nag_opt_sparse_convex_qp (e04nkc)

1. Purpose

nag_opt_sparse_convex_qp (e04nkc) solves sparse linear programming or convex quadratic programming problems.

2. Specification

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

void nag_opt_sparse_convex_qp(Integer n, Integer m, Integer nnz,
                              Integer iobj, Integer ncolh,
                              void (*qphx)(Integer ncolh, double x[],
                                           double hx[], Nag_Comm *comm),
                              double a[], Integer ha[], Integer ka[],
                              double bl[], double bu[], double xs[],
                              Integer *ninf, double *sinf, double *obj,
                              Nag_E04_Opt *options, Nag_Comm *comm,
                              NagError *fail)
```

3. Description

nag_opt_sparse_convex_qp is designed to solve a class of quadratic programming problems that are assumed to be stated in the following general form:

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

where x is a set of variables, A is an m by n matrix and the objective function $f(x)$ may be specified in a variety of ways depending upon the particular problem to be solved. The optional parameter **minimize** (see Section 8.2) may be used to specify an alternative problem in which $f(x)$ is maximized. The possible forms for $f(x)$ are listed in Table 1 below, in which the prefixes FP, LP and QP stand for 'feasible point', 'linear programming' and 'quadratic programming' respectively, c is an n element vector and H is the n by n second-derivative matrix $\nabla^2 f(x)$ (the *Hessian matrix*).

Problem Type	**Objective Function** $f(x)$	**Hessian Matrix** H
FP	Not applicable	Not applicable
LP	$c^T x$	Not applicable
QP	$c^T x + \frac{1}{2} x^T H x$	Symmetric positive semi-definite

Table 1

For LP and QP problems, the unique global minimum value of $f(x)$ is found. For FP problems, $f(x)$ is omitted and the function attempts to find a feasible point for the set of constraints. For QP problems, a function must also be provided to compute Hx for any given vector x. (H need not be stored explicitly.)

nag_opt_sparse_convex_qp is intended to solve large-scale linear and quadratic programming problems in which the constraint 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). nag_opt_sparse_convex_qp also takes advantage of sparsity in c. (Sparsity in H can be exploited in the function that computes Hx.) 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).

If H is positive-definite, then the final x will be unique. If nag_opt_sparse_convex_qp detects that H is indefinite, it terminates immediately with an error condition (see Section 9). If H is the zero matrix, the function will still solve the resulting LP problem; however, this can be accomplished more efficiently by setting the parameter **ncolh** $= 0$ (see Section 4).

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 converts the general constraints to equalities by introducing a set of *slack variables* s, where $s = (s_1, s_2, \ldots, s_m)^T$. 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:

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

Since the slack variables s are subject to the same upper and lower bounds as the elements of Ax, the bounds on Ax and x 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 an *active-set* method. This is an iterative procedure with two phases: a *feasibility phase*, in which the sum of infeasibilities is minimized to find a feasible point; and an *optimality phase*, in which $f(x)$ is minimized by constructing a sequence of iterations that lies within the feasible region.

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 non-singular. 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 $g - A^T\pi$, 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 parameter **optim_tol** in Section 8.2).

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 $d_j = g_j - a^T\pi$, where a_j is the jth column of $(A \;\; -I)$. (The variable selected by such a process and the corresponding value of d_j (i.e., its reduced gradient) are the quantities `+S` and `dj` in the detailed printed output from nag_opt_sparse_convex_qp; see Section

8.3.) 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_j's.

nag_opt_sparse_convex_qp is based on SQOPT, which is part of the SNOPT package described in Gill *et al* (1996), which in turn utilizes routines from the MINOS package (see Murtagh and Saunders (1995)). 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 7.

4. Parameters

n

Input: n, the number of variables (excluding slacks). This is the number of columns in the linear constraint matrix A.
Constraint: $\mathbf{n} \geq 1$.

m

Input: m, the number of general linear constraints (or slacks). This is the number of rows in A, including the free row (if any; see parameter **iobj** below).
Constraint: $\mathbf{m} \geq 1$.

nnz

Input: the number of non-zero elements in A.
Constraint: $1 \leq \mathbf{nnz} \leq \mathbf{n} \times \mathbf{m}$.

iobj

Input: if $\mathbf{iobj} > 0$, row **iobj** of A is a free row containing the non-zero elements of the vector c appearing in the linear objective term $c^T x$. If $\mathbf{iobj} = 0$, there is no free row – i.e., the problem is either an FP problem (in which case **iobj** must be set to zero), or a QP problem with $c = 0$.
Constraint: $0 \leq \mathbf{iobj} \leq \mathbf{m}$.

ncolh

Input: n_H, the number of leading non-zero columns of the Hessian matrix H. For FP and LP problems, **ncolh** must be set to zero.
Constraint: $0 \leq \mathbf{ncolh} \leq \mathbf{n}$.

qphx

qphx must be supplied for QP problems to compute the matrix product Hx. If H has zero rows and columns, it is most efficient to order the variables $x = (y \quad z)^T$ so that

$$Hx = \begin{pmatrix} H_1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} H_1 y \\ 0 \end{pmatrix},$$

where the nonlinear variables y appear first as shown. For FP and LP problems, **qphx** will never be called and the NAG defined null function pointer, `NULLFN`, can be supplied in the call to nag_opt_sparse_convex_qp.

The specification for **qphx** is:

```
void qphx(Integer ncolh, double x[], double hx[], Nag_Comm *comm)
```

ncolh

Input: the number of leading non-zero columns of the Hessian matrix H, as supplied to nag_opt_sparse_convex_qp.

x[ncolh]

Input: the first **ncolh** elements of **x**.

hx[ncolh]

Output: the product Hx.

comm

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

first – Boolean

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

nf – Integer

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

user – double *
iuser – Integer *
p – Pointer

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

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

Note: **qphx** should be tested separately before being used in conjunction with nag_opt_sparse_convex_qp. The array **x** must **not** be changed by **qphx**.

a[nnz]

Input: the non-zero elements of A, ordered by increasing column index. Note that elements with the same row and column indices are not allowed. The row and column indices are specified by parameters **ha** and **ka** (see below).

ha[nnz]

Input: **ha**$[i]$ must contain the row index of the non-zero element stored in **a**$[i]$, for $i = 0, 1, \ldots,$**nnz**-1. Note that the row indices for a column may be supplied in any order.
Constraint: $1 \le$ **ha**$[i] \le$ **m**, for $i = 0, 1, \ldots,$**nnz**-1.

ka[n+1]

Input: **ka**$[j-1]$ must contain the index in **a** of the start of the jth column, for $j = 1, 2, \ldots,$**n**. To specify the jth column as empty, set **ka**$[j-1] =$ **ka**$[j]$. Note that the first and last elements of **ka** must be such that **ka**$[0] = 0$ and **ka**[**n**] = **nnz**.
Constraints:

ka$[0] = 0$,

ka$[j-1] \ge 0$ for $j = 2, 3, \ldots,$**n**,

ka[**n**] = **nnz**,

$0 \le$ **ka**$[j] -$ **ka**$[j-1] \le$ **m**, for $j = 1, 2, \ldots,$**n**.

bl[n+m]
bu[n+m]

Input: **bl** must contain the lower bounds and **bu** the upper bounds, for all the constraints in the following order. The first n elements of each array must contain the bounds on the

variables, and the next m elements the bounds for the general linear constraints Ax and the free row (if any). To specify a non-existent lower bound (i.e., $l_j = -\infty$), set **bl**[$j-1$] $\le$ $-$**inf_bound**, and to specify a non-existent upper bound (i.e., $u_j = +\infty$), set **bu**[$j-1$] $\ge$ **inf_bound**, where **inf_bound** is one of the optional parameters (default value 10^{20}, see Section 8.2). To specify the jth constraint as an equality, set **bl**[$j-1$] = **bu**[$j-1$] = β, say, where $|\beta|$ < **inf_bound**. Note that, for LP and QP problems, the lower bound corresponding to the free row must be set to $-\infty$ and stored in **bl**[**n**+**iobj**-1]; similarly, the upper bound must be set to $+\infty$ and stored in **bu**[**n**+**iobj**-1].
Constraints:

bl[j] $\le$ **bu**[j], for $j = 0, 1, \ldots,$**n**+**m**-1,

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

bl[**n**+**iobj**-1] $\le$ $-$**inf_bound** and **bu**[**n**+**iobj**-1] $\ge$ **inf_bound**, when **iobj** > 0.

xs[n+m]

Input: **xs**[$j-1$], for $j = 1, 2, \ldots,$**n** must contain the initial values of the variables, x. In addition, if a 'warm start' is specified by means of the optional parameter **start** (see Section 8.2) the elements **xs**[**n**+$i-1$], for $i = 1, 2, \ldots,$**m** must contain the initial values of the slack variables, s.
Output: the final values of the variables and slacks (x, s).

ninf

Output: the number of infeasibilities. This will be zero if an optimal solution is found, i.e., if nag_opt_sparse_convex_qp exits with **fail.code** = **NE_NOERROR** or **NW_SOLN_NOT_UNIQUE**.

sinf

Output: the sum of infeasibilities. This will be zero if **ninf** = 0. (Note that nag_opt_sparse_convex_qp does *not* attempt to compute the minimum value of **sinf** in the event that the problem is determined to be infeasible, i.e., when nag_opt_sparse_convex_qp exits with **fail.code** = **NW_NOT_FEASIBLE**.)

obj

Output: the value of the objective function. If **ninf** = 0, **obj** includes the quadratic objective term $\frac{1}{2}x^T Hx$ (if any). If **ninf** > 0, **obj** is just the linear objective term $c^T x$ (if any). For FP problems, **obj** is set to zero.

options

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

The **options** structure also allows names to be assigned to the columns and rows (i.e., the variables and constraints) of the problem, which are then used in solution output. In particular, if the problem data is defined by an MPSX file, the function nag_opt_sparse_mps_read (e04mzc) may be used to read the file, and to store the column and row names in **options** for use by nag_opt_sparse_convex_qp.

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

comm

Input/Output: structure containing pointers for communication to the user-supplied function, **qphx**, and the optional user-defined printing function; see the description of **qphx** and Section 8.3.1 for details. If the user does not need to make use of this communication feature the null pointer `NAGCOMM_NULL` may be used in the call to nag_opt_sparse_convex_qp; **comm** will then be declared internally for use in calls to user-supplied functions.

fail

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

4.1. Description of Printed Output

Intermediate and final results are printed out by default. The level of printed output can be controlled by the user with the structure member **options.print_level** (see Section 8.2). The default print level of **Nag_Soln_Iter** provides a single line of output at each iteration and the final result. This section describes the default printout produced by nag_opt_sparse_convex_qp.

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

`Itn` is the iteration count.

`Step` is the step taken along the computed search direction.

`Ninf` is the number of violated constraints (infeasibilities). This will be zero during the optimality phase.

`Sinf/Objective` is the current value of the objective function. If x is not feasible, `Sinf` gives the sum of magnitudes of constraint violations. If x is feasible, `Objective` is the value of the objective function. The output line for the final iteration of the feasibility phase (i.e., the first iteration for which `Ninf` is zero) will give the value of the true objective at the first feasible point.

During the optimality phase, the value of the objective function will be non-increasing. During the feasibility phase, the number of constraint infeasibilities will not increase until either a feasible point is found, or the optimality of the multipliers implies that no feasible point exists.

`Norm rg` is $\|d_S\|$, the Euclidean norm of the reduced gradient (see Section 7.3). During the optimality phase, this norm will be approximately zero after a unit step. For FP and LP problems, `Norm rg` is not printed.

The final printout includes a listing of the status of every variable and constraint. The following describes the printout for each variable.

`Variable` gives the name of variable j, for $j = 1, 2, \ldots,$**n**. If an **options** structure is supplied to nag_opt_sparse_convex_qp, and the **crnames** member is assigned to an array of column and row names (see Section 8.2 for details), the name supplied in **crnames**$[j-1]$ is assigned to the jth variable. Otherwise, a default name is assigned to the variable.

`State` gives the state of the variable (`LL` if nonbasic on its lower bound, `UL` if nonbasic on its upper bound, `EQ` if nonbasic and fixed, `FR` if nonbasic and strictly between its bounds, `BS` if basic and `SBS` if superbasic).

A key is sometimes printed before `State` to give some additional information about the state of a variable. Note that unless the optional parameter **scale** = **Nag_No_Scale** (default value is **Nag_ExtraScale**; see Section 8.2) 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 to (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 optional parameter **ftol** (default value = $\max(10^{-6}, \sqrt{\epsilon})$, where ϵ is the ***machine precision***; see Section 8.2).

`N` *Not precisely optimal.* The variable is nonbasic or superbasic. If the value of the reduced gradient for the variable exceeds the value of the optional parameter **optim_tol** (default value = $\max(10^{-6}, \sqrt{\epsilon})$; see Section 8.2), the solution would not be declared optimal because the reduced gradient for the variable would not be considered negligible.

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

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

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

`Lagr Mult` is the value of the Lagrange multiplier for the associated bound. This will be zero if `State` is `FR`. If x is optimal, the multiplier should be non-negative if `State` is `LL`, non-positive if `State` is `UL`, and zero if `State` is `BS` or `SBS`.

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

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

`Constrnt` gives the name of the linear constraint.

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

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

5. Comments

A list of possible error exits and warnings from nag_opt_sparse_convex_qp is given in Section 9. The accuracy of nag_opt_sparse_convex_qp is considered in Section 10.

6. Example 1

To minimize the quadratic function $f(x) = c^T x + \frac{1}{2} x^T H x$, where

$$c = (-200,\ -2000,\ -2000,\ -2000,\ -2000,\ 400,\ 400)^T$$

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

subject to the bounds

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

and the general constraints

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

The initial point, which is infeasible, is

$$x_0 = (0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0)^T.$$

The optimal solution (to five figures) is

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

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

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

The function to calculate Hx (**qphx** in the parameter list; see Section 4) is `qphess1` for this example.

6.1. Program Text

