NAG Library Function Document

nag_mip_sqp (h02dac)

1 Purpose

nag_mip_sqp (h02dac) solves general nonlinear programming problems with integer constraints on some of the variables.

2 Specification

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

void nag_mip_sqp (Integer n, Integer nclin, Integer ncnln, const double a[],
     Integer pda, const double d[], double ax[], const double bl[],
     const double bu[], const Integer varcon[], double x[],

     void (*confun)(Integer *mode, Integer ncnln, Integer n,
          const Integer varcon[], const double x[], double c[],
          double cjac[], Integer nstate, Nag_Comm *comm),

     double c[], double cjac[],

     void (*objfun)(Integer *mode, Integer n, const Integer varcon[],
          const double x[], double *objmip, double objgrd[], Integer nstate,
          Nag_Comm *comm),

     double objgrd[], Integer maxit, double acc, double *objmip,
     const Integer iopts[], const double opts[], Nag_Comm *comm,
     NagError *fail)
```

Before calling nag_mip_sqp (h02dac), nag_mip_opt_set (h02zkc) **must** be called with **optstr** set to '`Initialize = h02dac`'. Optional parameters may also be specified by calling nag_mip_opt_set (h02zkc) before the call to nag_mip_sqp (h02dac).

3 Description

nag_mip_sqp (h02dac) solves mixed integer nonlinear programming problems using a modified sequential quadratic programming method. The problem is assumed to be stated in the following general form:

$$\begin{array}{lll}\underset{x\in\{R^{n_c},Z^{n_i}\}}{\text{minimize}} & f(x) & \\ \text{subject to} & c_j(x)=0, & j=1,2,\dots,m_e \\ & c_j(x)\geq 0, & j=m_e+1,m_e+2,\dots,m \\ & l\leq x_i\leq u, & i=1,2,\dots,n\end{array}$$

with n_c continuous variables and n_i binary and integer variables in a total of n variables; m_e equality constraints in a total of m constraint functions.

Partial derivatives of $f(x)$ and $c(x)$ are not required for the n_i integer variables. Gradients with respect to integer variables are approximated by difference formulae.

No assumptions are made regarding $f(x)$ except that it is twice continuously differentiable with respect to continuous elements of x. It is not assumed that integer variables are relaxable. In other words, problem functions are evaluated only at integer points.

The method seeks to minimize the exact penalty function:

$$P_\sigma(x)=f(x)+\sigma\|g(x)\|_\infty$$

where σ is adapted by the algorithm and $g(x)$ is given by:

$$\begin{array}{rcll} g(x) & = & c_j(x), & j = 1, 2, \ldots, m_e \\ & = & \min\bigl(c_j(x), 0\bigr), & j = m_e + 1, m_e + 2, \ldots, m. \end{array}$$

Successive quadratic approximations are applied under the assumption that integer variables have a smooth influence on the model functions, that is function values do not change drastically when incrementing or decrementing an integer value. In practice this requires integer variables to be ordinal not categorical. The algorithm is stabilised by a trust region method including Yuan's second order corrections, see Yuan and Sun (2006). The Hessian of the Lagrangian function is approximated by BFGS (see Section 11.4 in nag_opt_nlp (e04ucc)) updates subject to the continuous and integer variables.

The mixed-integer quadratic programming subproblems of the SQP-trust region method are solved by a branch and cut method with continuous subproblem solutions obtained by the primal-dual method of Goldfarb and Idnani, see Powell (1983). Different strategies are available for selecting a branching variable:

Maximal fractional branching. Select an integer variable from the relaxed subproblem solution with largest distance from next integer value

Minimal fractional branching. Select an integer variable from the relaxed subproblem solution with smallest distance from next integer value

and a node from where branching, that is the generation of two new subproblems, begins:

Best of two. The optimal objective function values of the two child nodes are compared and the node with a lower value is chosen

Best of all. Select an integer variable from the relaxed subproblem solution with the smallest distance from the next integer value

Depth first. Select a child node whenever possible.

This implementation is based on the algorithm MISQP as described in Exler *et al.* (2013).

