insufficient. Please contact NAG.

NE_ARRAY_INPUT

On entry, **loca**[0] = ⟨*value*⟩, **loca**[⟨*value*⟩] = ⟨*value*⟩, **ne** = ⟨*value*⟩.
Constraint: **loca**[0] = 1 or **loca**[⟨*value*⟩] = **ne** + 1.

On entry, row index ⟨*value*⟩ in **inda**[⟨*value*⟩] is outside the range 1 to **m** = ⟨*value*⟩.

NE_BAD_PARAM

Basis file dimensions do not match this problem.

On entry, argument ⟨*value*⟩ had an illegal value.

NE_BASIS_FAILURE

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

NE_BASIS_ILL_COND

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

Numerical error in trying to satisfy the general constraints. The basis is very ill-conditioned.

An LU factorization of the basis has just been obtained and used to recompute the basic variables x_B, given the present values of the superbasic and nonbasic variables. However, a row check has revealed that the resulting solution does not satisfy the current constraints $Ax - s = 0$ sufficiently well.

This probably means that the current basis is very ill-conditioned. Request the **Scale Option** *if there are any linear constraints and variables.*

For certain highly structured basis matrices (notably those with band structure), a systematic growth may occur in the factor U. Consult the description of `Umax`*,* `Umin` *and* `Growth` *in Section 13, and set the optional argument* **LU Factor Tolerance** *to 2.0 (or possibly even smaller, but not less than 1.0).*

NE_BASIS_SINGULAR

The basis is singular after several attempts to factorize it (and add slacks where necessary).

Either the problem is badly scaled or the value of the optional argument **LU Factor Tolerance** *is too large.*

NE_E04NPC_NOT_INIT

The initialization function nag_opt_sparse_convex_qp_init (e04npc) has not been called.

NE_HESS_INDEF

Error in **qphx**: the QP Hessian is indefinite.

An indefinite matrix was detected during the computation of the reduced Hessian factor R (see Section 11.2). This may be caused by H being indefinite. Check also that ***qphx*** *has been coded correctly and that all relevant elements of Hx have been assigned their correct values. If* ***qphx*** *is*

coded correctly and H is positive semidefinite, the failure may be caused by ill conditioning. Try reducing the values of the optional arguments **LU Factor Tolerance** *and* **LU Update Tolerance**. *If there are very large values in H, check the scaling of the variables and constraints.*

NE_HESS_TOO_BIG

The value of the optional argument **Superbasics Limit** is too small.

The current set of basic and superbasic variables have been optimized as much as possible and a pricing operation is necessary to continue, but there are already **Superbasics Limit** *superbasics (and no room for any more).*

In general, raise the **Superbasics Limit** s *by a reasonable amount, bearing in mind the storage needed for reduced Hessian (see Section 11.2). (The* **Reduced Hessian Dimension** h *will also increase to s unless specified otherwise, and the associated storage will be about $\frac{1}{2}s^2$ words.) In some cases you may have to set $h < s$ to conserve storage, but beware that the rate of convergence will probably fall off severely.*

NE_INT

On entry, $\mathbf{m} = \langle value \rangle$.
Constraint: $\mathbf{m} \geq 1$.

On entry, $\mathbf{n} = \langle value \rangle$.
Constraint: $\mathbf{n} \geq 1$.

NE_INT_2

On entry, $\mathbf{iobj} = \langle value \rangle$ and $\mathbf{m} = \langle value \rangle$.
Constraint: $0 \leq \mathbf{iobj} \leq \mathbf{m}$.

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

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

On entry, **ne** is not equal to the number of nonzeros in **acol**. $\mathbf{ne} = \langle value \rangle$, nonzeros in $\mathbf{acol} = \langle value \rangle$.

NE_INT_3

On entry, $\mathbf{n} = \langle value \rangle$, $\mathbf{m} = \langle value \rangle$ and $\mathbf{ne} = \langle value \rangle$.
Constraint: $1 \leq \mathbf{ne} \leq \mathbf{n} \times \mathbf{m}$.

On entry, $\mathbf{n} = \langle value \rangle$, $\mathbf{m} = \langle value \rangle$ and $\mathbf{nname} = \langle value \rangle$.
Constraint: $\mathbf{nname} = 1$ or $\mathbf{n} + \mathbf{m}$.

On entry, $\mathbf{ne} = \langle value \rangle$, $\mathbf{n} = \langle value \rangle$ and $\mathbf{m} = \langle value \rangle$.
Constraint: $1 \leq \mathbf{ne} \leq \mathbf{n} \times \mathbf{m}$.

On entry, $\mathbf{nname} = \langle value \rangle$, $\mathbf{n} = \langle value \rangle$ and $\mathbf{m} = \langle value \rangle$.
Constraint: $\mathbf{nname} = 1$ or $\mathbf{n} + \mathbf{m}$.

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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

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

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

NE_NOT_REQUIRED_ACC

The requested accuracy could not be achieved.

NE_REAL_2

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

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

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

NE_UNBOUNDED

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

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

The problem is unbounded (or badly scaled). For a minimization problem, the objective function is not bounded below in the feasible region.

For linear problems, unboundedness is detected by the simplex method when a nonbasic variable can be increased or decreased by an arbitrary amount without causing a basic variable to violate a bound. Consider adding an upper or lower bound to the variable. Also, examine the constraints that have nonzeros in the associated column, to see if they have been formulated as intended.

Very rarely, the scaling of the problem could be so poor that numerical error will give an erroneous indication of unboundedness. Consider using the optional argument **Scale Option**.

NW_NOT_FEASIBLE

The linear constraints appear to be infeasible.

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

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

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

The problem is infeasible. The general constraints cannot all be satisfied simultaneously to within the value of the optional argument **Feasibility Tolerance**.

Feasibility is measured with respect to the upper and lower bounds on the variables and slacks. The message tells us that among all the points satisfying the general constraints $Ax - s = 0$, there is apparently no point that satisfies the bounds on x and s. Violations as small as the **Feasibility Tolerance** *are ignored, but at least one component of x or s violates a bound by more than the tolerance.*

Note: *although the objective function is the sum of infeasibilities (when* **ninf** > 0*), this sum will not necessarily have been minimized when* **Elastic Mode** $= 1$.

If **Elastic Mode** $\neq 0$*, nag_opt_sparse_convex_qp_solve (e04nqc) will optimize the QP objective and the sum of infeasibilities, suitably weighted using the optional argument* **Elastic Mode**. *The function will tend to determine a 'good' infeasible point if the elastic weight is sufficiently large.*

NW_SOLN_NOT_UNIQUE

Weak solution found – the solution is not unique.

NW_TOO_MANY_ITER

Iteration limit reached.

Major iteration limit reached.

Too many iterations. The value of the optional argument **Iterations Limit** *is too small.*

The Iterations limit was exceeded before the required solution could be found. Check the iteration log to be sure that progress was being made. If so, restart the run using a Basis file that was saved at the end of the run.

7 Accuracy

nag_opt_sparse_convex_qp_solve (e04nqc) 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

nag_opt_sparse_convex_qp_solve (e04nqc) is not threaded by NAG in any implementation.

nag_opt_sparse_convex_qp_solve (e04nqc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the X06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

This section contains a description of the printed output.

9.1 Description of the Printed Output

If **Print Level** > 0, one line of information is output to the **Print File** every kth iteration, where k is the specified **Print Frequency**. A heading is printed before the first such line following a basis factorization. The heading contains the items described below. In this description, a pricing operation is defined to be the process by which one or more nonbasic variables are selected to become superbasic (in addition to those already in the superbasic set). The variable selected will be denoted by `jq`. If the problem is purely linear, variable `jq` will usually become basic immediately (unless it should happen to reach its opposite bound and return to the nonbasic set).

If optional argument **Partial Price** is in effect, variable `jq` is selected from $A_{\mathtt{pp}}$ or $I_{\mathtt{pp}}$, the `pp`th segments of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$.

Label	**Description**
`Itn`	is the iteration count.
`pp`	is the partial-price indicator. The variable selected by the last pricing operation came from the `pp`th partition of A and $-I$. Note that `pp` is reset to zero whenever the basis is refactorized.
`dj`	is the value of the reduced gradient (or reduced cost) for the variable selected by the pricing operation at the start of the current iteration. Algebraically, `dj` is $d_j = g_j - \pi^{\mathrm{T}} a_j$, for $j =$ `jq`, where g_j is the gradient of the current objective function, π is the vector of dual variables, and a_j is the jth column of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$. Note that `dj` is the norm of the reduced-gradient vector at the start of the iteration, just after the pricing operation.
`+SBS`	is the variable `jq` selected by the pricing operation to be added to the superbasic set.

`-SBS`	is the variable chosen to leave the superbasic set. It has become basic if the entry under `-B` is nonzero, otherwise it becomes nonbasic.
`-BS`	is the variable removed from the basis to become nonbasic.
`Step`	is the value of the step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. If a variable is made superbasic during the current iteration (i.e., `+SBS` is positive), `Step` will be the step to the nearest bound. During the optimality phase, the step can be greater than unity only if the reduced Hessian is not positive definite.
`Pivot`	is the rth element of a vector y satisfying $By = a_q$ whenever a_q (the qth column of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$ replaces the rth column of the basis matrix B. Wherever possible, `Step` is chosen so as to avoid extremely small values of `Pivot` (since they may cause the basis to be nearly singular). In extreme cases, it may be necessary to increase the value of the optional argument **Pivot Tolerance** to exclude very small elements of y from consideration during the computation of `Step`.
`nInf`	is the number of violated constraints (infeasibilities) before the present iteration. This number will not increase unless iterations are in elastic mode.
`sInf`	is the sum of infeasibilities before the present iteration. It will usually decrease at each nonzero step, but if `nInf` decreases by 2 or more, `sInf` may occasionally increase. However, in elastic mode it will decrease monotonically.
`Objective`	is the value of the current objective function after the present iteration. Note, if **Elastic Mode** is 2, the heading is `Composite Obj`.
`L+U`	`L` is the number of nonzeros in the basis factor L. Immediately after a basis factorization $B = LU$, `L` contains `lenL` (see Section 13). Further nonzeros are added to `L` when various columns of B are later replaced. (Thus, `L` increases monotonically.) `U` is the number of nonzeros in the basis factor U. Immediately after a basis factorization $B = LU$, `U` contains `lenU` (see Section 13). As columns of B are replaced, the matrix U is maintained explicitly (in sparse form). The value of `U` may fluctuate up or down; in general, it will tend to increase.
`ncp`	is the number of compressions required to recover workspace in the data structure for U. This includes the number of compressions needed during the previous basis factorization. Normally, `ncp` should increase very slowly.