```
/* nag_opt_sparse_convex_qp(e04nkc) Example Program.
 *
 * Copyright 1998 Numerical Algorithms Group.
 *
 * Mark 5, 1998.
 *
 */
#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nage04.h>

#ifdef NAG_PROTO
static void qphess1(Integer ncolh, double x[], double hx[], Nag_Comm *comm);
static void qphess2(Integer ncolh, double x[], double hx[], Nag_Comm *comm);

static void ex1(void);
static void ex2(void);
#else
static void qphess1();
static void qphess2();
static void ex1();
static void ex2();
#endif
```

```
main()
{
  /* Two examples are called: ex1() uses the
   * default settings to solve a problem while
   * ex2() solves another problem with some
   * of the optional parameters set by the user.
   */
  Vprintf("e04nkc Example Program Results.\n");
  ex1();
  ex2();
  exit(EXIT_SUCCESS);
}

#ifdef NAG_PROTO
static void ex1(void)
#else
static void ex1()
#endif
{
#define MAXN 10
#define MAXM 10
#define MAXBND MAXN+MAXM
#define MAXNNZ 50

  double a[MAXNNZ], bl[MAXBND], bu[MAXBND];
  double x[MAXBND];
  double sinf, obj;
  Integer i, icol, j, jcol;
  Integer iobj, ncolh;
  Integer m, n, nbnd, nnz;
  Integer ninf;
  Integer ha[MAXNNZ], ka[MAXN+1];
  static NagError fail;


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

  fail.print = TRUE;

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

  /* Read nnz, iobj, ncolh */
  Vscanf(" %*[^\n]");
  Vscanf("%ld%ld%ld", &nnz, &iobj, &ncolh);

  /* Read the matrix and set up ka */
  jcol = 1;
  ka[jcol-1] = 0;
  Vscanf(" %*[^\n]");
  for (i = 0; i < nnz; ++i)
    {
      /* a[i] stores the (ha[i], icol) element of matrix */
      Vscanf("%lf%ld%ld", &a[i], &ha[i], &icol);

      /* Check whether we have started a new column */
      if (icol == jcol+1)
        {
          ka[icol-1] = i;  /* Start of icol-th column in a */
          jcol = icol;
        }
      else if (icol > jcol+1)
        {
          /* Index in a 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 ka to i.
           */
          for (j = jcol+1; j < icol; ++j)
            ka[j-1] = i;

          ka[icol-1] = i;
        }
    }
  ka[n] = nnz;

  /* Read the bounds */
  nbnd = n+m;
  Vscanf(" %*[^\n]"); /* Skip heading in data file */
  for (i = 0; i < nbnd; ++i)
    Vscanf("%lf", &bl[i]);
  Vscanf(" %*[^\n]");
  for (i = 0; i < nbnd; ++i)
    Vscanf("%lf", &bu[i]);

  /* Read the initial estimate of x */
  Vscanf(" %*[^\n]"); /* Skip heading in data file */
  for (i = 0; i < n; ++i)
    Vscanf("%lf", &x[i]);

  /* Solve the problem */
  e04nkc(n, m, nnz, iobj, ncolh, qphess1, a, ha, ka, bl, bu,
         x, &ninf, &sinf, &obj, E04_DEFAULT, NAGCOMM_NULL, &fail);

  if (fail.code != NE_NOERROR)
    exit(EXIT_FAILURE);

}    /* ex1 */

#ifdef NAG_PROTO
static void qphess1(Integer ncolh,  double x[], double hx[], Nag_Comm *comm)
#else
     static void qphess1(ncolh, x, hx, comm)
     Integer ncolh;
     double x[], hx[];
     Nag_Comm *comm;
#endif
{
  /*  Function to compute H*x. */
  hx[0] = 2.0*x[0];          /*   2.0  0.0  0.0  0.0  0.0  0.0  0.0  */
  hx[1] = 2.0*x[1];          /*   0.0  2.0  0.0  0.0  0.0  0.0  0.0  */
  hx[2] = 2.0*(x[2] + x[3]); /*   0.0  0.0  2.0  2.0  0.0  0.0  0.0  */
  hx[3] = hx[2];             /*   0.0  0.0  2.0  2.0  0.0  0.0  0.0  */
  hx[4] = 2.0*x[4];          /*   0.0  0.0  0.0  0.0  2.0  0.0  0.0  */
  hx[5] = 2.0*(x[5] + x[6]); /*   0.0  0.0  0.0  0.0  0.0  2.0  2.0  */
  hx[6] = hx[5];             /*   0.0  0.0  0.0  0.0  0.0  2.0  2.0  */

}  /* qphess1 */
```

6.2. Program Data

```
e04nkc Example Program Data

Data for example 1.

Values of n and m
7  8

Values of nnz, iobj and ncolh
48  8  7

Matrix nonzeros: value, row index, column index
    0.02   7   1
    0.02   5   1
    0.03   3   1
    1.00   1   1
    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

Lower bounds
 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

Upper bounds
 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

Initial estimate of x
 0.0  0.0  0.0  0.0  0.0  0.0  0.1
```

6.3. Program Results

```
e04nkc Example Program Results.

Example 1: default options used.

Parameters to e04nkc
--------------------

Problem type............sparse QP    Number of variables.....        7
Linear constraints......        8    Hessian columns.........        7

prob_name...............
obj_name................             rhs_name................
range_name..............             bnd_name................
crnames..............not supplied
```

```
minimize................       TRUE    start................... Nag_Cold
ftol.................... 1.00e-06    reset_ftol.............     10000
fcheck..................         60    factor_freq............       100
scale.............Nag_ExtraScale    scale_tol............... 9.00e-01
optim_tol.............. 1.00e-06    max_iter................        75
crash.............Nag_CrashTwice    crash_tol............... 1.00e-01
partial_price...........         10    pivot_tol.............. 2.04e-11
max_sb..................          7
inf_bound.............. 1.00e+20    inf_step................ 1.00e+20
lu_factor_tol........... 1.00e+02    lu_update_tol........... 1.00e+01
lu_sing_tol............. 2.04e-11    machine precision....... 1.11e-16
print_level.........Nag_Soln_Iter
outfile.................     stdout

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

   Itn      Step   Ninf   Sinf/Objective   Norm rg
Itn 0 -- Infeasible
     0   0.0e+00      1     1.152802e+03   0.0e+00
     1   4.3e+02      0     0.000000e+00   0.0e+00
Itn 1 -- Feasible point found (for 1 equality constraints).
     1   0.0e+00      0     0.000000e+00   0.0e+00
     1   0.0e+00      0     1.460000e+06   0.0e+00
Itn 1 -- Feasible QP solution.
     2   8.7e-02      0     9.409959e+05   0.0e+00
     3   5.3e-01      0    -1.056552e+06   0.0e+00
     4   1.0e+00      0    -1.462190e+06   2.3e-12
     5   1.0e+00      0    -1.698092e+06   2.2e-12
     6   4.6e-02      0    -1.764906e+06   7.0e+02
     7   1.0e+00      0    -1.811946e+06   2.8e-12
     8   1.7e-02      0    -1.847325e+06   1.7e+02
     9   1.0e+00      0    -1.847785e+06   7.0e-12

Variable State        Value   Lower Bound  Upper Bound   Lagr Mult    Residual
X1......    LL  0.00000e+00   0.0000e+00   2.0000e+02   2.361e+03   0.000e+00
X2......    BS  3.49399e+02   0.0000e+00   2.5000e+03  -3.657e-12   3.494e+02
X3......   SBS  6.48853e+02   4.0000e+02   8.0000e+02  -5.924e-12   1.511e+02
X4......   SBS  1.72847e+02   1.0000e+02   7.0000e+02   1.949e-12   7.285e+01
X5......    BS  4.07521e+02   0.0000e+00   1.5000e+03   0.000e+00   4.075e+02
X6......    BS  2.71356e+02   0.0000e+00         None  -3.280e-12   2.714e+02
X7......    BS  1.50023e+02   0.0000e+00         None  -1.413e-12   1.500e+02

Constrnt State        Value   Lower Bound  Upper Bound   Lagr Mult    Residual
R1......    EQ  2.00000e+03   2.0000e+03   2.0000e+03  -1.290e+04  -0.000e+00
R2......    BS  4.92316e+01         None   6.0000e+01   0.000e+00  -1.077e+01
R3......    UL  1.00000e+02         None   1.0000e+02  -2.325e+03   0.000e+00
R4......    BS  3.20719e+01         None   4.0000e+01   0.000e+00  -7.928e+00
R5......    BS  1.45572e+01         None   3.0000e+01   0.000e+00  -1.544e+01
R6......    LL  1.50000e+03   1.5000e+03         None   1.445e+04  -0.000e+00
R7......    LL  2.50000e+02   2.5000e+02   3.0000e+02   1.458e+04  -0.000e+00
Free Row    BS -2.98869e+06         None         None  -1.000e+00  -2.989e+06

Exit after 9 iterations.