Linear constraints may optionally be supplied by a matrix A and vector d rather than the constraint functions $c(x)$ such that

$$Ax = d \quad \text{or} \quad Ax \geq d.$$

Partial derivatives with respect to x of these constraint functions are not requested by nag_mip_sqp (h02dac).

4 References

Exler O, Lehmann T and Schittkowski K (2013) A comparative study of SQP-type algorithms for nonlinear and nonconvex mixed-integer optimization *Mathematical Programming Computation* **4** 383–412

Mann A (1986) GAMS/MINOS: Three examples *Department of Operations Research Technical Report* Stanford University

Powell M J D (1983) On the quadratic programming algorithm of Goldfarb and Idnani *Report DAMTP 1983/Na 19* University of Cambridge, Cambridge

Yuan Y-x and Sun W (2006) *Optimization Theory and Methods* Springer–Verlag

5 Arguments

1: **n** – Integer *Input*

On entry: n, the total number of variables, $n_c + n_i$.

Constraint: $\mathbf{n} > 0$.

2: **nclin** – Integer *Input*

On entry: n_l, the number of general linear constraints defined by A and d.

Constraint: **nclin** ≥ 0.

3: **ncnln** – Integer *Input*

On entry: n_N, the number of constraints supplied by $c(x)$.

Constraint: **ncnln** ≥ 0.

4: **a**[*dim*] – const double *Input*

Note: the dimension, *dim*, of the array **a** must be at least **n** when **nclin** > 0.

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

On entry: the ith row of **a** must contain the coefficients of the ith general linear constraint, for $i = 1, 2, \ldots, n_l$. Any equality constraints must be specified first.

If **nclin** $= 0$, the array **a** is not referenced and may be **NULL**.

5: **pda** – Integer *Input*

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

Constraint: **pda** $\geq$ **nclin**.

6: **d**[**nclin**] – const double *Input*

On entry: d_i, the constant for the ith linear constraint.

If **nclin** $= 0$, the array **d** is not referenced and may be **NULL**.

7: **ax**[**nclin**] – double *Output*

On exit: the final values of the linear constraints Ax.

If **nclin** $= 0$, **ax** is not referenced and may be **NULL**.

8: **bl**[**n**] – const double *Input*
9: **bu**[**n**] – const double *Input*

On entry: **bl** must contain the lower bounds, l_i, and **bu** the upper bounds, u_i, for the variables; bounds on integer variables are rounded, bounds on binary variables need not be supplied.

Constraint: **bl**$[i-1] \leq$ **bu**$[i-1]$, for $i = 1, 2, \ldots,$ **n**.

10: **varcon**[**n** + **nclin** + **ncnln**] – const Integer *Input*

On entry: **varcon** indicates the nature of each variable and constraint in the problem. The first n elements of the array must describe the nature of the variables, the next n_L elements the nature of the general linear constraints (if any) and the next n_N elements the general constraints (if any).

varcon$[j-1] = 0$
A continuous variable.

varcon$[j-1] = 1$
A binary variable.

varcon$[j-1] = 2$
An integer variable.

varcon$[j-1] = 3$
An equality constraint.

varcon$[j-1] = 4$
An inequality constraint.

Constraints:

varcon$[j-1] = 0$, 1 or 2, for $j = 1, 2, \ldots, $**n**;
varcon$[j-1] = 3$ or 4, for $j = $ **n** $+ 1, \ldots,$ **n** $+$ **nclin** $+$ **ncnln**;
At least one variable must be either binary or integer;
Any equality constraints must precede any inequality constraints.

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

On entry: an initial estimate of the solution, which need not be feasible. Values corresponding to integer variables are rounded; if an initial value less than half is supplied for a binary variable the value zero is used, otherwise the value one is used.

On exit: the final estimate of the solution.

12: **confun** – function, supplied by the user *External Function*

confun must calculate the constraint functions supplied by $c(x)$ and their Jacobian at x. If all constraints are supplied by A and d (i.e., **ncnln** $= 0$), **confun** will never be called by nag_mip_sqp (h02dac) and the NAG defined null void function pointer, **NULLFN**, may be supplied in the call instead. If **ncnln** > 0, the first call to **confun** will occur after the first call to **objfun**.

The specification of **confun** is:

```
void confun (Integer *mode, Integer ncnln, Integer n,
     const Integer varcon[], const double x[], double c[],
     double cjac[], Integer nstate, Nag_Comm *comm)
```

1: **mode** – Integer * *Input/Output*

On entry: indicates which values must be assigned during each call of **objfun**. Only the following values need be assigned:

mode $= 0$
Elements of **c** containing continuous variables.

mode $= 1$
Elements of **cjac** containing continuous variables.

On exit: may be set to a negative value if you wish to terminate the solution to the current problem, and in this case nag_mip_sqp (h02dac) will terminate with **fail** set to **mode**.

2: **ncnln** – Integer *Input*

On entry: the dimension of the array **c** and the first dimension of the array **cjac**. The number of constraints supplied by $c(x)$, n_N.

3: **n** – Integer *Input*

On entry: the second dimension of the array **cjac**. n, the total number of variables, $n_c + n_i$.

4: **varcon**[**n** $+$ **nclin** $+$ **ncnln**] – const Integer *Input*

Note: the dimension, *dim*, of the array **varcon** is **n** $+$ **nclin** $+$ **ncnln**.

On entry: the array **varcon** as supplied to nag_mip_sqp (h02dac).

5: **x**[**n**] – const double *Input*

On entry: the vector of variables at which the objective function and/or all continuous elements of its gradient are to be evaluated.

6: **c**[**ncnln**] – double *Output*

On exit: must contain **ncnln** constraint values, with the value of the jth constraint $c_j(x)$ in **c**[$j - 1$].

7: **cjac**[**ncnln** × **n**] – double *Input/Output*

Note: the derivative of the ith constraint with respect to the jth variable, $\frac{\partial c_i}{\partial x_{\mathrm{j}}}$, is stored in **cjac**[$(j - 1) \times$ **ncnln** $+ i - 1$].

On entry: continuous elements of **cjac** are set to the value of NaN.

On exit: the ith row of **cjac** must contain elements of $\frac{\partial c_i}{\partial x_{\mathrm{j}}}$ for each continuous variable x_j.

8: **nstate** – Integer *Input*

On entry: if **nstate** $= 1$, nag_mip_sqp (h02dac) is calling **confun** for the first time. This argument setting allows you to save computation time if certain data must be read or calculated only once.

9: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_mip_sqp (h02dac) you may allocate memory and initialize these pointers with various quantities for use by **confun** when called from nag_mip_sqp (h02dac) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

13: **c**[**ncnln**] – double *Output*

On exit: if **ncnln** > 0, **c**[$j - 1$] contains the value of the jth constraint function $c_j(x)$ at the final iterate, for $j = 1, 2, \ldots,$ **ncnln**.

If **ncnln** $= 0$, the array **c** is not referenced and may be **NULL**.

14: **cjac**[**ncnln** × **n**] – double *Output*

Note: the derivative of the ith constraint with respect to the jth variable, $\frac{\partial c_i}{\partial x_{\mathrm{j}}}$, is stored in **cjac**[$(j - 1) \times$ **ncnln** $+ i - 1$].

On exit: if **ncnln** > 0, **cjac** contains the Jacobian matrix of the constraint functions at the final iterate, i.e., **cjac**[$(j - 1) \times$ **ncnln** $+ i - 1$] contains the partial derivative of the ith constraint function with respect to the jth variable, for $i = 1, 2, \ldots,$ **ncnln** and $j = 1, 2, \ldots,$ **n**. (See the discussion of argument **cjac** under **confun**.)

If **ncnln** $= 0$, the array **cjac** is not referenced and may be **NULL**.

15: **objfun** – function, supplied by the user *External Function*

objfun must calculate the objective function $f(x)$ and its gradient for a specified n-element vector x.

The specification of **objfun** is:

```
void objfun (Integer *mode, Integer n, const Integer varcon[],
     const double x[], double *objmip, double objgrd[],
     Integer nstate, Nag_Comm *comm)
```

1: **mode** – Integer * *Input/Output*

On entry: indicates which values must be assigned during each call of **objfun**. Only the following values need be assigned:

mode $= 0$
The objective function value, **objmip**.

mode $= 1$
The continuous elements of **objgrd**.

On exit: may be set to a negative value if you wish to terminate the solution to the current problem, and in this case nag_mip_sqp (h02dac) will terminate with **fail** set to **mode**.

2: **n** – Integer *Input*

On entry: n, the total number of variables, $n_c + n_i$.

3: **varcon**[**n** + **nclin** + **ncnln**] – const Integer *Input*

Note: the dimension, *dim*, of the array **varcon** is **n** + **nclin** + **ncnln**.

On entry: the array **varcon** as supplied to nag_mip_sqp (h02dac).

4: **x**[**n**] – const double *Input*

On entry: the vector of variables at which the objective function and/or all continuous elements of its gradient are to be evaluated.

5: **objmip** – double * *Output*

On exit: must be set to the objective function value, f, if **mode** $= 0$; otherwise **objmip** is not referenced.

6: **objgrd**[**n**] – double *Input/Output*

On entry: continuous elements of **objgrd** are set to the value of NaN.

On exit: must contain the gradient vector of the objective function if **mode** $= 1$, with **objgrd**$[j-1]$ containing the partial derivative of f with respect to continuous variable x_j; otherwise **objgrd** is not referenced.

7: **nstate** – Integer *Input*

On entry: if **nstate** $= 1$, nag_mip_sqp (h02dac) is calling **objfun** for the first time. This argument setting allows you to save computation time if certain data must be read or calculated only once.

8: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_mip_sqp (h02dac) you may allocate memory and initialize these pointers with various quantities for use by **objfun** when called from nag_mip_sqp (h02dac) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

16: **objgrd**[**n**] – double *Output*

On exit: the objective function gradient at the solution.

17: **maxit** – Integer *Input*

On entry: the maximum number of iterations within which to find a solution. If **maxit** is less than or equal to zero, the suggested value below is used.

Suggested value: **maxit** $= 500$.

18: **acc** – double *Input*

On entry: the requested accuracy for determining feasible points during iterations and for halting the method when the predicted improvement in objective function is less than **acc**. If **acc** is less than or equal to ϵ (ϵ being the ***machine precision*** as given by nag_machine_precision (X02AJC)), the below suggested value is used.

Suggested value: **acc** $= 0.0001$.

19: **objmip** – double * *Output*

On exit: with **fail.code** = NE_NOERROR, **objmip** contains the value of the objective function for the MINLP solution.

20: **iopts**[*dim*] – const Integer *Communication Array*

Note: the dimension, *dim*, of this array is dictated by the requirements of associated functions that must have been previously called. This array MUST be the same array passed as argument **iopts** in the previous call to nag_mip_opt_set (h02zkc).

21: **opts**[*dim*] – const double *Communication Array*

Note: the dimension, *dim*, of this array is dictated by the requirements of associated functions that must have been previously called. This array MUST be the same array passed as argument **opts** in the previous call to nag_mip_opt_set (h02zkc).

On entry: the real option array as returned by nag_mip_opt_set (h02zkc).

22: **comm** – Nag_Comm *

The NAG communication argument (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

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

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 3.2.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

On entry, argument $\langle value \rangle$ had an illegal value.

NE_BOUND

On entry, **bl**$[\langle value \rangle] >$ **bu**$[\langle value \rangle]$.