The following will be output if the problem is QP or if the superbasic set is non-empty.

Label	**Description**
`rgNorm`	is the largest reduced-gradient among the superbasic variables after the current iteration. During the optimality phase, this will be approximately zero after a unit step.
`nS`	is the current number of superbasic variables.
`condHz`	is a lower bound on the condition number of the reduced Hessian (see Section 11.2). The larger this number, the more difficult the problem. Attention should be given to the scaling of the variables and the constraints to guard against high values of `condHz`.

10 Example

This example minimizes the quadratic function $f(x) = c^{\mathrm{T}}x + \frac{1}{2}x^{\mathrm{T}}Hx$, where

$$c = (-200.0, -2000.0, -2000.0, -2000.0, -2000.0, 400.0, 400.0)^{\mathrm{T}}$$

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

subject to the bounds

$$\begin{aligned} 0 &\leq x_1 \leq 200 \\ 0 &\leq x_2 \leq 2500 \\ 400 &\leq x_3 \leq 800 \\ 100 &\leq x_4 \leq 700 \\ 0 &\leq x_5 \leq 1500 \\ 0 &\leq x_6 \\ 0 &\leq x_7 \end{aligned}$$

and to the linear constraints

$$\begin{array}{rcrcrcrcrcrcrcr} & & x_1 & + & x_2 & + & x_3 & + & x_4 & + & x_5 & + & x_6 & + & x_7 & = & 2000 \\ & & 0.15x_1 & + & 0.04x_2 & + & 0.02x_3 & + & 0.04x_4 & + & 0.02x_5 & + & 0.01x_6 & + & 0.03x_7 & \leq & 60 \\ & & 0.03x_1 & + & 0.05x_2 & + & 0.08x_3 & + & 0.02x_4 & + & 0.06x_5 & + & 0.01x_6 & + & 0.03x_7 & \leq & 100 \\ & & 0.02x_1 & + & 0.04x_2 & + & 0.01x_3 & + & 0.02x_4 & + & 0.02x_5 & & & & & \leq & 40 \\ & & 0.02x_1 & + & 0.03x_2 & & & & & + & 0.01x_5 & & & & & \leq & 30 \\ 1500 & \leq & 0.70x_1 & + & 0.75x_2 & + & 0.80x_3 & + & 0.75x_4 & + & 0.80x_5 & + & 0.97x_6 & & & & \\ 250 & \leq & 0.02x_1 & + & 0.06x_2 & + & 0.08x_3 & + & 0.12x_4 & + & 0.02x_5 & + & 0.01x_6 & + & 0.97x_7 & \leq & 300 \end{array}$$

The initial point, which is infeasible, is

$$x_0 = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)^{\mathrm{T}}.$$

The optimal solution (to five figures) is

$$x^* = (0.0, 349.40, 648.85, 172.85, 407.52, 271.36, 150.02)^{\mathrm{T}}.$$

One bound constraint and four linear constraints are active at the solution. Note that the Hessian matrix H is positive semidefinite.

10.1 Program Text

```
/* nag_opt_sparse_convex_qp_solve (e04nqc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 8, 2004.
 */

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

#ifdef __cplusplus
extern "C" {
#endif
static void NAG_CALL qphx(Integer ncolh, const double x[], double hx[],
                          Integer nstate, Nag_Comm *comm);
#ifdef __cplusplus
}
#endif

int main(void)
{
  /* Scalars */
  double       obj, objadd, sinf;
  Integer      exit_status, i, icol, iobj, j, jcol, lenc, m, n, ncolh, ne,
```

```
              ninf;
  Integer     nname, ns;

  /* Arrays */
  char        nag_enum_arg[40];
  char        prob[9];
  char        **names;
  double      *acol = 0, *bl = 0, *bu = 0, *c = 0, *pi = 0, *rc = 0, *x = 0;
  Integer     *helast = 0, *hs = 0, *inda = 0, *loca = 0;

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

  exit_status = 0;
  INIT_FAIL(fail);

  printf("nag_opt_sparse_convex_qp_solve (e04nqc) Example Program Results\n");
  fflush(stdout);

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

  /* Read ne, iobj, ncolh, start and nname from data file. */
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &n, &m);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &n, &m);
#endif
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT" %39s %"NAG_IFMT"%*[^\n] ",
        &ne, &iobj, &ncolh, nag_enum_arg, _countof(nag_enum_arg), &nname);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT" %39s %"NAG_IFMT"%*[^\n] ",
        &ne, &iobj, &ncolh, nag_enum_arg, &nname);
#endif
  /* nag_enum_name_to_value (x04nac).
   * Converts NAG enum member name to value
   */
  start = (Nag_Start) nag_enum_name_to_value(nag_enum_arg);
  if (n >= 1 && m >= 1)
    {
      /* Allocate memory */
      if (!(names = NAG_ALLOC(n+m, char *)) ||
          !(acol = NAG_ALLOC(ne, double)) ||
          !(bl = NAG_ALLOC(m+n, double)) ||
          !(bu = NAG_ALLOC(m+n, double)) ||
          !(c = NAG_ALLOC(1, double)) ||
          !(pi = NAG_ALLOC(m, double)) ||
          !(rc = NAG_ALLOC(n+m, double)) ||
          !(x = NAG_ALLOC(n+m, double)) ||
          !(helast = NAG_ALLOC(n+m, Integer)) ||
          !(hs = NAG_ALLOC(n+m, Integer)) ||
          !(inda = NAG_ALLOC(ne, Integer)) ||
          !(loca = NAG_ALLOC(n+1, Integer)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
    }
  else
    {
      printf("%s", "Either m or n invalid\n");
```

```
      exit_status = 1;
      return exit_status;
    }

  /* Read names from data file. */
  for (i = 1; i <= nname; ++i)
    {
      names[i-1] = NAG_ALLOC(9, char);
#ifdef _WIN32
      scanf_s(" ' %8s '", names[i-1], 9);
#else
      scanf(" ' %8s '", names[i-1]);
#endif
    }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read the matrix acol from data file. Set up LOCA. */
  jcol = 1;
  loca[jcol - 1] = 1;
  for (i = 1; i <= ne; ++i)
    {
      /* Element (inda[i-1], icol) is stored in acol[i-1]. */
#ifdef _WIN32
      scanf_s("%lf%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &acol[i - 1], &inda[i - 1],
              &icol);
#else
      scanf("%lf%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &acol[i - 1], &inda[i - 1],
              &icol);
#endif
      if (icol < jcol)
        {
          /* Elements not ordered by increasing column index. */
          printf("%s%5"NAG_IFMT"%s%5"NAG_IFMT"%s%s\n", "Element in column",
                 icol, " found after element in column", jcol, ". Problem",
                 " abandoned.");
        }
      else if (icol == jcol + 1)
        {
          /* Index in ACOL of the start of the ICOL-th column equals I. */
          loca[icol - 1] = i;
          jcol = icol;
        }
      else if (icol > jcol + 1)
        {
          /* Index in acol of the start of the icol-th column equals i, */
          /* but columns jcol+1,jcol+2,...,icol-1 are empty. Set the */
          /* corresponding elements of loca to i. */
          for (j = jcol + 1; j <= icol - 1; ++j)
            {
              loca[j - 1] = i;
            }
          loca[icol - 1] = i;
          jcol = icol;
        }
    }
  loca[n] = ne + 1;

  if (n > icol)
    {
      /* Columns n,n-1,...,icol+1 are empty. Set the corresponding */
      /* elements of loca accordingly. */
      for (i = n; i >= icol + 1; --i)
        {
          loca[i - 1] = loca[i];
        }
    }
```

```
  /* Read bl, bu, hs and x from data file. */
  for (i = 1; i <= n + m; ++i)
    {
#ifdef _WIN32
      scanf_s("%lf", &bl[i - 1]);
#else
      scanf("%lf", &bl[i - 1]);
#endif
    }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  for (i = 1; i <= n + m; ++i)
    {
#ifdef _WIN32
      scanf_s("%lf", &bu[i - 1]);
#else
      scanf("%lf", &bu[i - 1]);
#endif
    }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  if (start == Nag_Cold)
    {
      for (i = 1; i <= n; ++i)
        {
#ifdef _WIN32
          scanf_s("%"NAG_IFMT"", &hs[i - 1]);
#else
          scanf("%"NAG_IFMT"", &hs[i - 1]);
#endif
        }
#ifdef _WIN32
      scanf_s("%*[^\n] ");
#else
      scanf("%*[^\n] ");
#endif
    }
  else if (start == Nag_Warm)
    {
      for (i = 1; i <= n + m; ++i)
        {
#ifdef _WIN32
          scanf_s("%"NAG_IFMT"", &hs[i - 1]);
#else
          scanf("%"NAG_IFMT"", &hs[i - 1]);
#endif
        }
#ifdef _WIN32
      scanf_s("%*[^\n] ");
#else
      scanf("%*[^\n] ");
#endif
    }
  for (i = 1; i <= n; ++i)
    {
#ifdef _WIN32
      scanf_s("%lf", &x[i - 1]);
#else
      scanf("%lf", &x[i - 1]);
#endif
    }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
```

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

  /* nag_opt_sparse_convex_qp_init (e04npc).
   * Initialization function for
   * nag_opt_sparse_convex_qp_solve (e04nqc)
   */
  nag_opt_sparse_convex_qp_init(&state, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Initialisation of "
             "nag_opt_sparse_convex_qp_solve (e04nqc) failed.\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }
  /* By default nag_opt_sparse_convex_qp_solve (e04nqc) does not print
   * monitoring information. Call nag_open_file (x04acc) to set the print file
   * fileid */
  /* nag_open_file (x04acc).
   * Open unit number for reading, writing or appending, and
   * associate unit with named file
   */
  nag_open_file("", 2, &fileid, &fail);
  if (fail.code != NE_NOERROR)
    {
      exit_status = 2;
      goto END;
    }

  /* nag_opt_sparse_convex_qp_option_set_integer (e04ntc).
   * Set a single option for nag_opt_sparse_convex_qp_solve (e04nqc)
   * from an integer argument
   */
  nag_opt_sparse_convex_qp_option_set_integer("Print file", fileid, &state,
                                              &fail);
  if (fail.code != NE_NOERROR)
    {
      exit_status = 1;
      goto END;
    }