Optimal QP solution found.

Final QP objective value =  -1.8477847e+06
```

7. Further Description

This section gives a detailed description of the algorithm used in nag_opt_sparse_convex_qp. This, and possibly the next section, Section 8, may be omitted if the more sophisticated features of the algorithm and software are not currently of interest.

7.1. Overview

nag_opt_sparse_convex_qp 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 summarize the main features of the method. Where possible, explicit reference is made to the names of variables that are parameters 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 code. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities (the quantity `Sinf` described in Section 4.1) to the quadratic objective function (the quantity `Objective`, see Section 4.1).

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

$$\begin{pmatrix} \bar{x} \\ \bar{s} \end{pmatrix} = \begin{pmatrix} x \\ s \end{pmatrix} + \alpha p, \tag{2}$$

where the *step length* α is a non-negative scalar (the printed quantity `Step`, see Section 4.1), 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.

7.2. 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 parameter **ftol**; see Section 8.2). 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

$$Wp = 0. \tag{3}$$

This characterization 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 (6) and (7) below). The direction p will satisfy (3) if

$$p = Zp_Z, \tag{4}$$

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 $(A \;\; -I)$ 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 $(A \;\; -I)$. As in Section 3 we partition the constraints $Ax - s = 0$ so that

$$Bx_B + Sx_S + Nx_N = 0, \tag{5}$$

where B is a square non-singular 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 quantity `Ns` in the detailed printed output; see Section 8.3). Given values of x_N and x_S, the basic variables x_B are adjusted so that (x, s) satisfies (5).

If P is a permutation matrix such that $(A \;\; -I)P = (B \;\; S \;\; N)$, then the working set matrix W satisfies

$$WP = \begin{pmatrix} B & S & N \\ 0 & 0 & I_N \end{pmatrix}, \tag{6}$$

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.* (1987)) to maintain sparse LU factors of the basis matrix B that alters as the working set W changes. Given the permutation P, the null space basis is given by

$$Z = P \begin{pmatrix} -B^{-1}S \\ I \\ 0 \end{pmatrix}. \tag{7}$$

This matrix is used only as an operator, i.e., it is never computed explicitly. Products of the form Zv and $Z^T g$ 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:

$$g_Z = Z^T g \quad \text{and} \quad H_Z = Z^T HZ, \tag{8}$$

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 `Cond Hz` in the detailed printed output; see Section 8.3.)

At each iteration, an upper triangular factor R is available such that $H_Z = R^T R$. Normally, R is computed from $R^T R = Z^T HZ$ 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, $n_S \le 1000$). This is guaranteed if the number of nonlinear variables is 'moderate'.

If the QP problem contains linear variables, H is positive semi-definite 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 non-singular 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 non-singular).

7.3. The 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

$$W^T \lambda = g(x). \tag{9}$$

A Lagrange multiplier λ_j corresponding to an inequality constraint in the working set is said to be *optimal* if $\lambda_j \le \sigma$ when the associated constraint is at its *upper bound*, or if $\lambda_j \ge -\sigma$ when the associated constraint is at its *lower bound*, where σ depends on the value of the optional parameter **optim_tol** (see Section 8.2). If a multiplier is non-optimal, 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 non-zero, there is no feasible point and the function terminates immediately with **fail.code** = **NW_NOT_FEASIBLE** (see Section 9).

The special form (6) of the working set allows the multiplier vector λ, the solution of (9), to be written in terms of the vector

$$d = \begin{pmatrix} g \\ 0 \end{pmatrix} - (A \;\; -I)^T \pi = \begin{pmatrix} g - A^T \pi \\ \pi \end{pmatrix}, \tag{10}$$

where π satisfies the equations $B^T\pi = g_B$, 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 (8). 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 `Norm rg`, `Reduced Gradnt`, `Obj Gradient` and `Dual Activity` in the detailed printed output; see Section 8.3.)

If the reduced gradient is not zero, Lagrange multipliers need not be computed and the search direction is given by $p = Zp_Z$ (see (7) and (11)). 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 non-singular, R is non-singular and p_Z in (4) is computed from the equations

$$R^T R p_Z = -g_Z, \tag{11}$$

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_{M}, the step to the 'nearest' constraint along p. This constraint is 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

$$p_Z^T H_Z p_Z = 0 \quad \text{and} \quad g_Z^T p_Z \leq 0, \tag{12}$$

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** (see Section 9). 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.

7.4. Miscellaneous

If the basis matrix is not chosen carefully, the condition of the null space matrix Z in (7) could be arbitrarily high. To guard against this, the function implements a 'basis repair' feature in which the LUSOL package (see Gill *et al.* (1987)) is used to compute the rectangular factorization

$$(B\ S)^T = LU, \tag{13}$$

returning just the permutation P that makes PLP^T unit lower triangular. The pivot tolerance is set to require $|PLP^T|_{ij} \leq 2$, and the permutation is used to define P in (6). 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 parameter **ftol** (see Section 8.2) is δ. Over a period of K iterations (where K is the value of the optional parameter **reset_ftol**; see Section 8.2), the feasibility tolerance actually used by nag_opt_sparse_convex_qp (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 non-trivial adjustments made. If the count is non-zero, 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 infeasibilites with δ.)

The resetting procedure is also invoked when nag_opt_sparse_convex_qp reaches an apparently optimal, infeasible or unbounded solution, unless this situation has already occurred twice. If any non-trivial 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 $\alpha \leq \alpha_{\mathrm{M}}$) along p from the current point are then viewed as acceptable candidates for inclusion in the working set. The constraint whose normal makes the largest angle with the search direction is added to the working set. This strategy helps keep the basis matrix B well conditioned.

8. Optional Parameters

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

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

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

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

8.1. Optional Parameter Checklist and Default Values

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

`Nag_Start start`	**Nag_Cold**
`Boolean list`	**TRUE**
`Nag_PrintType print_level`	**Nag_Soln_Iter**
`char outfile[80]`	`stdout`
`void (*print_fun)()`	NULL
`char prob_name[9]`	'\0'
`char obj_name[9]`	'\0'
`char rhs_name[9]`	'\0'
`char range_name[9]`	'\0'
`char bnd_name[9]`	'\0'
`char **crnames`	NULL
`Boolean minimize`	TRUE
`Integer max_iter`	max(50,5(**n**+**m**))
`Nag_CrashType crash`	Nag_CrashTwice
`double crash_tol`	0.1
`Nag_ScaleType scale`	Nag_ExtraScale

`double scale_tol`	0.9
`double optim_tol`	$\max(10^{-6}, \sqrt{\epsilon})$
`double ftol`	$\max(10^{-6}, \sqrt{\epsilon})$
`Integer reset_ftol`	10000
`Integer fcheck`	60
`Integer factor_freq`	100
`Integer partial_price`	10
`double pivot_tol`	$\epsilon^{0.67}$
`double lu_factor_tol`	100.0
`double lu_update_tol`	10.0
`double lu_sing_tol`	$\epsilon^{0.67}$
`Integer max_sb`	min(**ncolh**+1,**n**)
`double inf_bound`	10^{20}
`double inf_step`	max(**inf_bound**,10^{20})
`Integer *state`	size **n**+**m**
`double *lambda`	size **n**+**m**
`Integer nsb`	
`Integer iter`	
`Integer nf`	

8.2. Description of Optional Parameters

start – Nag_Start Default = **Nag_Cold**

Input: specifies how the initial working set is to be chosen.

If **start** = **Nag_Cold**, an internal Crash procedure will be used to choose an initial basis matrix, B.

If **start** = **Nag_Warm**, the user must provide a valid definition of every array element of the optional parameter **state** (see below), probably obtained from a previous call of nag_opt_sparse_convex_qp, while, for QP problems, the optional parameter **nsb** (see below) must retain its value from a previous call.
Constraint: **options.start** = **Nag_Cold** or **Nag_Warm**.

list – Boolean Default = **TRUE**

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

print_level – Nag_PrintType Default = **Nag_Soln_Iter**

Input: the level of results printout produced by nag_opt_sparse_convex_qp. The following values are available.

Nag_NoPrint	No output.
Nag_Soln	The final solution.
Nag_Iter	One line of output for each iteration.
Nag_Iter_Long	A longer line of output for each iteration with more information (line exceeds 80 characters).
Nag_Soln_Iter	The final solution and one line of output for each iteration.
Nag_Soln_Iter_Long	The final solution and one long line of output for each iteration (line exceeds 80 characters).
Nag_Soln_Iter_Full	As **Nag_Soln_Iter_Long** with the matrix statistics (initial status of rows and columns, number of elements, density, biggest and smallest elements, etc.), factors resulting from the scaling procedure (if **options.scale** = **Nag_RowColScale** or **Nag_ExtraScale**; see below), basis factorization statistics and details of the initial basis resulting from the Crash procedure (if **options.start** = **Nag_Cold**).

Details of each level of results printout are described in Section 8.3.
Constraint: **options.print_level** = **Nag_NoPrint**, **Nag_Soln**, **Nag_Iter**, **Nag_Soln_Iter**, **Nag_Iter_Long**, **Nag_Soln_Iter_Long** or **Nag_Soln_Iter_Full**.

outfile – char[80] Default = `stdout`

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

print_fun – pointer to function Default = NULL

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

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

See Section 8.3.1 below for further details.

prob_name – char[9] Default: **prob_name**[0] = '\0'
obj_name – char[9] Default: **obj_name**[0] = '\0'
rhs_name – char[9] Default: **rhs_name**[0] = '\0'
range_name – char[9] Default: **range_name**[0] = '\0'
bnd_name – char[9] Default: **bnd_name**[0] = '\0'

Input: these options contain the names associated with the so-called MPSX form of the problem. MPSX files may be read by calling nag_opt_sparse_mps_read (e04mzc) prior to calling nag_opt_sparse_convex_qp. The parameters contain, respectively, the names of: the problem; the objective (or free) row; the constraint right hand side; the ranges, and the bounds. They are used in the detailed output when optional parameter **print_level** = **Nag_Soln_Iter_Full**.

crnames – char ** Default = NULL

Input: if **crnames** is not NULL then it must point to an array of **n**+**m** character strings with maximum string length 8, containing the names of the columns and rows (i.e., variables and constraints) of the problem. Thus, **crnames**$[j-1]$ contains the name of the the jth column (variable), $j = 1, 2, \ldots,$**n**, and **crnames**[**n**+i−1] contains the names of the ith row (constraint), $i = 1, 2, \ldots,$**m**. If supplied, the names are used in the solution output (see Section 4.1 and Section 8.3).

If a problem is defined by an MPSX file, it may be read by calling nag_opt_sparse_mps_read (e04mzc) prior to calling nag_opt_sparse_convex_qp. In this case, nag_opt_sparse_mps_read (e04mzc) may optionally be used to allocate memory to **crnames** and to read the column and row names defined in the MPSX file into **crnames**. In this case, the memory freeing function nag_opt_free (e04xzc) should be used to free the memory pointed to by **crnames** on return from nag_opt_sparse_convex_qp. Users should **not** use the standard C function `free()` for this purpose.

minimize – Boolean Default = **TRUE**

Input: **minimize** specifies the required direction of 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 7.3) will be reversed.

max_iter – Integer Default = max(50,5(**n**+**m**))

Input: **max_iter** specifies the maximum number of iterations allowed before termination.

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

Constraint: **options.max_iter** ≥ 0.

crash – Nag_Crash_Type Default = **Nag_CrashTwice**

This option does not apply when optional parameter **start** = **Nag_Warm**.

Input: if **start** = **Nag_Cold**, and internal Crash procedure is used to select an initial basis from various rows and columns of the constraint matrix $(A \;\; -I)$. The value of **crash** determines which rows and columns are initially eligible for the basis, and how many times the Crash procedure is called. If **crash** = **Nag_NoCrash**, the all-slack basis $B = -I$ is chosen. If **crash** = **Nag_CrashOnce**, the Crash procedure is called once (looking for a triangular basis in all rows and columns of the linear constraint matrix A). If **crash** = **Nag_CrashTwice**, the Crash procedure is called twice (looking at any *equality* constraints first followed by any *inequality* constraints).