Constraint: **bl**$[i-1] \leq$ **bu**$[i-1]$, for $i = 1, 2, \ldots,$ **n**.

NE_CONSTRAINT

On entry, linear equality constraints do not precede linear inequality constraints.

On entry, nonlinear equality constraints do not precede nonlinear inequality constraints.

NE_DERIV_ERRORS

One or more constraint gradients appear to be incorrect.

One or more objective gradients appear to be incorrect.

NE_INFEASIBLE

Termination at an infeasible iterate; if the problem is feasible, try a different starting value.

NE_INFINITE

Penalty parameter tends to infinity in an underlying mixed-integer quadratic program; the problem may be infeasible. If σ is relatively low value, try a higher one, for example 10^{20}.
Optional parameter **Penalty** $= \langle value \rangle$.

NE_INITIALIZATION

On entry, the optional parameter arrays **iopts** and **opts** have either not been initialized or been corrupted.

NE_INT

On entry, **n** $= \langle value \rangle$.
Constraint: **n** > 0.

On entry, **nclin** $= \langle value \rangle$.
Constraint: **nclin** ≥ 0.

On entry, **ncnln** $= \langle value \rangle$.
Constraint: **ncnln** ≥ 0.

NE_INT_3

On entry, **pda** $= \langle value \rangle$ and **nclin** $= \langle value \rangle$.
Constraint: **pda** $\geq$ **nclin**.

NE_INT_ARRAY_CONS

On entry, **varcon**$[\langle value \rangle] = \langle value \rangle$.
Constraint: **varcon**$[i-1] = 0$, 1 or 2, for $i = 1, 2, \ldots,$ **n**.

On entry, **varcon**[⟨*value*⟩] = ⟨*value*⟩.
Constraint: **varcon**$[i-1] = 3$ or 4, for $i =$ **n** $+ 1, \ldots,$ **n** + **nclin** + **ncnln**.

NE_INTERNAL_ERROR

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

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in How to Use the NAG Library and its Documentation for further information.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in How to Use the NAG Library and its Documentation for further information.

NE_NUM_DIFFICULTIES

The optimization failed due to numerical difficulties. Set optional parameter **Print Level** = 3 for more information.

NE_TOO_MANY

More than the maximum number of feasible steps without improvement in the objective function. If the maximum number of feasible steps is small, say less than 5, try increasing it. Optional parameter **Feasible Steps** = ⟨*value*⟩.

NE_USER_NAN

The supplied **confun** returned a NaN value.

The supplied **objfun** returned a NaN value.

NE_USER_STOP

The optimization halted because you set **mode** negative in **objfun** or **mode** negative in **confun**, to ⟨*value*⟩.

NE_ZERO_COEFF

Termination with zero integer trust region for integer variables; try a different starting value. Optional parameter **Integer Trust Radius** = ⟨*value*⟩.

NE_ZERO_VARS

On entry, there are no binary or integer variables.

NW_TOO_MANY_ITER

On entry, **maxit** = ⟨*value*⟩. Exceeded the maximum number of iterations.

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_mip_sqp (h02dac) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the x06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Notefor your implementation for any additional implementation-specific information.

9 Further Comments

None.

10 Example

Select a portfolio of at most p assets from n available with expected return ρ, is fully invested and that minimizes

$$\begin{array}{lll} & x^{\mathrm{T}}\Sigma x & & \\ \text{subject to} & r^{\mathrm{T}}x & = & \rho \\ & \displaystyle\sum_{i=1}^{n} x_i & = & 1 \\ & x_i & \leq & y_i \\ & \displaystyle\sum_{i=1}^{n} y_i & \leq & p \\ & x_i & \geq & 0 \\ & y_i & = & 0 \quad \text{or} \quad 1 \end{array}$$

where

x is a vector of proportions of selected assets

y is an indicator variable that describes if an asset is in or out

r is a vector of mean returns

Σ is the covariance matrix of returns.

This example is taken from Mann (1986) with

$$\begin{array}{lll} r & = & \begin{pmatrix} 8 & 9 & 12 & 7 \end{pmatrix} \\ \Sigma & = & \begin{pmatrix} 4 & 3 & -1 & 0 \\ 3 & 6 & 1 & 0 \\ -1 & 1 & 10 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \\ p & = & 3 \\ \rho & = & 10. \end{array}$$

Linear constraints are supplied through both A and d, and **confun**.

10.1 Program Text