  /* We have no explicit objective vector so set lenc = 0; the
   * objective vector is stored in row iobj of acol.
   */
  lenc = 0;
  objadd = 0.;
#ifdef _WIN32
  strcpy_s(prob, _countof(prob), "        ");
#else
  strcpy(prob, "        ");
#endif

  /* Do not allow any elastic variables (i.e. they cannot be */
  /* infeasible). If we'd set optional argument "Elastic mode" to 0, */
  /* we wouldn't need to set the individual elements of array helast. */
  for (i = 1; i <= n + m; ++i)
    {
      helast[i - 1] = 0;
    }

  /* Illustrate how to pass information to the user-supplied
     function qphx via the comm structure */
  comm.p = 0;

  /* Solve the QP problem. */
  /* nag_opt_sparse_convex_qp_solve (e04nqc).
   * LP or QP problem (suitable for sparse problems)
   */
  nag_opt_sparse_convex_qp_solve(start, qphx, m, n, ne, nname, lenc, ncolh,
```

```
                                iobj, objadd, prob, acol, inda, loca, bl, bu,
                                c,
                                (const char **) names, helast, hs, x, pi, rc,
  &ns, &ninf, &sinf, &obj, &state, &comm,
  &fail);

  if (fail.code != NE_NOERROR)
  {
    printf("nag_opt_sparse_convex_qp_solve (e04nqc) failed.\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }
  if (fail.code != NE_NOERROR)
  {
    exit_status = 2;
    goto END;
  }

  printf("Final objective value = %12.3e\n", obj);
  printf("Optimal X = ");

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

 END:

  for (i = 0; i < n+m; i++)
  {
    NAG_FREE(names[i]);
  }
  NAG_FREE(names);
  NAG_FREE(acol);
  NAG_FREE(bl);
  NAG_FREE(bu);
  NAG_FREE(c);
  NAG_FREE(pi);
  NAG_FREE(rc);
  NAG_FREE(x);
  NAG_FREE(helast);
  NAG_FREE(hs);
  NAG_FREE(inda);
  NAG_FREE(loca);
  return exit_status;
}

static void NAG_CALL qphx(Integer ncolh, const double x[], double hx[],
                          Integer nstate, Nag_Comm *comm)
{
  /* Routine to compute H*x. (In this version of qphx, the Hessian
   * matrix H is not referenced explicitly.)
   */

  /* Parameter adjustments */
#define HX(I) hx[(I) -1]
#define X(I)  x[(I) -1]

  /* Check whether information came from the main program
     via the comm structure. Even if it was, we ignore it
     in this example. */
  if (comm->p)
    printf("Pointer %p was passed to qphx via the comm struct\n", comm->p);

  /* Function Body */
  HX(1) = X(1) * 2;
  HX(2) = X(2) * 2;
  HX(3) = (X(3) + X(4)) * 2;
  HX(4) = HX(3);
  HX(5) = X(5) * 2;
```

```
  HX(6) = (X(6) + X(7)) * 2;
  HX(7) = HX(6);
  return;
} /* qphx */
```

10.2 Program Data

```
nag_opt_sparse_convex_qp_solve (e04nqc) Example Program Data
 7  8                      : Values of n and m
48  8  7  Nag_Cold  15     : Values of nnz, iobj, ncolh, start and nname

'...X1...'  '...X2...'  '...X3...'  '...X4...'  '...X5...'
'...X6...'  '...X7...'  '..ROW1..'  '..ROW2..'  '..ROW3..'
'..ROW4..'  '..ROW5..'  '..ROW6..'  '..ROW7..'  '..COST..' : End of array NAMES

    0.02   7   1   : Sparse matrix A, ordered by increasing column index;
    0.02   5   1   : each row contains ACOL(i), INDA(i), ICOL (= column index)
    0.03   3   1   : The row indices may be in any order. In this example
    1.00   1   1   : row 8 defines the linear objective term transpose(C)*X.
    0.70   6   1
    0.02   4   1
    0.15   2   1
 -200.00   8   1
    0.06   7   2
    0.75   6   2
    0.03   5   2
    0.04   4   2
    0.05   3   2
    0.04   2   2
    1.00   1   2
-2000.00   8   2
    0.02   2   3
    1.00   1   3
    0.01   4   3
    0.08   3   3
    0.08   7   3
    0.80   6   3
-2000.00   8   3
    1.00   1   4
    0.12   7   4
    0.02   3   4
    0.02   4   4
    0.75   6   4
    0.04   2   4
-2000.00   8   4
    0.01   5   5
    0.80   6   5
    0.02   7   5
    1.00   1   5
    0.02   2   5
    0.06   3   5
    0.02   4   5
-2000.00   8   5
    1.00   1   6
    0.01   2   6
    0.01   3   6
    0.97   6   6
    0.01   7   6
  400.00   8   6
    0.97   7   7
    0.03   2   7
    1.00   1   7
  400.00   8   7        : End of matrix A

 0.0       0.0       4.0E+02   1.0E+02   0.0       0.0
 0.0       2.0E+03  -1.0E+25  -1.0E+25  -1.0E+25  -1.0E+25
 1.5E+03   2.5E+02  -1.0E+25             : End of lower bounds array BL

 2.0E+02   2.5E+03   8.0E+02   7.0E+02  1.5E+03  1.0E+25
```

```
 1.0E+25   2.0E+03   6.0E+01   1.0E+02  4.0E+01  3.0E+01
 1.0E+25   3.0E+02   1.0E+25             : End of upper bounds array BU

 0    0    0    0    0    0    0         : Initial array HS
 0.0  0.0  0.0  0.0  0.0  0.0  0.0       : Initial vector X
```

10.3 Program Results

```
nag_opt_sparse_convex_qp_solve (e04nqc) Example Program Results

 Parameters
 ==========

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

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

 LP/QP Parameters
 ----------------
 Minimize...............                QPsolver Cholesky......                Cold start.............
 Scale tolerance........     0.900      Feasibility tolerance..  1.00E-06      Iteration limit........     10000
 Scale option...........         2      Optimality tolerance...  1.00E-06      Print level............         1
 Crash tolerance........     0.100      Pivot tolerance........  2.05E-11      Partial price..........         1
 Crash option...........         3      Elastic weight.........  1.00E+00      Prtl price section ( A)         7
 Elastic mode...........         1      Elastic objective......         1      Prtl price section (-I)         8

 QP objective
 ------------
 Objective variables....         7      Hessian columns........         7      Superbasics limit......         7
 Nonlin Objective vars..         7      Unbounded step size....  1.00E+20
 Linear Objective vars..         0

 Miscellaneous
 -------------
 LU factor tolerance....      3.99      LU singularity tol.....  2.05E-11      Timing level...........         0
 LU update tolerance....      3.99      LU swap tolerance......  1.03E-04      Debug level............         0
 LU partial  pivoting...                eps (machine precision)  1.11E-16      System information.....        No



 Matrix statistics
 -----------------
               Total      Normal        Free       Fixed     Bounded
 Rows              8           5           1           1           1
 Columns           7           2           0           0           5

 No. of matrix elements                   48     Density      85.714
 Biggest                           1.0000E+00  (excluding fixed columns,
 Smallest                          1.0000E-02   free rows, and RHS)

 No. of objective coefficients             7
 Biggest                           2.0000E+03  (excluding fixed columns)
 Smallest                          2.0000E+02

 Nonlinear constraints       0     Linear constraints       8
 Nonlinear variables         7     Linear variables         0
 Jacobian  variables         0     Objective variables      7
 Total constraints           8     Total variables          7


 Itn      1: Feasible linear constraints
```

```
 E04NQT EXIT   0 -- finished successfully
 E04NQT INFO   1 -- optimality conditions satisfied

 Problem name
 No. of iterations                  9   Objective value     -1.8477846771E+06
 No. of Hessian products           16   Objective row       -2.9886903537E+06
                                        Quadratic objective  1.1409056766E+06
 No. of superbasics                 2   No. of basic nonlinears             4
 No. of degenerate steps            0   Percentage                       0.00
 Max x       (scaled)       3 2.4E-01   Max pi      (scaled)        6 4.7E+07
 Max x                      3 6.5E+02   Max pi                      7 1.5E+04
 Max Prim inf(scaled)       0 0.0E+00   Max Dual inf(scaled)        6 1.1E-08
 Max Primal infeas          0 0.0E+00   Max Dual infeas             9 6.4E-12

 Name                                   Objective Value     -1.8477846771E+06

 Status         Optimal Soln            Iteration      9    Superbasics     2

 Section 1 - Rows

  Number  ...Row.. State  ...Activity...  Slack Activity  ..Lower Limit.  ..Upper Limit.  .Dual Activity    ..i

       8  ..ROW1..    EQ      2000.00000           .          2000.00000      2000.00000    -12900.76766      1
       9  ..ROW2..    BS        49.23160      -10.76840           None          60.00000         0.00000      2
      10  ..ROW3..    UL       100.00000           .              None         100.00000     -2324.86620      3
      11  ..ROW4..    BS        32.07187       -7.92813           None          40.00000           .          4
      12  ..ROW5..    BS        14.55719      -15.44281           None          30.00000           .          5
      13  ..ROW6..    LL      1500.00000           .          1500.00000           None      14454.60290      6
      14  ..ROW7..    LL       250.00000           .           250.00000       300.00000     14580.95432      7
      15  ..COST..    BS  -2988690.35370  -2988690.35370           None            None         -1.0          8

 Section 2 - Columns

  Number  .Column. State  ...Activity...  .Obj Gradient.  ..Lower Limit.  ..Upper Limit.  Reduced Gradnt    m+j