If **crash** = **Nag_CrashOnce** or **Nag_CrashTwice**, certain slacks on inequality rows are selected for the basis first. (If **crash** = **Nag_CrashTwice**, 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.
Constraint: **crash** = **Nag_NoCrash**, **Nag_CrashOnce** or **Nag_CrashTwice**.

crash_tol – double Default = 0.1

Input: **crash_tol** allows the Crash procedure to ignore certain 'small' non-zero elements in the constraint matrix A while searching for a triangular basis. For each column of A, if a_{max} is the largest element in the column, other non-zeros in that column are ignored if they are less than (or equal to) $a_{max} \times$ **crash_tol**.

When **crash_tol** > 0, the basis obtained by the Crash procedure may not be strictly triangular, but it is likely to be non-singular and almost triangular. The intention is to obtain a starting basis with more column variables and fewer (arbitrary) slacks. A feasible solution may be reached earlier for some problems.
Constraint: $0.0 \leq$ **options.crash_tol** < 1.0.

scale – Nag_ScaleType Default = **Nag_ExtraScale**

Input: this option enables the scaling of the variables and constraints using an iterative procedure due to Fourer (1982), which attempts to compute row scales r_i and column scales c_j such that the scaled matrix coefficients $\bar{a}_{ij} = a_{ij} \times (c_j/r_i)$ are as close as possible to unity. This may improve the overall efficiency of the function on some problems. (The lower and upper bounds on the variables and slacks for the scaled problem are redefined as $\bar{l}_j = l_j/c_j$ and $\bar{u}_j = u_j/c_j$ respectively, where $c_j \equiv r_{j-n}$ if $j > n$.)

If **scale** = **Nag_NoScale**, no scaling is performed. If **scale** = **Nag_RowColScale**, all rows and columns of the constraint matrix A are scaled. If **scale** = **Nag_ExtraScale**, an additional scaling is performed that may be helpful when the solution x is large; it takes into account columns of $(A \;\; -I)$ that are fixed or have positive lower bounds or negative upper bounds.
Constraint: **options.scale** = **Nag_NoScale**, **Nag_RowColScale** or **Nag_ExtraScale**.

scale_tol – double Default = 0.9

This option does not apply when optional parameter **scale** = **Nag_NoScale**.

Input: **scale_tol** is used to control the number of scaling passes to be made through the constraint matrix A. At least 3 (and at most 10) passes will be made. More precisely, let a_p denote the largest column ratio (i.e., $\frac{\text{'biggest' element}}{\text{'smallest' element}}$ in some sense) after the pth scaling pass through A. The scaling procedure is terminated if $a_p \geq a_{p-1} \times$ **scale_tol** for some $p \geq 3$. Thus, increasing the value of **scale_tol** from 0.9 to 0.99 (say) will probably increase the number of passes through A.
Constraint: $0.0 <$ **options.scale_tol** < 1.0.

optim_tol – double Default = $\max(10^{-6},\sqrt{\epsilon})$

Input: **optim_tol** is used to judge the size of the reduced gradients $d_j = g_j - \pi^T a_j$. By definition, the reduced gradients for basic variables are always zero. Optimality is declared if the reduced gradients for any nonbasic variables at their lower or upper bounds satisfy $-$**optim_tol** $\times \max(1,|\pi|) \leq d_j \leq$ **optim_tol** $\times \max(1,|\pi|)$, and if $|d_j| \leq$ **optim_tol** $\times \max(1,|\pi|)$ for any superbasic variables.
Constraint: **options.optim_tol** $\geq \epsilon$.

ftol – double Default = $\max(10^{-6}, \sqrt{\epsilon})$

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

nag_opt_sparse_convex_qp 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 **ftol** by a factor of 10 or 100. Otherwise, some error in the data should be suspected. Note that nag_opt_sparse_convex_qp does *not* attempt to find the minimum value of `Sinf`.

If the constraints and variables have been scaled (see optional parameter **scale** above), then feasibility is defined in terms if the scaled problem (since it is more likely to be meaningful).
Constraint: **options.ftol** $\geq \epsilon$.

reset_ftol – Integer Default = 5

Input: this option is part of an anti-cycling procedure designed to guarantee progress even on highly degenerate problems (see Section 7.4).

For LP 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 parameter **ftol** is δ. Over a period of **reset_ftol** iterations, the feasibility tolerance actually used by nag_opt_sparse_convex_qp (i.e., the *working* feasibility tolerance) increases from 0.5δ to δ (in steps of $0.5\delta/$**reset_ftol**).

For QP problems, the same procedure is used for iterations in which there is only one superbasic variable. (Cycling can only occur when the current solution is at a vertex of the feasible region.) Thus, zero steps are allowed if there is more than one superbasic variable, but otherwise positive steps are enforced.

Increasing the value of **reset_ftol** helps reduce the number of slightly infeasible nonbasic basic variables (most of which are eliminated during the resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see **pivot_tol** below).
Constraint: $0 <$ **options.reset_ftol** < 10000000.

fcheck – Integer Default = 60

Input: every **fcheck**-th 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.
Constraint: **options.fcheck** ≥ 1.

factor_freq – Integer Default = 60

Input: at most **factor_freq** basis changes will occur between factorizations of the basis matrix. For LP problems, the basis factors are usually updated at every iteration. For QP problems, fewer basis updates will occur as the solution is apporached. The number of iterations between basis factorizations will therefore increase. During these iterations a test is made regularly according to the value of optional parameter **fcheck** to ensure that the linear constraints $Ax - s = 0$ are satisfied. If necessary, the basis will be refactorized before the limit of **factor_freq** updates is reached.
Constraint: **options.factor_freq** ≥ 1.

partial_price – Integer Default = 10

This option does not apply to QP problems.

Input: 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 **partial_price** $= 1$, all columns of the constraint matrix $(A \; -I)$ are searched. If **partial_price** > 1, A and $-I$ are partitioned to give **partial_price** roughly equal segments A_j, K_j, for $j = 1, 2, \ldots, p$ (modulo p). If the previous pricing search was successful on A_{j-1}, K_{j-1}, the next search begins on the segments A_j, K_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}, K_{j+1}, and so on.
Constraint: **options.partial_price** ≥ 1.

pivot_tol – double Default = $\epsilon^{0.67}$

Input: **pivot_tol** is used to prevent columns entering the basis if they would cause the basis to become almost singular.
Constraint: **options.pivot_tol** > 0.0.

lu_factor_tol – double Default = 100.0
lu_update_tol – double Default = 10.0

Input: **lu_factor_tol** and **lu_update_tol** 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 $|\mu| <$ **lu_factor_tol** during refactorization or $|\mu| <$ **lu_update_tol** during update. The default values of **lu_factor_tol** and **lu_update_tol** usually strike a good compromise between stability and sparsity. For large and relatively dense problems, setting **lu_factor_tol** and **lu_update_tol** to 25 (say) may give a marked improvement in sparsity without impairing stability to a serious degree. Note that for band matrices it may be necessary to set **lu_factor_tol** in the range $1 \leq$ **lu_factor_tol** < 2 in order to achieve stability.
Constraints:

options.lu_factor_tol ≥ 1.0,

options.lu_update_tol ≥ 1.0.

lu_sing_tol – double Default = $\epsilon^{0.67}$

Input: **lu_sing_tol** defines the singularity tolerance used to guard against ill conditioned basis matrices. Whenever the basis is refactorized, the diagonal elements of U are tested as follows. If $|u_{jj}| \leq$ **lu_sing_tol** or $|u_{jj}| <$ **lu_sing_tol** $\times \max_i |u_{ij}|$, the jth column of the basis is replaced by the corresponding slack variable.
Constraint: **options.lu_sing_tol** > 0.0.

max_sb – Integer Default = min(**ncolh**+1,**n**)

This option does not apply to FP or LP problems.

Input: **max_sb** places an upper bound on the number of variables which may enter the set of superbasic variables (see Section 7.2). If the number of superbasics exceeds this bound then nag_opt_sparse_convex_qp will terminate with **fail.code** = **NE_HESS_TOO_BIG**. In effect, **max_sb** specifies 'how nonlinear' the QP problem is expected to be.
Constraint: **options.max_sb** > 0.

inf_bound – double Default = 10^{20}

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

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

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

state – Integer * Default memory = **n**+**m**

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

If the option **start** = **Nag_Warm** has been chosen, **state** must point to a minimum of **n**+**m** elements of memory. This memory will already be available if the **options** structure has been used in a previous call to nag_opt_sparse_convex_qp from the calling program, with **start** = **Nag_Cold** and the same values of **n** and **m**. If a previous call has not been made sufficient memory must be allocated by the user.

If the user does supply a **state** vector and **start** = **Nag_Cold**, then the first **n** elements of **state** must specify the initial states of the problem variables. (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 $(A \quad -I)$. Possible values for **state**$[j-1]$ $(j = 1, 2, \ldots,$**n**$)$ are:

state$[j]$	**State of xs**$[j]$ **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, the user may set **state**$[j] = 0$ (and **xs**$[j] = 0.0$), for $j = 0, 1, \ldots,$**n**-1. 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, the user should set **state**$[j] = 4$ and **xs**$[j]$ = **bl**$[j]$ or **state**$[j] = 5$ and **xs**$[j]$ = **bu**$[j]$ as appropriate.

Following the Crash procedure, variables for which **state**$[j] = 2$ are made superbasic. Other variables not selected for the basis are then made nonbasic at the value **xs**$[j]$ if **bl**$[j] \leq$ **xs**$[j]$ $\leq$ **bu**$[j]$, or at the value **bl**$[j]$ or **bu**$[j]$ closest to **xs**$[j]$.

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

Constraints:

If **start** = **Nag_Cold**, $0 \leq$ **state**$[j] \leq 5$, for $j = 0, 1, \ldots,$**n**-1.

If **start** = **Nag_Warm**, $0 \leq$ **state**$[j] \leq 3$, for $j = 0, 1, \ldots,$**n**+**m**-1.

Output: the final states of the variables and slacks (x, s). The significance of each possible value of **state** is as follows:

state$[j]$	**State of variable** j	**Normal value of xs**$[j]$
0	Nonbasic	**bl**$[j]$
1	Nonbasic	**bu**$[j]$
2	Superbasic	Between **bl**$[j]$ and **bu**$[j]$
3	Basic	Between **bl**$[j]$ and **bu**$[j]$

If the problem is feasible (i.e., **ninf** = 0), basic and superbasic variables may be outside their bounds by as much as optional parameter **ftol**. Note that unless the optional parameter **scale** = **Nag_NoScale**, **ftol** 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 **ftol**, and there may be some nonbasic variables for which **xs**$[j]$ lies strictly between its bounds.

If the problem is infeasible (i.e., **ninf** > 0), some basic and superbasic variables may be outside their bounds by an arbitrary amount (bounded by **sinf** if **scale** = **Nag_NoScale**).

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

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

Output: the values of the multipliers for each constraint with respect to the current working set. The first **n** elements contain the multipliers (*reduced costs*) for the bound constraints on the variables, and the next **m** elements contain the Lagrange multipliers (*shadow prices*) for the general linear constraints.

nsb – Integer

Input: n_S, the number of superbasics. For QP problems, **nsb** need not be specified if optional parameter **start** = **Nag_Cold**, but must retain its value from a previous call when **start** = **Nag_Warm**. For FP and LP problems, **nsb** is not referenced.
Constraint: **options.nsb** ≥ 0.
Output: the final number of superbasics. This will be zero for FP and LP problems.

iter – Integer

Output: the total number of iterations performed.

nf – Integer

Output: the number of times the product Hx has been calculated (i.e., number of calls of **qphx**).

8.3. Description of Printed Output

The level of printed output can be controlled by the user with the structure members **options.list** and **options.print_level** (see Section 8.2). If **list** = **TRUE** then the parameter values to nag_opt_sparse_convex_qp are listed, whereas the printout of results is governed by the value of **print_level**. The default of **print_level** = **Nag_Soln_Iter** provides a single short line of output at each iteration and the final result. This section describes all of the possible levels of results printout available from nag_opt_sparse_convex_qp.

When **print_level** = **Nag_Iter** or **Nag_Soln_Iter** the output produced at each iteration is as described in Section 4.1. If **print_level** = **Nag_Iter_Long**, **Nag_Soln_Iter_Long** or **Nag_Soln_Iter_Full** the following, more detailed, line of output produced at every iteration. In all cases, the values of the quantities printed are those in effect *on completion* of the given iteration.

`Itn` is the iteration count.

`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.