```
/* nag_mip_sqp (h02dac) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */
#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagh02.h>

#ifdef __cplusplus
extern "C"
{
#endif
  static void NAG_CALL confun(Integer *mode, Integer ncnln, Integer n,
                              const Integer varcon[], const double x[],
                              double c[], double cjac[], Integer nstate,
                              Nag_Comm *comm);
  static void NAG_CALL objfun(Integer *mode, Integer n,
                              const Integer varcon[], const double x[],
                              double *objmip, double objgrd[], Integer nstate,
                              Nag_Comm *comm);
#ifdef __cplusplus
}
#endif

#define CJAC(I, J) cjac[(J-1)*ncnln+I-1]
#define A(I, J)    a[(J-1)*pda+I-1]

int main(void)
{
  /* Integer scalar and array declarations */
  const Integer liopts = 200, lopts = 100, lcvalue = 40;
  Integer i, j, pda, maxit, n, nclin, ncnln, exit_status = 0;
  Integer iopts[200], p, *varcon = 0, ivalue;

  /* NAG structures and types */
  Nag_VariableType optype;
  NagError fail;
  Nag_Comm comm;

  /* Double scalar and array declarations */
  double acc, accqp, objmip;
  double *a = 0, *ax = 0, *bl = 0, *bu = 0, *c = 0, *cjac = 0;
  double *d = 0, *objgrd = 0, *x = 0, opts[200], rho;
  static double ruser[2] = { -1.0, -1.0 };

  /* Character declarations */
  char cvalue[40];

  /* Initialize the error structure */
  INIT_FAIL(fail);

  printf("nag_mip_sqp (h02dac) Example Program Results\n\n");

  n = 8;
  nclin = 5;
  ncnln = 2;

  pda = nclin;

  if (!(a = NAG_ALLOC(n * pda, double)) ||
      !(d = NAG_ALLOC(nclin, double)) ||
      !(ax = NAG_ALLOC(nclin, double)) ||
      !(bl = NAG_ALLOC(n, double)) ||
      !(bu = NAG_ALLOC(n, double)) ||
      !(varcon = NAG_ALLOC(n + nclin + ncnln, Integer)) ||
      !(x = NAG_ALLOC(n, double)) ||
      !(c = NAG_ALLOC(ncnln, double)) ||
      !(cjac = NAG_ALLOC(ncnln * n, double)) ||
      !(objgrd = NAG_ALLOC(n, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

  for (i = 0; i < 4; i++) {
    /* Set variable types: continuous then binary */
    varcon[i] = 0;
    varcon[4 + i] = 1;

    /* Set continuous variable bounds */
    bl[i] = 0.0;
    bu[i] = 1.0e3;
  }

  /* Bounds of binary variables need not be provided */
  for (i = 4; i < 8; i++) {
    bl[i] = 0.0;
    bu[i] = 1.0;
  }

  /* Set linear constraint, equality first */
  varcon[n] = 3;
  varcon[n + 1] = varcon[n + 2] = varcon[n + 3] = varcon[n + 4] = 4;

  /* Set Ax=d then Ax>=d */
  for (i = 1; i <= nclin; i++) {
    for (j = 1; j <= n; j++) {
      A(i, j) = 0.0;
    }
  }
  A(1, 1) = A(1, 2) = A(1, 3) = A(1, 4) = 1.0;
  A(2, 1) = -1.0;
  A(2, 5) = 1.0;
  A(3, 2) = -1.0;
  A(3, 6) = 1.0;
  A(4, 3) = -1.0;
  A(4, 7) = 1.0;
  A(5, 4) = -1.0;
  A(5, 8) = 1.0;
  d[0] = 1.0;
  d[1] = d[2] = d[3] = d[4] = 0.0;

  /* Set constraints supplied by CONFUN, equality first */
  varcon[n + nclin] = 3;
  varcon[n + nclin + 1] = 4;

  /* Initialize communication arrays */
  nag_mip_opt_set("Initialize = nag_mip_sqp", iopts, liopts, opts, lopts,
                  &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_mip_opt_set (h02zkc).\n%s\n", fail.message);
    exit_status = -1;
    goto END;
  }