       1  ...X1...    LL           .          -200.00000          .           200.00000      2360.67253      9
       2  ...X2...    BS       349.39923      -1301.20153          .          2500.00000        -0.00000     10
       3  ...X3...   SBS       648.85342       -356.59829       400.00000       800.00000         0.00000     11
       4  ...X4...   SBS       172.84743       -356.59829       100.00000       700.00000        -0.00000     12
       5  ...X5...    BS       407.52089      -1184.95822          .          1500.00000        -0.00000     13
       6  ...X6...    BS       271.35624       1242.75804          .               None         -0.00000     14
       7  ...X7...    BS       150.02278       1242.75804          .               None          0.00000     15
Final objective value =  -1.848e+06
Optimal X =      0.00    349.40    648.85    172.85    407.52    271.36    150.02
```

Note: *the remainder of this document is intended for more advanced users. Section 11 contains a detailed description of the algorithm which may be needed in order to understand Sections 12 and 13. Section 12 describes the optional arguments which may be set by calls to nag_opt_sparse_convex_qp_option_set_file (e04nrc), nag_opt_sparse_convex_qp_option_set_string (e04nsc), nag_opt_sparse_convex_qp_option_set_integer (e04ntc) and/or nag_opt_sparse_convex_qp_option_set_double (e04nuc). Section 13 describes the quantities which can be requested to monitor the course of the computation.*

11 Algorithmic Details

This section contains a detailed description of the method used by nag_opt_sparse_convex_qp_solve (e04nqc).

11.1 Overview

nag_opt_sparse_convex_qp_solve (e04nqc) is based on an inertia-controlling method that maintains a Cholesky factorization of the reduced Hessian (see below). The method is similar to that of Gill and Murray (1978), and is described in detail by Gill *et al.* (1991). Here we briefly summarise the main features of the method. Where possible, explicit reference is made to the names of variables that are arguments of the function or appear in the printed output.

The method used has two distinct 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 computations in both phases are performed by the same functions. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities (the printed quantity `sInf`; see Section 9.1) to the quadratic objective function (the printed quantity `Objective`; see Section 9.1).

In general, an iterative process is required to solve a quadratic program. Given an iterate (x, s) in both the original variables x and the slack variables s, a new iterate $(\bar{x}, \bar{s})$ is defined by

$$\begin{pmatrix} \bar{x} \\ \bar{s} \end{pmatrix} = \begin{pmatrix} x \\ s \end{pmatrix} + \alpha p, \tag{3}$$

where the *step length* α is a non-negative scalar (the printed quantity `Step`; see Section 13), and p is called the *search direction*. (For simplicity, we shall consider a typical iteration and avoid reference to the index of the iteration.) Once an iterate is feasible (i.e., satisfies the constraints), all subsequent iterates remain feasible.

11.2 Definition of the Working Set and Search Direction

At each iterate (x, s), a *working set* of constraints is defined to be a linearly independent subset of the constraints that are satisfied 'exactly' (to within the value of the optional argument **Feasibility Tolerance**). The working set is the current prediction of the constraints that hold with equality at a solution of the LP or QP problem. Let m_W denote the number of constraints in the working set (including bounds), and let W denote the associated m_W by $(n + m)$ *working set matrix* consisting of the m_W gradients of the working set constraints.

The search direction is defined so that constraints in the working set remain *unaltered* for any value of the step length. It follows that p must satisfy the identity

$$Wp = 0. \tag{4}$$

This characterisation allows p to be computed using any n by n_Z full-rank matrix Z that spans the null space of W. (Thus, $n_Z = n - m_W$ and $WZ = 0$.) The null space matrix Z is defined from a sparse LU factorization of part of W (see (7) and (8)). The direction p will satisfy (4) if

$$p = Zp_Z, \tag{5}$$

where p_Z is any n_Z-vector.

The working set contains the constraints $Ax - s = 0$ and a subset of the upper and lower bounds on the variables (x, s). Since the gradient of a bound constraint $x_j \geq l_j$ or $x_j \leq u_j$ is a vector of all zeros except for ± 1 in position j, it follows that the working set matrix contains the rows of $\begin{pmatrix} A & -I \end{pmatrix}$ and the unit rows associated with the upper and lower bounds in the working set.

The working set matrix W can be represented in terms of a certain column partition of the matrix $\begin{pmatrix} A & -I \end{pmatrix}$ by (conceptually) partitioning the constraints $Ax - s = 0$ so that

$$Bx_B + Sx_S + Nx_N = 0, \tag{6}$$

where B is a square nonsingular basis and x_B, x_S and x_N are the basic, superbasic and nonbasic variables respectively. The nonbasic variables are equal to their upper or lower bounds at (x, s), and the superbasic variables are independent variables that are chosen to improve the value of the current objective function. The number of superbasic variables is n_S (the printed quantity `nS`; see Section 9.1). Given values of x_N and x_S, the basic variables x_B are adjusted so that (x, s) satisfies (6).

If P is a permutation matrix such that $\begin{pmatrix} A & -I \end{pmatrix} P = \begin{pmatrix} B & S & N \end{pmatrix}$, then W satisfies

$$WP = \begin{pmatrix} B & S & N \\ 0 & 0 & I_N \end{pmatrix}, \tag{7}$$

where I_N is the identity matrix with the same number of columns as N.

The null space matrix Z is defined from a sparse LU factorization of part of W. In particular, Z is maintained in 'reduced gradient' form, using the LUSOL package (see Gill *et al.* (1991)) to maintain

sparse LU factors of the basis matrix B as the BSN partition changes. Given the permutation P, the null space basis is given by

$$Z = P\begin{pmatrix} -B^{-1}S \\ I \\ 0 \end{pmatrix}. \tag{8}$$

This matrix is used only as an operator, i.e., it is never computed explicitly. Products of the form Zv and $Z^{\mathrm{T}}g$ are obtained by solving with B or B^{T}. This choice of Z implies that n_Z, the number of 'degrees of freedom' at (x, s), is the same as n_S, the number of superbasic variables.

Let g_Z and H_Z denote the *reduced gradient* and *reduced Hessian* of the objective function:

$$g_Z = Z^{\mathrm{T}}g \quad \text{and} \quad H_Z = Z^{\mathrm{T}}HZ, \tag{9}$$

where g is the objective gradient at (x, s). Roughly speaking, g_Z and H_Z describe the first and second derivatives of an n_S-dimensional *unconstrained* problem for the calculation of p_Z. (The condition estimator of H_Z is the quantity `condHz` in the monitoring file output; see Section 9.1.)

At each iteration, an upper triangular factor R is available such that $H_Z = R^{\mathrm{T}}R$. Normally, R is computed from $R^{\mathrm{T}}R = Z^{\mathrm{T}}HZ$ at the start of the optimality phase and then updated as the QP working set changes. For efficiency, the dimension of R should not be excessive (say, $n_S \leq 1000$). This is guaranteed if the number of nonlinear variables is 'moderate'.

If the QP problem contains linear variables, H is positive semidefinite and R may be singular with at least one zero diagonal element. However, an inertia-controlling strategy is used to ensure that only the last diagonal element of R can be zero. (See Gill *et al.* (1991) for a discussion of a similar strategy for indefinite quadratic programming.)

If the initial R is singular, enough variables are fixed at their current value to give a nonsingular R. This is equivalent to including temporary bound constraints in the working set. Thereafter, R can become singular only when a constraint is deleted from the working set (in which case no further constraints are deleted until R becomes nonsingular).

11.3 Main Iteration

If the reduced gradient is zero, (x, s) is a constrained stationary point on the working set. During the feasibility phase, the reduced gradient will usually be zero only at a vertex (although it may be zero elsewhere in the presence of constraint dependencies). During the optimality phase, a zero reduced gradient implies that x minimizes the quadratic objective function when the constraints in the working set are treated as equalities. At a constrained stationary point, Lagrange multipliers λ are defined from the equations

$$W^{\mathrm{T}}\lambda = g(x). \tag{10}$$

A Lagrange multiplier, λ_j, corresponding to an inequality constraint in the working set is said to be *optimal* if $\lambda_j \leq \sigma$ when the associated constraint is at its *upper bound*, or if $\lambda_j \geq -\sigma$ when the associated constraint is at its *lower bound*, where σ depends on the value of the optional argument **Optimality Tolerance**. If a multiplier is nonoptimal, the objective function (either the true objective or the sum of infeasibilities) can be reduced by continuing the minimization with the corresponding constraint excluded from the working set. (This step is sometimes referred to as 'deleting' a constraint from the working set.) If optimal multipliers occur during the feasibility phase but the sum of infeasibilities is nonzero, there is no feasible point and the function terminates immediately with **fail**.**code** = NE_NOT_REQUIRED_ACC.