`+S` is the variable selected by the pricing operation to be added to the superbasic set.

`-S` is the variable chosen to leave the superbasic set.

`-B` is the variable removed from the basis (if any) to become nonbasic.

`-B` is the variable chosen to leave the set of basics (if any) in a special basic $\leftrightarrow$ superbasic swap. The entry under `-S` has become basic if this entry is non-zero, and nonbasic otherwise. The swap is done to ensure that there are no superbasic slacks.

`Step` is the value of the steplength α taken allong the computed search direction p. The variables x have been changed to $x + \alpha p$. If a variable is made superbasic during the current iteration (i.e., `+S` 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 $(A \;\; -I)$) 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 parameter **pivot_tol** (default value = $\epsilon^{0.67}$, where ϵ is the ***machine precision***; see Section 8.2) to exclude very small elements of y from consideration during the computation of `Step`.

`Ninf` is the number of violated constraints (infeasibilities). This will be zero during the optimality phase.

`Sinf/Objective`	is the current value of the objective function. If x is not feasible, `Sinf` gives the sum of magnitudes of constraint violations. If x is feasible, `Objective` is the value of the objective function. The output line for the final iteration of the feasibility phase (i.e., the first iteration for which `Ninf` is zero) will give the value of the true objective at the first feasible point. During the optimality phase, the value of the objective function will be non-increasing. During the feasibility phase, the number of constraint infeasibilities will not increase until either a feasible point is found, or the optimality of the multipliers implies that no feasible point exists.
`L`	is the number of non-zeros in the basis factor L. Immediately after a basis factorization $B = LU$, this is `lenL`, the number of subdiagonal elements in the columns of a lower triangular matrix. Further non-zeros are added to `L` when various columns of B are later replaced. (Thus, `L` increases monotonically.)
`U`	is the number of non-zeros in the basis factor U. Immediately after a basis factorization, this is `lenU`, the number of diagonal and superdiagonal elements in the rows of an upper triangular matrix. 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. If it does not, nag_opt_sparse_convex_qp will attempt to expand the internal workspace allocated for the basis factors.
`Norm rg`	is $\|d_S\|$, the Euclidean norm of the reduced gradient (see Section 7.3). During the optimality phase, this norm will be approximately zero after a unit step. For FP and LP problems, `Norm rg` is not printed.
`Ns`	is the current number of superbasic variables. For FP and LP problems, `Ns` is not printed.
`Cond Hz`	is a lower bound on the condition number of the reduced Hessian (see Section 7.2). The larger this number, the more difficult the problem. For FP and LP problems, `Cond Hz` is not printed.

When **print_level** = **Nag_Soln_Iter_Full** the following intermediate printout (< 120 characters) is produced whenever the matrix B or $B_S = (B\ S)^T$ 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 parameter **lu_factor_tol** (see Section 8.2).

`Factorize`	is the factorization count.
`Demand`	is a code giving the reason for the present factorization as follows:

Code	**Meaning**
0	First LU factorization.
1	Number of updates reached the value of the optional parameter **factor_freq** (see Section 8.2).
2	Excessive non-zeros in updated factors.
7	Not enough storage to update factors.
10	Row residuals too large (see the description for the optional parameter **fcheck** in Section 8.2).
11	Ill conditioning has caused inconsistent results.

`Iteration`	is the iteration count.

`Nonlinear`	is the number of nonlinear variables in B (not printed if B_S is factorized).
`Linear`	is the number of linear variables in B (not printed if B_S is factorized).
`Slacks`	is the number of slack variables in B (not printed if B_S is factorized).
`Elems`	is the number of non-zeros in B (not printed if B_S is factorized).
`Density`	is the percentage non-zero density of B (not printed if B_S is factorized). More precisely, `Density` $= 100\times$`Elems`/(`Nonlinear`+`Linear`+`Slacks`)^2.
`Compressns`	is the number of times the data structure holding the partially factorized matrix needed to be compressed, in order to recover unused workspace.
`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 non-zeros in the column and row containing the element at the time it is selected to be the next diagonal. `Merit` is the average of `m` such quantities. It gives an indication of how much work was required to preserve sparsity during the factorization.
`lenL`	is the number of non-zeros in L.
`lenU`	is the number of non-zeros in U.
`Increase`	is the percentage increase in the number of non-zeros in L and U relative to the number of non-zeros in B. More precisely, `Increase` $= 100\times$ (`lenL`+`lenU`−`Elems`)/`Elems`.
`m`	is the number of rows in the problem. Note that `m` = `Ut` + `Lt` + `bp`.
`Ut`	is the number of triangular rows of B at the top of U.
`d1`	is the number of columns remaining when the density of the basis matrix being factorized reached 0.3.
`Lmax`	is the maximum subdiagonal element in the columns of L (not printed if B_S is factorized). This will not exceed the value of the optional parameter **lu_factor_tol**.
`Bmax`	is the maximum non-zero element in B (not printed if B_S is factorized).
`BSmax`	is the maximum non-zero element in B_S (not printed if B is factorized).
`Umax`	is the maximum non-zero element in U, excluding elements of B that remain in U unchanged. (For example, if a slack variable is in the basis, the corresponding row of B will become a row of U without modification. Elements in such rows will not contribute to `Umax`. If the basis is strictly triangular, *none* of the elements of B will contribute, and `Umax` will be zero.) Ideally, `Umax` should not be significantly larger than `Bmax`. If it is several orders of magnitude larger, it may be advisable to reset the optional parameter **lu_factor_tol** to a value near 1.0. `Umax` is not printed if B_S is factorized.
`Umin`	is the magnitude of the smallest diagonal element of PUQ (not printed if B_S is factorized).
`Growth`	is the value of the ratio `Umax`/`Bmax`, which should not be too large. Providing `Lmax` is not large (say < 10.0), the ratio max(`Bmax`, `Umax`)/`Umin` is an estimate of the condition number of B. If this number is extremely large, the basis is nearly singular and some numerical difficulties could occur in subsequent computations. (However, an effort is made to avoid near singularity by using slacks to replace columns of B that would have made `Umin` extremely small, and the modified basis is refactorized.) `Growth` is not printed if B_S is factorized.
`Lt`	is the number of triangular columns of B at the beginning of L.
`bp`	is the size of the 'bump' or block to be factorized nontrivially after the triangular rows and columns have been removed.

`d2` is the number of columns remaining when the density of the basis matrix being factorized reached 0.6.

When **print_level** = **Nag_Soln_Iter_Full** the following lines of intermediate printout ($<$ 80 characters) are produced whenever **start** = **Nag_Cold** (see Section 8.2). 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.

`Slacks` is the number of slacks selected initially.

`Free Cols` is the number of free columns in the basis.

`Preferred` is the number of 'preferred' columns in the basis (i.e., **state**$[j] = 3$ for some $j < n$).

`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.

When **print_level** = **Nag_Soln_Iter_Full** the following lines of intermediate printout ($<$ 80 characters) are produced, following the final iteration. They refer to the 'MPSX names' stored in the optional parameters **prob_name**, **obj_name**, **rhs_name**, **range_name** and **bnd_name** (see Section 8.2).

`Name` gives the name for the problem (blank if none).

`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)`).

`Objective` gives the name of the free row for the problem (blank if none).

`RHS` gives the name of the constraint right-hand side for the problem (blank if none).

`Ranges` gives the name of the ranges for the problem (blank if none).

`Bounds` gives the name of the bounds for the problem (blank if none).

When **print_level** = **Nag_Soln** or **Nag_Soln_Iter** the final solution printout for each column and row is as described in Section 4.1. When **print_level** = **Nag_Soln_Iter_Long** or **Nag_Soln_Iter_Full**, the following longer lines of final printout ($<$ 120 characters) are produced.

Let a_j denote the jth column of A, for $j = 1, 2, \ldots, n$. The following describes the printout for each column (or variable).

`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` gives the state of x_j (`LL` if nonbasic on its lower bound, `UL` if nonbasic on its upper bound, `EQ` if nonbasic and fixed, `FR` if nonbasic and strictly between its bounds, `BS` if basic and `SBS` if superbasic).

A key is sometimes printed before `State` to give some additional information about the state of x_j. Note that unless the optional parameter **scale** = **Nag_No_Scale** (default value is **Nag_ExtraScale**; see Section 8.2) is specified, the tests for assigning a key are applied to the variables of the scaled problem.

`A` *Alternative optimum possible.* x_j is nonbasic, but its reduced gradient is essentially zero. This means that if x_j 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 basic and superbasic 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.* x_j is basic or superbasic, but it is equal to (or very close to) one of its bounds.

`I` *Infeasible.* x_j is basic or superbasic and is currently violating one of its bounds by more than the value of the optional parameter **ftol** (default value $= \max(10^{-6}, \sqrt{\epsilon})$, where ϵ is the ***machine precision***; see Section 8.2).

`N` *Not precisely optimal.* x_j is nonbasic or superbasic. If the value of the reduced gradient for x_j exceeds the value of the optional parameter **optim_tol** (default value $= \max(10^{-6}, \sqrt{\epsilon})$; see Section 8.2), the solution would not be declared optimal because the reduced gradient for x_j would not be considered negligible.

`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 Bound` is the lower bound specified for x_j. (`None` indicates that **bl**$[j-1] \leq -$**inf_bound**, where **inf_bound** is the optional parameter.)

`Upper Bound` is the upper bound specified for x_j. (`None` indicates that **bu**$[j-1] \geq$ **inf_bound**.)

`Reduced Gradnt` is the value of d_j at the final iterate (see Section 7.3). For FP problems, d_j is set to zero.

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

Let v_i denote the ith row of A, for $i = 1, 2, \ldots, m$. The following describes the printout for each row (or constraint).

`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 v_i.

`State` gives the state of v_i (`LL` if active on its lower bound, `UL` if active on its upper bound, `EQ` if active and fixed, `BS` if inactive when s_i is basic and `SBS` if inactive when s_i is superbasic).

A key is sometimes printed before `State` to give some additional information about the state of s_i. Note that unless the optional parameter **scale** = **Nag_No_Scale** (default value is **Nag_ExtraScale**; see Section 8.2) is specified, the tests for assigning a key are applied to the variables of the scaled problem.

`A` *Alternative optimum possible.* s_i is nonbasic, but its reduced gradient is essentially zero. This means that if s_i 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 basic and superbasic 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 dual variables (or Lagrange multipliers) *might* also change.

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

`I` *Infeasible.* s_i is basic or superbasic and is currently violating one of its bounds by more than the value of the optional parameter **ftol** (default value $= \max(10^{-6}, \sqrt{\epsilon})$, where ϵ is the ***machine precision***; see Section 8.2).

`N` *Not precisely optimal.* s_i is nonbasic or superbasic. If the value of the reduced gradient for s_i exceeds the value of the optional parameter **optim_tol** (default value = $\max(10^{-6}, \sqrt{\epsilon})$; see Section 8.2), the solution would not be declared optimal because the reduced gradient for s_i would not be considered negligible.

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

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

`Lower Bound` is the lower bound specified for v_j. `None` indicates that **bl**$[n+j-1] \leq -$**inf_bound**, where **inf_bound** is the optional parameter.

`Upper Bound` is the upper bound specified for v_j. `None` indicates that **bu**$[n+j-1] \geq$ **inf_bound**.

`Dual Activity` is the value of the dual variable π_i (the Lagrange multiplier for v_i; see Section 7.3). For FP problems, π_i is set to zero.

`i` gives the index i of v_i.

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

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

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

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

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

The rest of this section can be skipped by a user who only wishes to use the default printing facilities.