The special form (7) of the working set allows the multiplier vector λ, the solution of (10), to be written in terms of the vector

$$d = \begin{pmatrix} g \\ 0 \end{pmatrix} - \begin{pmatrix} A^{\mathrm{T}} \\ -I \end{pmatrix}\pi = \begin{pmatrix} g - A^{\mathrm{T}}\pi \\ \pi \end{pmatrix}, \tag{11}$$

where π satisfies the equations $B^{\mathrm{T}}\pi = g_B$, and g_B denotes the basic elements of g. The elements of π are the Lagrange multipliers λ_j associated with the equality constraints $Ax - s = 0$. The vector d_N of nonbasic elements of d consists of the Lagrange multipliers λ_j associated with the upper and lower

bound constraints in the working set. The vector d_S of superbasic elements of d is the reduced gradient g_Z in (9). The vector d_B of basic elements of d is zero, by construction. (The Euclidean norm of d_S and the final values of d_S, g and π are the quantities `rgNorm`, `Reduced Gradnt`, `Obj Gradient` and `Dual Activity` in the monitoring file output; see Section 13.)

If the reduced gradient is not zero, Lagrange multipliers need not be computed and the search direction is given by $p = Zp_Z$ (see (8) and (12)). The step length is chosen to maintain feasibility with respect to the satisfied constraints.

There are two possible choices for p_Z, depending on whether or not H_Z is singular. If H_Z is nonsingular, R is nonsingular and p_Z in (5) is computed from the equations

$$R^{\mathrm{T}}Rp_Z = -g_Z, \tag{12}$$

where g_Z is the reduced gradient at x. In this case, $(x, s) + p$ is the minimizer of the objective function subject to the working set constraints being treated as equalities. If $(x, s) + p$ is feasible, α is defined to be unity. In this case, the reduced gradient at $(\bar{x}, \bar{s})$ will be zero, and Lagrange multipliers are computed at the next iteration. Otherwise, α is set to α_N, the step to the 'nearest' constraint along p. This constraint is then added to the working set at the next iteration.

If H_Z is singular, then R must also be singular, and an inertia-controlling strategy is used to ensure that only the last diagonal element of R is zero. (See Gill *et al.* (1991) for a discussion of a similar strategy for indefinite quadratic programming.) In this case, p_Z satisfies

$$p_Z^{\mathrm{T}}H_Zp_Z = 0 \quad \text{and} \quad g_Z^{\mathrm{T}}p_Z \leq 0, \tag{13}$$

which allows the objective function to be reduced by any step of the form $(x, s) + \alpha p$, where $\alpha > 0$. The vector $p = Zp_Z$ is a direction of unbounded descent for the QP problem in the sense that the QP objective is linear and decreases without bound along p. If no finite step of the form $(x, s) + \alpha p$ (where $\alpha > 0$) reaches a constraint not in the working set, the QP problem is unbounded and the function terminates immediately with **fail.code** = NE_UNBOUNDED. Otherwise, α is defined as the maximum feasible step along p and a constraint active at $(x, s) + \alpha p$ is added to the working set for the next iteration.

nag_opt_sparse_convex_qp_solve (e04nqc) makes explicit allowance for infeasible constraints. Infeasible linear constraints are detected first by solving a problem of the form

$$\underset{x,v,w}{\text{minimize}} e^{\mathrm{T}}(v + w) \quad \text{subject to } l \leq \begin{pmatrix} x \\ Gx - v + w \end{pmatrix} \leq u, \quad v \geq 0, \quad w \geq 0, \tag{14}$$

where $e^{\mathrm{T}} = (1, 1, \ldots, 1)$. This is equivalent to minimizing the sum of the general linear constraint violations subject to the simple bounds. (In the linear programming literature, the approach is often called *elastic programming*.)

11.4 Miscellaneous

If the basis matrix is not chosen carefully, the condition of the null space matrix Z in (8) could be arbitrarily high. To guard against this, the function implements a 'basis repair' feature in which the LUSOL package (see Gill *et al.* (1991)) is used to compute the rectangular factorization

$$\begin{pmatrix} B & S \end{pmatrix}^{\mathrm{T}} = LU, \tag{15}$$

returning just the permutation P that makes PLP^{T} unit lower triangular. The pivot tolerance is set to require $|PLP^{\mathrm{T}}|_{ij} \leq 2$, and the permutation is used to define P in (7). It can be shown that $\|Z\|$ is likely to be little more than unity. Hence, Z should be well-conditioned *regardless of the condition of* W. This feature is applied at the beginning of the optimality phase if a potential $B - S$ ordering is known.

The EXPAND procedure (see Gill *et al.* (1989)) is used to reduce the possibility of cycling at a point where the active constraints are nearly linearly dependent. Although there is no absolute guarantee that cycling will not occur, the probability of cycling is extremely small (see Hall and McKinnon (1996)). The main feature of EXPAND is that the feasibility tolerance is increased at the start of every iteration. This allows a positive step to be taken at every iteration, perhaps at the expense of violating the bounds on (x, s) by a small amount.

Suppose that the value of the optional argument **Feasibility Tolerance** is δ. Over a period of K iterations (where K is the value of the optional argument **Expand Frequency**), the feasibility tolerance actually used by the function (i.e., the *working* feasibility tolerance) increases from 0.5δ to δ (in steps of $0.5\delta/K$).

At certain stages the following 'resetting procedure' is used to remove small constraint infeasibilities. First, all nonbasic variables are moved exactly onto their bounds. A count is kept of the number of nontrivial adjustments made. If the count is nonzero, the basic variables are recomputed. Finally, the working feasibility tolerance is reinitialized to 0.5δ.

If a problem requires more than K iterations, the resetting procedure is invoked and a new cycle of iterations is started. (The decision to resume the feasibility phase or optimality phase is based on comparing any constraint infeasibilities with δ.)

The resetting procedure is also invoked when the function reaches an apparently optimal, infeasible or unbounded solution, unless this situation has already occurred twice. If any nontrivial adjustments are made, iterations are continued.

The EXPAND procedure not only allows a positive step to be taken at every iteration, but also provides a potential *choice* of constraints to be added to the working set. All constraints at a distance α (where $\alpha \leq \alpha_N$) 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. This strategy helps keep the basis matrix B well-conditioned.

12 Optional Arguments

Several optional arguments in nag_opt_sparse_convex_qp_solve (e04nqc) define choices in the problem specification or the algorithm logic. In order to reduce the number of formal arguments of nag_opt_sparse_convex_qp_solve (e04nqc) these optional arguments have associated *default values* that are appropriate for most problems. Therefore, you need only specify those optional arguments whose values are to be different from their default values.

The remainder of this section can be skipped if you wish to use the default values for all optional arguments.

The following is a list of the optional arguments available. A full description of each optional argument is provided in Section 12.1.

Backup Basis File
Check Frequency
Crash Option
Crash Tolerance
Defaults
Dump File
Elastic Mode
Elastic Objective
Elastic Weight
Expand Frequency
Factorization Frequency
Feasibility Tolerance
Feasible Point
Infinite Bound Size
Insert File
Iterations Limit
List
Load File
LU Complete Pivoting

LU Density Tolerance
LU Factor Tolerance
LU Partial Pivoting
LU Rook Pivoting
LU Singularity Tolerance
LU Update Tolerance
Maximize
Minimize
New Basis File
Nolist
Old Basis File
Optimality Tolerance
Partial Price
Pivot Tolerance
Print File
Print Frequency
Print Level
Punch File
QPSolver CG
QPSolver Cholesky
QPSolver QN
Reduced Hessian Dimension
Save Frequency
Scale Option
Scale Print
Scale Tolerance
Solution File
Solution No
Solution Yes
Summary File
Summary Frequency
Superbasics Limit
Suppress Parameters
System Information No
System Information Yes
Timing Level
Unbounded Step Size

Optional arguments may be specified by calling one, or any, of the functions nag_opt_sparse_convex_qp_option_set_file (e04nrc), nag_opt_sparse_convex_qp_option_set_string (e04nsc), nag_opt_sparse_convex_qp_option_set_integer (e04ntc) and nag_opt_sparse_convex_qp_option_set_double (e04nuc) before a call to nag_opt_sparse_convex_qp_solve (e04nqc), but after a call to nag_opt_sparse_convex_qp_init (e04npc).

nag_opt_sparse_convex_qp_option_set_file (e04nrc) reads options from an external options file, with `Begin` and `End` as the first and last lines respectively and each intermediate line defining a single optional argument. For example,

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

The call