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

If **comm->it_prt** = **TRUE** then the results from the last iteration of nag_opt_sparse_convex_qp are provided through **st**. Note that **print_fun** will be called with **comm->it_prt** = **TRUE** only if **print_level** = **Nag_Iter**, **Nag_Iter_Long**, **Nag_Soln_Iter**, **Nag_Soln_Iter_Long** or **Nag_Soln_Iter_Full**. The following members of **st** are set:

iter – Integer
the iteration count.

qp – Boolean
TRUE if a QP problem is being solved; **FALSE** otherwise.

pprice – Integer
the partial price indicator.

rgval – double
the value of the reduced gradient (or reduced cost) for the variable selected by the pricing operation at the start of the current iteration.

sb_add – Integer
the variable selected to enter the superbasic set.

sb_leave – double
the variable chosen to leave the superbasic set.

b_leave – Integer
the variable chosen to leave the basis (if any) to become nonbasic.

bswap_leave – Integer
the variable chosen to leave the basis (if any) in a special basic $\leftrightarrow$ superbasic swap.

step – double
the step length taken along the computed search direction.

pivot – Integer
the rth element of a vector y satisfying $By = a_q$ whenever a_q (the qth column of the constraint matrix $(A \ \ -I)$) replaces the rth column of the basis matrix B.

ninf – Integer
the number of violated constraints or infeasibilities.

f – double
the current value of the objective function if **st->ninf** is zero; otherwise, the sum of the magnitudes of constraint violations.

nnz_l – Integer
the number of non-zeros in the basis factor L.

nnz_u – Integer
the number of non-zeros in the basis factor U.

ncp – Integer
the number of compressions of the basis factorization workspace carried out so far.

norm_rg – double
the Euclidean norm of the reduced gradient at the start of the current iteration. This value is meaningful only if **st->qp** = **TRUE**.

nsb – Integer
the number of superbasic variables. This value is meaningful only if **st->qp** = **TRUE**.

cond_hz – double
a lower bound on the condition number of the reduced Hessian. This value is meaningful only if **st->qp** = **TRUE**.

If **comm->sol_prt** = **TRUE** then the final results for one row or column are provided through **st**. Note that **print_fun** will be called with **comm->sol_print** = **TRUE** only if **print_level** = **Nag_Soln**, **Nag_Soln_Iter**, **Nag_Soln_Iter_Long** or **Nag_Soln_Iter_Full**. The following members of **st** are set (note that **print_fun** is called repeatedly, for each row and column):

m – Integer
the number of rows (or general constraints) in the problem.

n – Integer
the number of columns (or variables) in the problem.

col – Boolean
TRUE if column information is being provided; **FALSE** if row information is being provided.

index – Integer
if **col** = **TRUE** then **index** is the index j (in the range $1 \leq j \leq$ **n**) of the current column (variable) for which the remaining members of **st**, as described below, are set.
If **col** = **FALSE** then **index** is the index i (in the range $1 \leq i \leq$ **m**) of the current row (constraint) for which the remaining members of **st**, as described below, are set.

name – char *
the name of row i or column j.

sstate – char *
sstate is a character string describing the state of row i or column j. This may be `"LL"`, `"UL"`, `"EQ"`, `"FR"`, `"BS"` or `"SBS"`. The meaning of each of these is described in Section 8.3 (`State`).

key – char *
key is a character string which gives additional information about the current row or column. The possible values of **key** are: `" "`, `"A"`, `"D"`, `"I"` or `"N"`. The meaning of each of these is described in Section 8.3 (`State`).

val – double
the activity of row i or column j at the final iterate.

blo – double
the lower bound on row i or column j.

bup – double
the upper bound on row i or column j.

lmult – double
the value of the Lagrange multiplier associated with the current row or column (i.e., the dual activity π_i for a row, or the reduced gradient d_j for a column) at the final iterate.

objg – double
the value of the objective gradient g_j at the final iterate. **objg** is meaningful only when **st->col** = **TRUE** and should not be accessed otherwise. It is set to zero for FP problems.

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

it_prt – Boolean
will be **TRUE** when the print function is called with the result of the current iteration.

sol_prt – Boolean
will be **TRUE** when the print function is called with the final result.

user – double *
iuser – Integer *
p – Pointer
pointers for communication of user information. If used they must be allocated memory by the user either before entry to nag_opt_sparse_convex_qp or during a call to **qphess** or **print_fun**. The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.

9. Error Indications and Warnings

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

NE_INVALID_INT_RANGE_1
Value ⟨*value*⟩ given to **nnz** is not valid. Correct range is $1 \leq$ **nnz** $\leq$ **n** $\times$ **m**.
Value ⟨*value*⟩ given to **iobj** is not valid. Correct range is $0 \leq$ **iobj** $\leq$ **m**.
Value ⟨*value*⟩ given to **ncolh** is not valid. Correct range is $0 \leq$ **ncolh** $\leq$ **n**.

NE_NULL_QPHX
Since argument **ncolh** is non-zero, the problem is assumed to be of type QP. However, the argument **qphx** is a null function. **qphx** must be non-null for QP problems.

NE_INT_ARRAY_2
Value ⟨*value*⟩ given to **ha**[⟨*value*⟩] not valid. Correct range for elements of **ha** is 1 to **m**.

NE_ARRAY_CONS
The contents of array **ka** are not valid. Constraint: **ka**[0] = 0.
The contents of array **ka** are not valid. Constraint: **ka**[**n**] = **nnz**.
The contents of array **ka** are not valid. Constraint: $0 \leq$ **ka**$[i+1] -$ **ka**$[i] \leq$ **m**, for $0 \leq i <$ **n**.

NE_INT_ARRAY_1
Value ⟨*value*⟩ given to **ka**[⟨*value*⟩] not valid. Correct range for elements of **ka** is ≥ 0.

NE_DUPLICATE_ELEMENT
Duplicate sparse matrix element found in row ⟨*value*⟩, column ⟨*value*⟩.

NE_OBJ_BOUND
Invalid lower bound for objective row. Bound should be $\leq$ ⟨*value*⟩.
Invalid upper bound for objective row. Bound should be $\geq$ ⟨*value*⟩.

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

NE_BOUND_EQ

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

NE_BOUND_LCON

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

NE_BOUND_EQ_LCON

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

NE_OPT_NOT_INIT

Options structure not initialized.

NE_INT_OPT_ARG_LT

On entry, **options.fcheck** = ⟨*value*⟩. Constraint: **fcheck** ≥ 1
On entry, **options.factor_freq** = ⟨*value*⟩. Constraint: **factor_freq** ≥ 1
On entry, **options.max_iter** = ⟨*value*⟩. Constraint: **max_iter** ≥ 0
On entry, **options.max_sb** = ⟨*value*⟩. Constraint: **max_sb** ≥ 1
On entry, **options.nsb** = ⟨*value*⟩. Constraint: **nsb** ≥ 0
On entry, **options.partial_price** = ⟨*value*⟩. Constraint: **partial_price** ≥ 1.

NE_BAD_PARAM

On entry parameter **options.crash** had an illegal value.
On entry parameter **options.print_level** had an illegal value.
On entry parameter **options.scale** had an illegal value.
On entry parameter **options.start** had an illegal value.

NE_INVALID_REAL_RANGE_FF

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

NE_INVALID_INT_RANGE_2

Value ⟨*value*⟩ given to **options.reset_ftol** is not valid. Correct range is
$0 <$ **reset_ftol** < 10000000.

NE_INVALID_REAL_RANGE_F

Value ⟨*value*⟩ given to **options.ftol** is not valid. Correct range is **ftol** $\geq \epsilon$.
Value ⟨*value*⟩ given to **options.inf_bound** is not valid. Correct range is **inf_bound** > 0.0.
Value ⟨*value*⟩ given to **options.inf_step** is not valid. Correct range is **inf_step** > 0.0.
Value ⟨*value*⟩ given to **options.lu_factor_tol** is not valid. Correct range is **lu_factor_tol** ≥ 1.0.
Value ⟨*value*⟩ given to **options.lu_sing_tol** is not valid. Correct range is **lu_sing_tol** > 0.0.
Value ⟨*value*⟩ given to **options.lu_update_tol** is not valid. Correct range is **lu_update_tol** ≥ 1.0.
Value ⟨*value*⟩ given to **options.lu_optim_tol** is not valid. Correct range is **optim_tol** $\geq \epsilon$.
Value ⟨*value*⟩ given to **options.pivot_tol** is not valid. Correct range is **pivot_tol** > 0.0.

NE_NAME_TOO_LONG

The string pointed to by **options.crnames**[⟨*value*⟩] is too long. It should be no longer than 8 characters.

NE_STATE_VAL

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

NW_SOLN_NOT_UNIQUE

Optimal solution is not unique.

Weak solution found. The final x is not unique, although x gives the global minimum value of the objective function.

NE_UNBOUNDED

Solution appears to be unbounded.

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

NW_NOT_FEASIBLE

No feasible point was found for the linear constraints.

The problem is infeasible. The general constraints cannot all be satisfied simultaneously to within the value of the optional parameter **ftol**; see Section 8.2.

NW_TOO_MANY_ITER

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

Too many iterations. The value of the optional parameter **max_iter** is too small; see Section 8.2.

NE_HESS_TOO_BIG

Reduced Hessian exceeds assigned dimension. **options.max_sb** = ⟨*value*⟩.

The reduced Hessian matrix Z^THZ (see Section 7.2) exceeds its assigned dimension. The value of the optional parameter **max_sb** is too small; see Section 8.2.

NE_HESS_INDEF

The Hessian matrix H appears to be indefinite.

The Hessian matrix H appears to be indefinite. Check that function **qphx** has been coded correctly and that all relevant elements of Hx have been assigned their correct values.

NE_BASIS_ILL_COND

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

NE_BASIS_SINGULAR

The basis is singular after 15 attempts to factorize it.

The basis is singular after 15 attempts to factorize it (adding slacks where necessary). Either the problem is badly scaled or the value of the optional parameter **lu_factor_tol** is too large; see Section 8.2.

NE_ALLOC_FAIL

Memory allocation failed.

NE_NOT_APPEND_FILE

Cannot open file ⟨*string*⟩ for appending.

NE_WRITE_ERROR

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

NE_NOT_CLOSE_FILE

Cannot close file ⟨*string*⟩.

NE_INTERNAL_ERROR

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

10. Further Comments

10.1. Accuracy

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

11. 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, 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.

Gill P E, Murray W and Saunders M A (1996) *SNOPT: An SQP Algorithm for Large-scale Constrained Optimization* Numerical Analysis Report 96–2. Department of Mathematics, University of California, San Diego.

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 96–010.* Department of Mathematics and Statistics, University of Edinburgh.

Murtagh B A and Saunders M A (1995) *MINOS 5.4 User's Guide* Report SOL 83–20R. Department of Operations Research, Stanford University.

12. See Also

nag_opt_lp (e04mfc)
nag_opt_sparse_mps_free (e04myc)
nag_opt_sparse_mps_read (e04mzc)
nag_opt_lin_lsq (e04ncc)
nag_opt_qp (e04nfc)
nag_opt_init (e04xxc)
nag_opt_read (e04xyc)
nag_opt_free (e04xzc)

13. Example 2

This example solves the same problem as Example 1 but illustrates the use of the **options** and **comm** structures. In addition to the data as read in Example 1, the data for this example also includes a set of user-defined column and row names, and data for the Hessian in a sparse storage format (see below for more details). The **options** structure is initialized by nag_opt_init (e04xxc) and the **crnames** member is assigned to the array of character strings into which the column and row names were read. The **p** member of **comm** is used to pass the Hessian into nag_opt_sparse_convex_qp for use by the function `qphess2`.

On return from nag_opt_sparse_convex_qp, the Hessian data is perturbed slightly and two further options set, selecting a warm start and a reduced level of printout. nag_opt_sparse_convex_qp is then called for a second time. Finally, the memory freeing function nag_opt_free (e04xzc) is used to free the memory assigned by nag_opt_sparse_convex_qp to the pointers in the options structure. Users should **not** use the standard C function `free()` for this purpose.

The sparse storage scheme used for the Hessian in this example is similar to that which nag_opt_sparse_convex_qp uses for the constraint matrix **a**, but since the Hessian is symmetric we need only store the lower triangle (including the diagonal) of the matrix. Thus, an array `hess` contains the nonzero elements of the lower triangle arranged in order of increasing column index. The array `khess` contains the indices in `hess` of the first element in each column, and the array `hhess` contains the row index associated with each element in `hess`. To allow the data to be passed via the **p** member of **comm**, a struct `HessianData` is declared, containing pointer members which are assigned to the three arrays defining the Hessian. Alternative approaches would have been to use the **user** and **iuser** members of **comm** to pass suitably partitioned arrays to `qphess2`, or to avoid the use of **comm** altogether and declare the Hessian data as global. The storage scheme suggested here is for illustrative purposes only.

13.1. Program Text

```
/* Example 2 */

/* Declare a data structure for passing sparse Hessian data to qphess2 */

typedef struct
{
  double *hess;
  Integer *khess;
```

```
  Integer *hhess;
} HessianData;

#ifdef NAG_PROTO
static void ex2(void)
#else
static void ex2()
#endif
{
#define MAXHESSNNZ MAXNNZ
  char names[MAXBND][9];
  char *crnames[MAXBND];
  double a[MAXNNZ], bl[MAXBND], bu[MAXBND];
  double hess[MAXHESSNNZ];
  double x[MAXBND];
  double sinf, obj;
  Integer i, icol, j, jcol;
  Integer iobj, ncolh;
  Integer m, n, nbnd, nnz, nnz_hess;
  Integer ninf;
  Integer ha[MAXNNZ], ka[MAXN+1];
  Integer hhess[MAXHESSNNZ], khess[MAXN+1];
  HessianData hess_data;
  Nag_Comm comm;
  Nag_E04_Opt options;
  static NagError fail;

  Vprintf("\nExample 2: some optional parameters are set.\n");
  Vscanf(" %*[^\n]");

  fail.print = TRUE;

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

  /* Read nnz, iobj, ncolh */
  Vscanf(" %*[^\n]");
  Vscanf("%ld%ld%ld", &nnz, &iobj, &ncolh);

  /* Read the matrix and set up ka */
  jcol = 1;
  ka[jcol-1] = 0;
  Vscanf(" %*[^\n]");
  for (i = 0; i < nnz; ++i)
    {
      /* a[i] stores the (ha[i], icol) element of matrix */
      Vscanf("%lf%ld%ld", &a[i], &ha[i], &icol);

      /* Check whether we have started a new column */
      if (icol == jcol+1)
        {
          ka[icol-1] = i;  /* Start of icol-th column in a */
          jcol = icol;
        }
      else if (icol > jcol+1)
        {
          /* Index in a 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 ka to i.
           */
          for (j = jcol+1; j < icol; ++j)
            ka[j-1] = i;

          ka[icol-1] = i;
        }
    }
  ka[n] = nnz;
```

```
/* Read the bounds */
nbnd = n+m;
Vscanf(" %*[^\n]"); /* Skip heading in data file */
for (i = 0; i < nbnd; ++i)
  Vscanf("%lf", &bl[i]);
Vscanf(" %*[^\n]");
for (i = 0; i < nbnd; ++i)
  Vscanf("%lf", &bu[i]);

/* Read the column and row names */
Vscanf(" %*[^\n]"); /* Skip heading in data file */
Vscanf(" %*[^']");
for (i = 0; i < nbnd; ++i)
  {
    Vscanf(" '%8c'", names[i]);
    names[i][8] = '\0';
    crnames[i] = names[i];
  }

/* Read the initial estimate of x */
Vscanf(" %*[^\n]"); /* Skip heading in data file */
for (i = 0; i < n; ++i)
  Vscanf("%lf", &x[i]);

/* Read nnz_hess */
Vscanf(" %*[^\n]");
Vscanf("%ld", &nnz_hess);

/* Read the hessian matrix and set up khess */
jcol = 1;
khess[jcol-1] = 0;
Vscanf(" %*[^\n]");
for (i = 0; i < nnz_hess; ++i)
  {
    /* hess[i] stores the (hhess[i], icol) element of matrix */
    Vscanf("%lf%ld%ld", &hess[i], &hhess[i], &icol);

    /* Check whether we have started a new column */
    if (icol == jcol+1)
      {
        khess[icol-1] = i;  /* Start of icol-th column in hess */
        jcol = icol;
      }
    else if (icol > jcol+1)
      {
        /* Index in hess 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 khess to i.
         */
        for (j = jcol+1; j < icol; ++j)
          khess[j-1] = i;

        khess[icol-1] = i;
      }
  }
khess[ncolh] = nnz_hess;

/* Initialize options structure */
e04xxc(&options);
options.crnames = crnames;

/* Package up Hessian data for communication via comm */
hess_data.hess = hess;
hess_data.khess = khess;
hess_data.hhess = hhess;

comm.p = (Pointer)&hess_data;

/* Solve the problem */
```

```
  e04nkc(n, m, nnz, iobj, ncolh, qphess2, a, ha, ka, bl, bu,
         x, &ninf, &sinf, &obj, &options, &comm, &fail);

  if (fail.code == NE_NOERROR)
    {
      Vprintf("\nPerturb the problem and re-solve with warm start.\n");
      for (i = 0; i < nnz_hess; ++i)
        hess[i] *= 1.001;

      options.start = Nag_Warm;
      options.print_level = Nag_Soln;
      e04nkc(n, m, nnz, iobj, ncolh, qphess2, a, ha, ka, bl, bu,
             x, &ninf, &sinf, &obj, &options, &comm, &fail);
    }

  /* Free memory NAG-allocated to members of options */
  e04xzc(&options, "", NAGERR_DEFAULT);

  if (fail.code != NE_NOERROR)
    exit(EXIT_FAILURE);

}  /* ex2 */

#ifdef NAG_PROTO
static void qphess2(Integer ncolh,  double x[], double hx[], Nag_Comm *comm)
#else
     static void qphess2(ncolh, x, hx, comm)
     Integer ncolh;
     double x[], hx[];
     Nag_Comm *comm;
#endif
{
  Integer i, j, jrow;
  HessianData *hd = (HessianData *)(comm->p);
  double *hess = hd->hess;
  Integer *hhess = hd->hhess;
  Integer *khess = hd->khess;


  for (i = 0; i < ncolh; ++i)
    hx[i] = 0.0;

  for (i = 0; i < ncolh; ++i)
    {
      /* For each column of Hessian */
      for (j = khess[i]; j < khess[i+1]; ++j)
        {
          /* For each non-zero in column */

          jrow = hhess[j] - 1;

          /* Using symmetry of hessian, add contribution
           * to hx of hess[j] as a diagonal or upper
           * triangular element of hessian.
           */
          hx[i] += hess[j]*x[jrow];

          /* If hess[j] not a diagonal element add its
           * contribution to hx as a lower triangular
           * element of hessian.
           */
          if (jrow != i)
            hx[jrow] += hess[j]*x[i];
        }
    }

}  /* qphess2 */
```

13.2. Program Data

```
Data for example 2.