```
e04nrc (ioptns, &state, &fail);
```

can then be used to read the file on descriptor `ioptns`. **fail.code** = NE_NOERROR on successful exit. nag_opt_sparse_convex_qp_option_set_file (e04nrc) should be consulted for a full description of this method of supplying optional arguments.

nag_opt_sparse_convex_qp_option_set_string (e04nsc), nag_opt_sparse_convex_qp_option_set_integer (e04ntc) or nag_opt_sparse_convex_qp_option_set_double (e04nuc) can be called to supply options directly, one call being necessary for each optional argument. nag_opt_sparse_convex_qp_option_set_string (e04nsc), nag_opt_sparse_convex_qp_option_set_integer (e04ntc) or nag_opt_sparse_convex_qp_option_set_double (e04nuc) should be consulted for a full description of this method of supplying optional arguments.

All optional arguments not specified by you are set to their default values. Optional arguments specified by you are unaltered by nag_opt_sparse_convex_qp_solve (e04nqc) (unless they define invalid values) and so remain in effect for subsequent calls unless altered by you.

12.1 Description of the Optional Arguments

For each option, we give a summary line, a description of the optional argument and details of constraints.

The summary line contains:

the keywords;

a parameter value, where the letters a, i and r denote options that take character, integer and real values respectively;

the default value is used whenever the condition $|i| \geq 100000000$ is satisfied and where the symbol ϵ is a generic notation for ***machine precision*** (see nag_machine_precision (X02AJC));

The variable $bigbnd$ holds the value of **Infinite Bound Size**.

Keywords and character values are case and white space insensitive.

Optional arguments used to specify files (e.g., optional arguments **Dump File** and **Print File**) have type Nag_FileID (see Section 3.2.1.1 in the Essential Introduction). This ID value must either be set to 0 (the default value) in which case there will be no output, or will be as returned by a call of nag_open_file (x04acc).

Check Frequency i Default $= 60$

Every ith iteration after the most recent basis factorization, a numerical test is made to see if the current solution (x, s) satisfies the linear constraints $Ax - s = 0$. If the largest element of the residual vector $r = Ax - s$ is judged to be too large, the current basis is refactorized and the basic variables recomputed to satisfy the constraints more accurately. If $i \leq 0$, the value $i = 99999999$ is used and effectively no checks are made.

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

Crash Option i Default $= 3$
Crash Tolerance r Default $= 0.1$

Note that these options do not apply when **start** = Nag_Warm (see Section 5).

If **start** = Nag_Cold, an internal Crash procedure is used to select an initial basis from various rows and columns of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$. The value of i determines which rows and columns of A are initially eligible for the basis, and how many times the Crash procedure is called. Columns of $-I$ are used to pad the basis where necessary.

i	**Meaning**
0	The initial basis contains only slack variables: $B = I$.
1	The Crash procedure is called once, looking for a triangular basis in all rows and columns of the matrix A.
2	The Crash procedure is called once, looking for a triangular basis in rows.
3	The Crash procedure is called twice, treating linear equalities and linear inequalities separately.

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

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

When $r > 0.0$, the basis obtained by the Crash procedure may not be strictly triangular, but it is likely to be nonsingular and almost triangular. The intention is to obtain a starting basis containing more columns of A and fewer (arbitrary) slacks. A feasible solution may be reached sooner on some problems.

For example, suppose the first m columns of A form the matrix shown under **LU Factor Tolerance**; i.e., a tridiagonal matrix with entries -1, 4, -1. To help the Crash procedure choose all m columns for the initial basis, we would specify a **Crash Tolerance** of r for some value of $r > 0.5$.

Defaults

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

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

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

Optional arguments **Dump File** and **Load File** are similar to optional arguments **Punch File** and **Insert File**, but they record solution information in a manner that is more direct and more easily modified. A full description of information recorded in optional arguments **Dump File** and **Load File** is given in Gill *et al.* (2005a).

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

If **Load File** > 0, the **Load File** containing basis information will be read. The file will usually have been output previously as a **Dump File**. The file will not be accessed if optional arguments **Old Basis File** or **Insert File** are specified.

Elastic Mode	i	Default $= 1$

This argument determines if (and when) elastic mode is to be started. Three elastic modes are available as follows:

i	**Meaning**
0	Elastic mode is never invoked. nag_opt_sparse_convex_qp_solve (e04nqc) will terminate as soon as infeasibility is detected. There may be other points with significantly smaller sums of infeasibilities.

1 Elastic mode is invoked only if the constraints are found to be infeasible (the default). If the constraints are infeasible, continue in elastic mode with the composite objective determined by the values of the optional arguments **Elastic Objective** and **Elastic Weight**.

2 The iterations start and remain in elastic mode. This option allows you to minimize the composite objective function directly without first performing Phase 1 iterations.

The success of this option will depend critically on your choice of **Elastic Weight**. If **Elastic Weight** is sufficiently large and the constraints are feasible, the minimizer of the composite objective and the solution of the original problem are identical. However, if the **Elastic Weight** is not sufficiently large, the minimizer of the composite function may be infeasible, even if a feasible point exists.

Elastic Objective i Default $= 1$

This determines the form of the composite objective $f(x) + \gamma \sum_j (v_j + w_j)$ in Phase 2 (γ). Three types of composite objectives are available.

i	**Meaning**
0	Include only the true objective $f(x)$ in the composite objective. This option sets $\gamma = 0$ in the composite objective and allows nag_opt_sparse_convex_qp_solve (e04nqc) to ignore the elastic bounds and find a solution that minimizes $f(x)$ subject to the non-elastic constraints. This option is useful if there are some 'soft' constraints that you would like to ignore if the constraints are infeasible.
1	Use a composite objective defined with γ determined by the value of **Elastic Weight**. This value is intended to be used in conjunction with **Elastic Mode** $= 2$.
2	Include only the elastic variables in the composite objective. The elastics are weighted by $\gamma = 1$. This choice minimizes the violations of the elastic variables at the expense of possibly increasing the true objective. This option can be used to find a point that minimizes the sum of the violations of a subset of constraints specified by the input array **helast**.

Elastic Weight r Default $= 1.0$

This defines the value of γ in the composite objective in Phase 2 (γ).

At each iteration of elastic mode, the composite objective is defined to be

$$\text{minimize } \sigma\ f(x) + \gamma\ (\text{sum of infeasibilities});$$

where $\sigma = 1$ for **Minimize**, $\sigma = -1$ for **Maximize**, and $f(x)$ is the quadratic objective.

Note that the effect of γ is *not* disabled once a feasible point is obtained.

Expand Frequency i Default $= 10000$

This option is part of an anti-cycling procedure (see Section 11.4) designed to allow progress even on highly degenerate problems.

The strategy is to force a positive step at every iteration, at the expense of violating the constraints by a small amount. Suppose that the value of the optional argument **Feasibility Tolerance** is δ. Over a period of i iterations, the feasibility tolerance actually used by nag_opt_sparse_convex_qp_solve (e04nqc) (i.e., the *working* feasibility tolerance) increases from 0.5δ to δ (in steps of $0.5\delta/i$).

Increasing the value of i helps reduce the number of slightly infeasible nonbasic variables (most of which are eliminated during the resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see the description of the optional argument **Pivot Tolerance**).

If $i \leq 0$, the value $i = 99999999$ is used and effectively no anti-cycling procedure is invoked.

Factorization Frequency i Default $= 100$(LP) or 50(QP)

If $i > 0$, at most i basis changes will occur between factorizations of the basis matrix.

For LP problems, the basis factors are usually updated at every iteration. Higher values of i may be more efficient on problems that are extremely sparse and well scaled.

For QP problems, fewer basis updates will occur as the solution is approached. The number of iterations between basis factorizations will therefore increase. During these iterations a test is made regularly according to the value of optional argument **Check Frequency** to ensure that the linear constraints $Ax - s = 0$ are satisfied. Occasionally, the basis will be refactorized before the limit of i updates is reached. If $i \leq 0$, the default value is used.

Feasibility Tolerance r Default $= \max\{10^{-6}, \sqrt{\epsilon}\}$

A *feasible problem* is one in which all variables satisfy their upper and lower bounds to within the absolute tolerance r. (This includes slack variables. Hence, the general constraints are also satisfied to within r.)

nag_opt_sparse_convex_qp_solve (e04nqc) attempts to find a feasible solution before optimizing the objective function. If the sum of infeasibilities cannot be reduced to zero, the problem is assumed to be *infeasible*. Let `sInf` be the corresponding sum of infeasibilities. If `sInf` is quite small, it may be appropriate to raise r by a factor of 10 or 100. Otherwise, some error in the data should be suspected.

Note that if `sInf` is not small and you have not asked nag_opt_sparse_convex_qp_solve (e04nqc) to minimize the violations of the elastic variables (i.e., you have not specified **Elastic Objective** $= 2$), there may be other points that have a *significantly smaller sum of infeasibilities*. nag_opt_sparse_convex_qp_solve (e04nqc) will not attempt to find the solution that minimizes the sum unless **Elastic Objective** $= 2$.

If the constraints and variables have been scaled (see the description of the optional argument **Scale Option**), then feasibility is defined in terms of the scaled problem (since it is more likely to be meaningful).

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

If $r \geq 0$, r defines the 'infinite' bound $infbnd$ in the definition of the problem constraints. Any upper bound greater than or equal to $infbnd$ will be regarded as $+\infty$ (and similarly any lower bound less than or equal to $-infbnd$ will be regarded as $-\infty$). If $r < 0$, the default value is used.

Iterations Limit i Default $= \max\{10000, 10\max\{m, n\}\}$

The value of i specifies the maximum number of iterations allowed before termination. Setting $i = 0$ and **Print Level** > 0 means that: the workspace needed to start solving the problem will be computed and printed; and feasibility and optimality will be checked. No iterations will be performed. If $i < 0$, the default value is used.

LU Density Tolerance r_1 Default $= 0.6$
LU Singularity Tolerance r_2 Default $= \epsilon^{\frac{2}{3}}$

The density tolerance r_1 is used during LU factorization of the basis matrix. Columns of L and rows of U are formed one at a time, and the remaining rows and columns of the basis are altered appropriately. At any stage, if the density of the remaining matrix exceeds r_1, the Markowitz strategy for choosing pivots is terminated. The remaining matrix is factored by a dense LU procedure. Raising the density tolerance towards 1.0 may give slightly sparser LU factors, with a slight increase in factorization time.

If $r_2 > 0$, r_2 defines the singularity tolerance used to guard against ill-conditioned basis matrices. After B is refactorized, the diagonal elements of U are tested as follows. If $|u_{jj}| \leq r_2$ or $|u_{jj}| < r_2 \max_i |u_{ij}|$, the jth column of the basis is replaced by the corresponding slack variable. If $r_2 \leq 0$, the default value is used.

LU Factor Tolerance r_1 Default $= 100.0$
LU Update Tolerance r_2 Default $= 10.0$

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

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

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

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

For certain very regular structures (e.g., band matrices) it may be necessary to reduce r_1 and/or r_2 in order to achieve stability. For example, if the columns of A include a sub-matrix of the form

$$\begin{pmatrix} 4 & -1 & & & & \\ -1 & 4 & -1 & & & \\ & -1 & 4 & -1 & & \\ & & \cdots & \cdots & \cdots & \\ & & & -1 & 4 & -1 \\ & & & & -1 & 4 \end{pmatrix},$$

one should set both r_1 and r_2 to values in the range $1.0 \leq r_i < 4.0$.

LU Partial Pivoting Default
LU Complete Pivoting
LU Rook Pivoting

The LU factorization implements a Markowitz-type search for pivots that locally minimize the fill-in subject to a threshold pivoting stability criterion. The default option is to use threshold partial pivoting. The options **LU Complete Pivoting** and **LU Rook Pivoting** are more expensive but more stable and better at revealing rank, as long as the **LU Factor Tolerance** is not too large (say < 2.0).

Minimize Default
Maximize
Feasible Point

This option specifies the required direction of the optimization. It applies to both linear and nonlinear terms (if any) in the objective function. Note that if two problems are the same except that one minimizes $f(x)$ and the other maximizes $-f(x)$, their solutions will be the same but the signs of the dual variables π_i and the reduced gradients d_j (see Section 11.3) will be reversed.

The option **Feasible Point** means 'ignore the objective function, while finding a feasible point for the linear constraints'. It can be used to check that the constraints are feasible without altering the call to nag_opt_sparse_convex_qp_solve (e04nqc).

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

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

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

If **New Basis File** > 0, a basis map will be saved on file **New Basis File** every i_3th iteration, where i_3 is the **Save Frequency**. The first record of the file will contain the word `PROCEEDING` if the run is still in progress. A basis map will also be saved at the end of a run, with some other word indicating the final solution status.

If **Backup Basis File** > 0, **Backup Basis File** is intended as a safeguard against losing the results of a long run. Suppose that a **New Basis File** is being saved every 100 (**Save Frequency**) iterations, and that nag_opt_sparse_convex_qp_solve (e04nqc) is about to save such a basis at iteration 2000. It is

conceivable that the run may be interrupted during the next few milliseconds (in the middle of the save). In this case the Basis file will be corrupted and the run will have been essentially wasted.

To eliminate this risk, both a **New Basis File** and a **Backup Basis File** may be specified. The following would be suitable for the above example:

```
Backup Basis FileID1
New Basis FileID2
```

where `FileID1` and `FileID2` are returned by nag_open_file (x04acc).

The current basis will then be saved every 100 iterations, first on `FileID2` and then immediately on `FileID1`. If the run is interrupted at iteration 2000 during the save on `FileID2`, there will still be a usable basis on `FileID1` (corresponding to iteration 1900).

Note that a new basis will be saved in **New Basis File** at the end of a run if it terminates normally, but it will not be saved in **Backup Basis File**. In the above example, if an optimum solution is found at iteration 2050 (or if the iteration limit is 2050), the final basis on `FileID2` will correspond to iteration 2050, but the last basis saved on `FileID1` will be the one for iteration 2000.

A full description of information recorded in **New Basis File** and **Backup Basis File** is given in Gill *et al.* (2005a).

Nolist Default
List

Normally each optional argument specification is printed to unit **Print File** as it is supplied. Optional argument **Nolist** may be used to suppress the printing and optional argument **List** may be used to restore printing.

Old Basis File i Default $= 0$

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

If **Old Basis File** > 0, the basis maps information will be obtained from the file associated with ID i. The file will usually have been output previously as a **New Basis File** or **Backup Basis File**. A full description of information recorded in **New Basis File** and **Backup Basis File** is given in Gill *et al.* (2005a).

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

Optimality Tolerance r Default $= \max\{10^{-6}, \sqrt{\epsilon}\}$

This is used to judge the size of the reduced gradients $d_j = g_j - a_j^{\mathrm{T}}\pi$, where g_j is the jth component of the gradient, a_j is the associated column of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$, and π is the set of dual variables.

By construction, the reduced gradients for basic variables are always zero. The problem will be declared optimal if the reduced gradients for nonbasic variables at their lower or upper bounds satisfy

$$d_j/\|\pi\| \geq -r \quad \text{or} \quad d_j/\|\pi\| \leq r$$

respectively, and if $|d_j|/\|\pi\| \leq r$ for superbasic variables.

In the above tests, $\|\pi\|$ is a measure of the size of the dual variables. It is included to make the tests independent of a scale factor on the objective function. The quantity $\|\pi\|$ actually used is defined by

$$\|\pi\| = \max(\sigma/\sqrt{m}, 1), \quad \text{where } \sigma = \sum_{i=1}^{m} |\pi_i|,$$

so that only large scale factors are allowed for.

If the objective is scaled down to be very *small*, the optimality test reduces to comparing d_j against $0.01r$.

Partial Price i Default $= 10$(LP) or 1(QP)

This option is recommended for large FP or LP problems that have significantly more variables than constraints (i.e., $n \gg m$). It reduces the work required for each pricing operation (i.e., when a nonbasic variable is selected to enter the basis). If $i = 1$, all columns of the constraint matrix $\begin{pmatrix} A & -I \end{pmatrix}$ are searched. If $i > 1$, A and I are partitioned to give i roughly equal segments A_j, I_j, for $j = 1, 2, \ldots, i$ (modulo i). If the previous pricing search was successful on A_{j-1}, I_{j-1}, the next search begins on the segments A_j and I_j. If a reduced gradient is found that is larger than some dynamic tolerance, the variable with the largest such reduced gradient (of appropriate sign) is selected to enter the basis. If nothing is found, the search continues on the next segments A_{j+1}, I_{j+1}, and so on. If $i \leq 0$, the default value is used.

Pivot Tolerance r Default $= \epsilon^{\frac{2}{3}}$

Broadly speaking, the pivot tolerance is used to prevent columns entering the basis if they would cause the basis to become almost singular.

When x changes to $x + \alpha p$ for some search direction p, a 'ratio test' determines which component of x reaches an upper or lower bound first. The corresponding element of p is called the pivot element. Elements of p are ignored (and therefore cannot be pivot elements) if they are smaller than the pivot tolerance r.

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

Print File i Default $= 0$

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

If **Print File** > 0, the following information is output to **Print File** during the solution of each problem:

– a listing of the optional arguments;

– some statistics about the problem;

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

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

– the iteration log;

– basis factorization statistics;

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

– the printed solution, if requested.

The last four items are described in Sections 9 and 13. Further brief output may be directed to the **Summary File**.

Print Frequency i Default $= 100$

If $i > 0$, one line of the iteration log will be printed every ith iteration. A value such as $i = 10$ is suggested for those interested only in the final solution. If $i \leq 0$, the value of $i = 99999999$ is used and effectively no checks are made.

Print Level i Default $= 1$

This controls the amount of printing produced by nag_opt_sparse_convex_qp_solve (e04nqc) as follows.

i	**Meaning**
0	No output except error messages. If you want to suppress all output, set **Print File** $= 0$.

$= 1$ The set of selected options, problem statistics, summary of the scaling procedure, information about the initial basis resulting from a Crash or a Basis file, a single line of output at each iteration (controlled by the optional argument **Print Frequency**), and the exit condition with a summary of the final solution.

≥ 10 Basis factorization statistics.

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

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

These files provide compatibility with commercial mathematical programming systems. The **Punch File** from a previous run may be used as an **Insert File** for a later run on the same problem. A full description of information recorded in **Insert File** and **Punch File** is given in Gill *et al.* (2005a).

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

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

QPSolver Cholesky Default
QPSolver CG
QPSolver QN

Specifies the active-set algorithm used to solve the quadratic program in Phase 2 (γ). **QPSolver Cholesky** holds the full Cholesky factor R of the reduced Hessian $Z^{\mathrm{T}}HZ$. As the QP iterations proceed, the dimension of R changes with the number of superbasic variables. If the number of superbasic variables needs to increase beyond the value of **Reduced Hessian Dimension**, the reduced Hessian cannot be stored and the solver switches to **QPSolver CG**. The Cholesky solver is reactivated if the number of superbasics stabilizes at a value less than **Reduced Hessian Dimension**.

QPSolver QN solves the QP using a quasi-Newton method. In this case, R is the factor of a quasi-Newton approximate Hessian.

QPSolver CG uses an active-set method similar to **QPSolver QN**, but uses the conjugate-gradient method to solve all systems involving the reduced Hessian.

The Cholesky QP solver is the most robust, but may require a significant amount of computation if there are many superbasics.

The quasi-Newton QP solver does not require computation of the exact R at the start of Phase 2 (γ). It may be appropriate when the number of superbasics is large but relatively few iterations are needed to reach a solution (e.g., if nag_opt_sparse_convex_qp_solve (e04nqc) is called with a Warm Start).

The conjugate-gradient QP solver is appropriate for problems with many degrees of freedom (say, more than 2000 superbasics).

Reduced Hessian Dimension i Default $= 1$(LP) or $\min(2000, n_H + 1, n)$(QP)

This specifies that an i by i triangular matrix R (to define the reduced Hessian according to $R^{\mathrm{T}}R = Z^{\mathrm{T}}HZ$). is to be available for use by the Cholesky QP solver.

Scale Option i Default $= 2$
Scale Tolerance r Default $= 0.9$
Scale Print

Three scale options are available as follows:

i	**Meaning**
0	No scaling. This is recommended if it is known that x and the constraint matrix never have very large elements (say, larger than 100).

1 The constraints and variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0 (see Fourer (1982)). This will sometimes improve the performance of the solution procedures.

2 The constraints and variables are scaled by the iterative procedure. Also, a certain additional scaling is performed that may be helpful if the right-hand side b or the solution x is large. This takes into account columns of $\begin{pmatrix} A & -I \end{pmatrix}$ that are fixed or have positive lower bounds or negative upper bounds.

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

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

If $\max_j \rho_j$ is less than r times its previous value, another scaling pass is performed to adjust the row and column scales. Raising r from 0.9 to 0.99 (say) usually increases the number of scaling passes through A. At most 10 passes are made. The value of r should lie in the range $0 < r < 1$.

Scale Print causes the row scales $r(i)$ and column scales $c(j)$ to be printed to **Print File**, if **System Information Yes** has been specified. The scaled matrix coefficients are $\bar{a}_{ij} = a_{ij}c(j)/r(i)$, and the scaled bounds on the variables and slacks are $\bar{l}_j = l_j/c(j)$, $\bar{u}_j = u_j/c(j)$, where $c(j) = r(j-n)$ if $j > n$.

Solution Yes Default
Solution No

This option determines if the final obtained solution is to be output to the **Print File**. Note that the **Solution File** option operates independently.

Solution File i Default $= 0$

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

If **Solution File** > 0, the final solution will be output to file **Solution File** (whether optimal or not).

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

Summary File i_1 Default $= 0$
Summary Frequency i_2 Default $= 100$

(See Section 3.2.1.1 in the Essential Introduction for further information on NAG data types.)

If **Summary File** > 0, a brief log will be output to file **Summary File**, including one line of information every i_2th iteration. In an interactive environment, it is useful to direct this output to the terminal, to allow a run to be monitored online. (If something looks wrong, the run can be manually terminated.) Further details are given in Section 13. If $i_2 \leq 0$, the value of $i_2 = 99999999$ is used and effectively no checks are made.

Superbasics Limit i Default $= 1$(LP) or $\min\{n_H + 1, n\}$(QP)

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

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

For quadratic problems, the number of degrees of freedom is often called the 'number of independent variables'. Normally, i need not be greater than $n_H + 1$, where n_H is the number of leading nonzero columns of H. For many problems, i may be considerably smaller than n_H. This will save storage if n_H is very large.

Suppress Parameters

Normally nag_opt_sparse_convex_qp_solve (e04nqc) prints the options file as it is being read, and then prints a complete list of the available keywords and their final values. The optional argument **Suppress Parameters** tells nag_opt_sparse_convex_qp_solve (e04nqc) not to print the full list.

System Information No Default
System Information Yes

This option prints additional information on the progress of major and minor iterations, and Crash statistics. See Section 13.

Timing Level i Default $= 0$

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

Unbounded Step Size r Default $= infbnd$

If $r > 0$, r 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 not positive definite.) If the change in x during an iteration would exceed the value of r, the objective function is considered to be unbounded below in the feasible region. If $r \leq 0$, the default value is used. See **Infinite Bound Size** for the definition of $infbnd$.

13 Description of Monitoring Information

This section describes the intermediate printout and final printout which constitutes the monitoring information produced by nag_opt_sparse_convex_qp_solve (e04nqc). (See also the description of the optional arguments **Print File** and **Print Level**.) You can control the level of printed output.

13.1 Crash Statistics

When **Print Level** ≥ 10, **Print File** > 0 and **System Information Yes** has been specified, the following lines of intermediate printout (less than 120 characters) are produced on the unit number specified by optional argument **Print File** whenever **start** = Nag_Cold (see Section 5). They refer to the number of columns selected by the Crash procedure during each of several passes through A, whilst searching for a triangular basis matrix.

Label	Description
`Slacks`	is the number of slacks selected initially.
`Free cols`	is the number of free columns in the basis, including those whose bounds are rather far apart.
`Preferred`	is the number of 'preferred' columns in the basis (i.e., $\mathbf{hs}[j-1] = 3$ for some $j \leq n$). It will be a subset of the columns for which $\mathbf{hs}[j-1] = 3$ was specified.
`Unit`	is the number of unit columns in the basis.
`Double`	is the number of double columns in the basis.
`Triangle`	is the number of triangular columns in the basis.
`Pad`	is the number of slacks used to pad the basis (to make it a nonsingular triangle).