Values of n and m
7  8

Values of nnz, iobj and ncolh
48  8  7

Matrix nonzeros: value, row index, column index
    0.02   7   1
    0.02   5   1
    0.03   3   1
    1.00   1   1
    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

Lower bounds
 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

Upper bounds
 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

Column and row names
'COLUMN 1' 'COLUMN 2' 'COLUMN 3' 'COLUMN 4' 'COLUMN 5' 'COLUMN 6' 'COLUMN 7'
```

```
'OBJECTIV' 'ROW    1' 'ROW    2' 'ROW    3' 'ROW    4' 'ROW    5' 'ROW    6'
'ROW    7'

Initial estimate of x
 0.0  0.0  0.0  0.0  0.0  0.0  0.0

Number of hessian nonzeros
9

Hessian nonzeros: value, row index, col index (diagonal/lower triangle elements)
 2.0  1  1
 2.0  2  2
 2.0  3  3
 2.0  4  3
 2.0  4  4
 2.0  5  5
 2.0  6  6
 2.0  7  6
 2.0  7  7

Data for example 2.

Values of n and m
5  4

Values of nnz, iobj and ncolh
11  1  5

Matrix nonzeros: value, row index, column index
 1.0  2 1
-4.0  1 2
 3.0  2 2
 1.0  4 2
-4.0  1 3
 1.0  3 3
-2.0  1 4
 1.0  3 4
-2.0  1 5
-2.0  3 5
-1.0  4 5

Lower bounds
-1.0e+01  -1.0e+01  -1.0e+01  -1.0e+01  -1.0e+01
-1.0e+25   0.0       0.0       0.0

Upper bounds
 1.0e+01   1.0e+01   1.0e+01   1.0e+01   1.0e+01
 1.0e+25   0.0       0.0       0.0

Column and row names
'COLUMN 1' 'COLUMN 2' 'COLUMN 3' 'COLUMN 4' 'COLUMN 5'
'OBJECTIV' 'ROW    1' 'ROW    2' 'ROW    3'

Initial estimate of x
 2.0  2.0  2.0  2.0  2.0

Number of hessian nonzeros
7

Hessian nonzeros: value, row index, col index (diagonal/lower triangle elements)
 2.0  1  1
-2.0  2  1
 4.0  2  2
 2.0  3  2
 2.0  3  3
 2.0  4  4
 2.0  5  5
```

13.3. Program Results

```
Example 2: some optional parameters are set.

Parameters to e04nkc
--------------------

Problem type...........sparse QP    Number of variables.....        7
Linear constraints......        8   Hessian columns.........        7

prob_name...............
obj_name................            rhs_name................
range_name..............            bnd_name................
crnames................. supplied

minimize................     TRUE   start................... Nag_Cold
ftol.................... 1.00e-06   reset_ftol..............    10000
fcheck..................       60   factor_freq.............      100
scale.............Nag_ExtraScale   scale_tol............... 9.00e-01
optim_tol............... 1.00e-06   max_iter................       75
crash.............Nag_CrashTwice   crash_tol............... 1.00e-01
partial_price...........       10   pivot_tol............... 2.04e-11
max_sb..................        7
inf_bound............... 1.00e+20   inf_step................ 1.00e+20
lu_factor_tol........... 1.00e+02   lu_update_tol........... 1.00e+01
lu_sing_tol............. 2.04e-11   machine precision....... 1.11e-16
print_level.........Nag_Soln_Iter
outfile.................   stdout

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

   Itn      Step   Ninf   Sinf/Objective   Norm rg
Itn 0 -- Infeasible
     0   0.0e+00      1     1.152891e+03   0.0e+00
     1   4.3e+02      0     0.000000e+00   0.0e+00
Itn 1 -- Feasible point found (for 1 equality constraints).
     1   0.0e+00      0     0.000000e+00   0.0e+00
     1   0.0e+00      0     1.460000e+06   0.0e+00
Itn 1 -- Feasible QP solution.
     2   8.7e-02      0     9.409959e+05   0.0e+00
     3   5.3e-01      0    -1.056552e+06   0.0e+00
     4   1.0e+00      0    -1.462190e+06   2.3e-12
     5   1.0e+00      0    -1.698092e+06   2.2e-12
     6   4.6e-02      0    -1.764906e+06   7.0e+02
     7   1.0e+00      0    -1.811946e+06   2.8e-12
     8   1.7e-02      0    -1.847325e+06   1.7e+02
     9   1.0e+00      0    -1.847785e+06   7.0e-12

Variable State        Value   Lower Bound  Upper Bound   Lagr Mult    Residual
COLUMN 1    LL  0.00000e+00   0.0000e+00   2.0000e+02   2.361e+03   0.000e+00
COLUMN 2    BS  3.49399e+02   0.0000e+00   2.5000e+03  -3.657e-12   3.494e+02
COLUMN 3   SBS  6.48853e+02   4.0000e+02   8.0000e+02  -5.924e-12   1.511e+02
COLUMN 4   SBS  1.72847e+02   1.0000e+02   7.0000e+02   1.949e-12   7.285e+01
COLUMN 5    BS  4.07521e+02   0.0000e+00   1.5000e+03   0.000e+00   4.075e+02
COLUMN 6    BS  2.71356e+02   0.0000e+00         None  -3.280e-12   2.714e+02
COLUMN 7    BS  1.50023e+02   0.0000e+00         None  -1.413e-12   1.500e+02

Constrnt State        Value   Lower Bound  Upper Bound   Lagr Mult    Residual
OBJECTIV    EQ  2.00000e+03   2.0000e+03   2.0000e+03  -1.290e+04  -0.000e+00
ROW    1    BS  4.92316e+01         None   6.0000e+01   0.000e+00  -1.077e+01
ROW    2    UL  1.00000e+02         None   1.0000e+02  -2.325e+03   0.000e+00
ROW    3    BS  3.20719e+01         None   4.0000e+01   0.000e+00  -7.928e+00
ROW    4    BS  1.45572e+01         None   3.0000e+01   0.000e+00  -1.544e+01
ROW    5    LL  1.50000e+03   1.5000e+03         None   1.445e+04  -0.000e+00
ROW    6    LL  2.50000e+02   2.5000e+02   3.0000e+02   1.458e+04  -0.000e+00
ROW    7    BS -2.98869e+06         None         None  -1.000e+00  -2.989e+06

Exit after 9 iterations.
```

```
Optimal QP solution found.

Final QP objective value =  -1.8477847e+06


Perturb the problem and re-solve with warm start.

Parameters to e04nkc
--------------------

Problem type...........sparse QP    Number of variables.....        7
Linear constraints......        8    Hessian columns.........        7

prob_name...............
obj_name................             rhs_name................
range_name..............             bnd_name................
crnames................. supplied

minimize................     TRUE    start................... Nag_Warm
ftol.................... 1.00e-06    reset_ftol..............    10000
fcheck..................       60    factor_freq.............      100
scale..............Nag_ExtraScale    scale_tol............... 9.00e-01
optim_tol............... 1.00e-06    max_iter................       75
crash..............Nag_CrashTwice    crash_tol............... 1.00e-01
partial_price...........       10    pivot_tol............... 2.04e-11
max_sb..................        7
inf_bound............... 1.00e+20    inf_step................ 1.00e+20
lu_factor_tol........... 1.00e+02    lu_update_tol........... 1.00e+01
lu_sing_tol............. 2.04e-11    machine precision....... 1.11e-16
print_level............. Nag_Soln
outfile.................   stdout

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

Variable State        Value  Lower Bound  Upper Bound   Lagr Mult    Residual
COLUMN 1    LL  0.00000e+00  0.0000e+00  2.0000e+02   2.360e+03   0.000e+00
COLUMN 2   SBS  3.49529e+02  0.0000e+00  2.5000e+03  -7.077e-13   3.495e+02
COLUMN 3    BS  6.48762e+02  4.0000e+02  8.0000e+02  -1.338e-12   1.512e+02
COLUMN 4   SBS  1.72618e+02  1.0000e+02  7.0000e+02   0.000e+00   7.262e+01
COLUMN 5    BS  4.07596e+02  0.0000e+00  1.5000e+03   6.891e-13   4.076e+02
COLUMN 6    BS  2.71446e+02  0.0000e+00        None   2.087e-12   2.714e+02
COLUMN 7    BS  1.50048e+02  0.0000e+00        None   7.850e-13   1.500e+02

Constrnt State        Value  Lower Bound  Upper Bound   Lagr Mult    Residual
OBJECTIV    EQ  2.00000e+03  2.0000e+03  2.0000e+03  -1.290e+04  -0.000e+00
ROW     1   BS  4.92290e+01        None  6.0000e+01   0.000e+00  -1.077e+01
ROW     2   UL  1.00000e+02        None  1.0000e+02  -2.325e+03   0.000e+00
ROW     3   BS  3.20731e+01        None  4.0000e+01   0.000e+00  -7.927e+00
ROW     4   BS  1.45618e+01        None  3.0000e+01   0.000e+00  -1.544e+01
ROW     5   LL  1.50000e+03  1.5000e+03        None   1.446e+04  -0.000e+00
ROW     6   LL  2.50000e+02  2.5000e+02  3.0000e+02   1.458e+04  -0.000e+00
ROW     7   BS -2.98841e+06        None        None  -1.000e+00  -2.988e+06

Exit after 1 iterations.

Optimal QP solution found.

Final QP objective value =  -1.8466439e+06
```