13.2 Basis Factorization Statistics

When **Print Level** ≥ 10 and **Print File** > 0, the first seven items of intermediate printout in the list below are produced on the unit number specified by optional argument **Print File** whenever the matrix B or $B_S = \begin{pmatrix} B & S \end{pmatrix}^{\mathrm{T}}$ is factorized. Gaussian elimination is used to compute an LU factorization of B or B_S, where PLP^{T} is a lower triangular matrix and PUQ is an upper triangular matrix for some permutation matrices P and Q. The factorization is stabilized in the manner described under the optional

argument **LU Factor Tolerance**. In addition, if **System Information Yes** has been specified, the entries from `Elems` onwards are also output.

Label **Description**

`Factor` the number of factorizations since the start of the run.

`Demand` a code giving the reason for the present factorization, as follows:

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

`Itn` is the current minor iteration number.

`Nonlin` is the number of nonlinear variables in the current basis B.

`Linear` is the number of linear variables in B.

`Slacks` is the number of slack variables in B.

`B`, `BR`, `BS` or `BT` `factorize`
is the type of LU factorization.

`B`	periodic factorization of the basis B.
`BR`	more careful rank-revealing factorization of B using threshold rook pivoting. This occurs mainly at the start, if the first basis factors seem singular or ill-conditioned. Followed by a normal `B` factorize.
`BS`	B_S is factorized to choose a well-conditioned B from the current $(B\ S)$. Followed by a normal `B` factorize.
`BT`	same as `BS` except the current B is tried first and accepted if it appears to be not much more ill-conditioned than after the previous `BS` factorize.

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

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

`Elems` is the number of nonzero elements in B or B_S.

`Amax` is the largest nonzero in B or B_S.

`Density` is the percentage nonzero density of B or B_S.

`Merit/MerRP/MerCP` `Merit` is the average Markowitz merit count for the elements chosen to be the diagonals of PUQ. Each merit count is defined to be $(c-1)(r-1)$ where c and r are the number of nonzeros in the column and row containing the element at the time it is selected to be the next diagonal. `Merit` is the average of `n` such quantities. It gives an indication of how much work was required to preserve sparsity during the factorization. If **LU Complete Pivoting** or **LU Rook Pivoting** has been selected, this heading is changed to `MerCP`, respectively `MerRP`.

`lenL` is the number of nonzeros in L.

`L+U` is the number of nonzeros representing the basis factors L and U. Immediately after a basis factorization $B = LU$, this is `lenL+lenU`, the number of subdiagonal elements in the columns of a lower triangular matrix and the number of diagonal and superdiagonal elements in the rows of an upper-triangular matrix. Further nonzeros are added to `L` when various columns of B are later replaced. As columns of B are replaced, the matrix U is maintained explicitly (in sparse form). The value of `L` will steadily increase, whereas the value of `U` may fluctuate up or

down. Thus the value of `L+U` may fluctuate up or down (in general, it will tend to increase).

`Cmpressns` is the number of times the data structure holding the partially factored matrix needed to be compressed to recover unused storage. Ideally this number should be zero. If it is more than 3 or 4, the amount of workspace available to nag_opt_sparse_convex_qp_solve (e04nqc) should be increased for efficiency.

`Incres` is the percentage increase in the number of nonzeros in L and U relative to the number of nonzeros in B or B_S.

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

`lenU` the number of nonzeros in U, including its diagonals.

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

`Umax` the maximum nonzero element in U.

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

As long as `Lmax` is not large (say 5.0 or less), $\max(\texttt{Amax},\texttt{Umax})/\texttt{DUmin}$ gives an estimate of the condition number B. If this is extremely large, the basis is nearly singular. Slacks are used to replace suspect columns of B and the modified basis is refactored.

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

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

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

`Akmax` is the largest nonzero generated at any stage of the LU factorization. (Values much larger than `Amax` indicate instability.) `Akmax` is not printed if **LU Partial Pivoting** is selected.

`Agrwth` is the ratio `Akmax`/`Amax`. Values much larger than 100 (say) indicate instability. `Agrwth` is not printed if **LU Partial Pivoting** is selected.

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

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

`DUmax` is the largest diagonal of PUQ.

`DUmin` is the smallest diagonal of PUQ.

`condU` the ratio `DUmax`/`DUmin`, which estimates the condition number of U (and of B if `Ltol` is less than 5.0, say).

13.3 Basis Map

When **Print Level** ≥ 10 and **Print File** > 0, the following lines of intermediate printout (less than 80 characters) are produced on the unit number specified by optional argument **Print File**. They refer to the elements of the **names** array (see Section 5).

Label	**Description**
`Name`	gives the name for the problem (blank if problem unnamed).
`Infeasibilities`	gives the number of infeasibilities. Printed only if the final point is infeasible.

`Objective Value`	gives the objective value at the final point (or the value of the sum of infeasibilities). Printed only if the final point is feasible.
`Status`	gives the exit status for the problem (i.e., `Optimal soln`, `Weak soln`, `Unbounded`, `Infeasible`, `Excess itns`, `Error condn` or `Feasble soln`) followed by details of the direction of the optimization (i.e., `(Min)` or `(Max)`).
`Iteration`	gives the iteration number when the file was created.
`Superbasics`	gives the number of superbasic variables.
`Objective`	gives the name of the free row for the problem (blank if objective unnamed).
`RHS`	gives the name of the constraint right-hand side for the problem (blank if objective unnamed).
`Ranges`	gives the name of the ranges for the problem (blank if objective unnamed).
`Bounds`	gives the name of the bounds for the problem (blank if objective unnamed).

13.4 Solution Output

At the end of a run, the final solution will be output to the Print file. Some header information appears first to identify the problem and the final state of the optimization procedure. A ROWS section and a COLUMNS section then follow, giving one line of information for each row and column.

13.4.1 The ROWS section

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

$$\alpha \leq \nu_i x \leq \beta,$$

where ν_i is the ith row of A.

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

Label	**Description**
`Number`	is the value of $n + i$. (This is used internally to refer to s_i in the intermediate output.)
`Row`	gives the name of ν_i.
`State`	the state of ν_i (the state of s_i relative to the bounds α and β). The various states possible are as follows:
	`LL` s_i is nonbasic at its lower limit, α.
	`UL` s_i is nonbasic at its upper limit, β.
	`EQ` s_i is nonbasic and fixed at the value $\alpha = \beta$.
	`FR` s_i is nonbasic and currently zero, even though it is free to take any value between its bounds α and β.
	`BS` s_i is basic.
	`SBS` s_i is superbasic.
	A key is sometimes printed before `State`. Note that unless the optional argument **Scale Option** $= 0$ is specified, the tests for assigning a key are applied to the variables of the scaled problem.
	`A` *Alternative optimum possible*. The variable is nonbasic, but its reduced gradient is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change in the value of the objective function. The values of the other free variables *might* change,

giving a genuine alternative solution. However, if there are any degenerate variables (labelled D), the actual change might prove to be zero, since one of them could encounter a bound immediately. In either case, the values of the Lagrange multipliers *might* also change.

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

I *Infeasible*. The variable is basic or superbasic and is currently violating one of its bounds by more than the value of the **Feasibility Tolerance**.

N *Not precisely optimal*. If the slack is superbasic, the dual variable π_i is not sufficiently small, as measured by the **Optimality Tolerance**. If the slack is nonbasic, π_i is not sufficiently positive or negative. If a loose **Optimality Tolerance** has been used, or if iterations were terminated before optimality, this key might be helpful in deciding whether or not to restart the run.

`Activity` is the value of $\nu_i x$ at the final iterate.

`Slack Activity` is the value by which the row differs from its nearest bound. (For the free row (if any), it is set to `Activity`.)

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

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

`Dual Activity` is the value of the dual variable π_i (the Lagrange multiplier for ν_i; see Section 11.3). For FP problems, π_i is set to zero.

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

13.4.2 The COLUMNS Section

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

Label	**Description**
`Number`	is the column number j. (This is used internally to refer to x_j in the intermediate output.)
`Column`	gives the name of x_j.
`State`	the state of x_j relative to the bounds α and β. The various states possible are as follows:

`LL` x_j is nonbasic at its lower limit, α.

`UL` x_j is nonbasic at its upper limit, β.

`EQ` x_j is nonbasic and fixed at the value $\alpha = \beta$.

`FR` x_j is nonbasic and currently zero, even though it is free to take any value between its bounds α and β.

`BS` x_j is basic.

`SBS` x_j is superbasic.

A key is sometimes printed before `State`. Note that unless the optional argument **Scale Option** $= 0$ is specified, the tests for assigning a key are applied to the variables of the scaled problem.

A *Alternative optimum possible*. The variable is nonbasic, but its reduced gradient is essentially zero. This means that if the variable were allowed to start moving away from its bound, there would be no change in the value of

the objective function. The values of the other free variables *might* change, giving a genuine alternative solution. However, if there are any degenerate variables (labelled `D`), the actual change might prove to be zero, since one of them could encounter a bound immediately. In either case, the values of the Lagrange multipliers *might* also change.

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

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

`N` *Not precisely optimal*. If the slack is superbasic, the dual variable π_i is not sufficiently small, as measured by the **Optimality Tolerance**. If the slack is nonbasic, π_i is not sufficiently positive or negative. If a loose **Optimality Tolerance** has been used, or if iterations were terminated before optimality, this key might be helpful in deciding whether or not to restart the run.

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

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

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

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

`Reduced Gradnt` is the value of d_j at the final iterate (see Section 11.3). For FP problems, d_j is set to zero.

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

Note: if two problems are the same except that one minimizes $f(x)$ and the other maximizes $-f(x)$, their solutions will be the same but the signs of the dual variables π_i and the reduced gradients d_j will be reversed.

13.5 The Solution File

If **Solution File** > 0, the information contained in a printed solution may also be output to the relevant file (which may be the Print file if so desired). Infinite Upper and Lower limits appear as $\pm 10^{20}$ rather than `None`. The maximum line length is 111 characters.

A Solution file is intended to be read from disk by a self-contained program that extracts and saves certain values as required for possible further computation. Typically the first 14 lines would be ignored. The end of the ROWS section is marked by a line that starts with a 1 and is otherwise blank. If this and the next 4 lines are skipped, the COLUMNS section (see Section 13.4.2) can then be read under the same format.

13.6 The Summary File

If **Summary File** > 0, certain brief information will be output to file. A disk file should be used to retain a concise log of each run if desired. (A **Summary File** is more easily perused than the associated **Print File**).

The following information is included:

1. The optional arguments supplied via the option setting functions, if any;
2. The Basis file loaded, if any;
3. The status of the solution after each basis factorization (whether feasible; the objective value; the number of function calls so far);
4. The same information every kth iteration, where k is the specified **Summary Frequency**;

5. Warnings and error messages;
6. The exit condition and a summary of the final solution.

Item 4 is preceded by a blank line, but item 5 is not.

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

`Constrnt` gives the name 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.