  /* Optimisiation parameters */
  maxit = 500;
  acc = 1.0e-6;

  /* Initial estimate (binary variables need not be given) */
  x[0] = x[1] = x[2] = x[3] = 1.0;
  x[4] = x[5] = x[6] = x[7] = 0.0;

  /* Portfolio parameters */
  p = 3;
  rho = 10.0;
  comm.iuser = &p;
  ruser[0] = rho;
  comm.user = ruser;

  /* Call MINLP solver h02dac (nag_mip_sqp) */
  nag_mip_sqp(n, nclin, ncnln, a, pda, d, ax, bl, bu, varcon, x, confun, c,
              cjac, objfun, objgrd, maxit, acc, &objmip, iopts, opts, &comm,
              &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_mip_sqp (h02dac).\n%s\n", fail.message);
    exit_status = -1;
    goto END;
  }

  /* Query the accuracy of the mixed integer QP solver */
  nag_mip_opt_get("QP Accuracy", &ivalue, &accqp, cvalue, lcvalue, &optype,
                  iopts, opts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_mip_opt_get (h02zlc).\n%s\n", fail.message);
    exit_status = -1;
    goto END;
  }
  /* Results */
  printf("\nFinal estimate:");
  for (i = 0; i < n; i++) {
    printf("\nx[%4" NAG_IFMT "] = %12.4f", i + 1, x[i]);
  }
  printf("\n\nRequested accuracy of QP subproblems = %12.4g\n", accqp);
  printf("\nOptimised value = %12.4g\n", objmip);

END:
  NAG_FREE(a);
  NAG_FREE(d);
  NAG_FREE(ax);
  NAG_FREE(bl);
  NAG_FREE(bu);
  NAG_FREE(varcon);
  NAG_FREE(x);
  NAG_FREE(c);
  NAG_FREE(cjac);
  NAG_FREE(objgrd);

  return (exit_status);
}

static void NAG_CALL confun(Integer *mode, Integer ncnln, Integer n,
                            const Integer varcon[], const double x[],
                            double c[], double cjac[], Integer nstate,
                            Nag_Comm *comm)
{
  Integer p;
  double rho;

  /* This problem has two nonlinear constraints.
   * As an example of using the mode mechanism,
   * terminate if any other size is supplied.
   */
  if (ncnln != 2) {
    *mode = -1;
    return;
  }

  if (nstate == 1)
    printf("\n(confun was just called for the first time)\n");

  if (*mode == 0) {
    /* Constraints */
    /* Mean return at least rho: */
    rho = comm->user[0];
    c[0] = 8.0 * x[0] + 9.0 * x[1] + 12.0 * x[2] + 7.0 * x[3] - rho;
    /* Maximum of p assets in portfolio: */
    p = *(comm->iuser);
    c[1] = (double) p - x[4] - x[5] - x[6] - x[7];
  }
  else {
    /* Jacobian */
    /* c[0] */
    CJAC(1, 1) = 8.0;
    CJAC(1, 2) = 9.0;
    CJAC(1, 3) = 12.0;
    CJAC(1, 4) = 7.0;
    /* c[1] does not include continuous variables which requires
       that their derivatives are zero */
    CJAC(2, 1) = CJAC(2, 2) = CJAC(2, 3) = CJAC(2, 4) = 0.0;
  }
}

static void NAG_CALL objfun(Integer *mode, Integer n, const Integer varcon[],
                            const double x[], double *objmip, double objgrd[],
                            Integer nstate, Nag_Comm *comm)
{
  /* This is an 8-dimensional problem.
   * As an example of using the mode mechanism,
   * terminate if any other size is supplied.
   */
  if (n != 8) {
    *mode = -1;
    return;
  }

  if (nstate == 1 || comm->user[1] == -1.0) {
    printf("\n(objfun was just called for the first time)\n");
    comm->user[1] = 0.0;
  }

  if (*mode == 0) {
    /* Objective value */
    *objmip =
           x[0] * (4.0 * x[0] + 3.0 * x[1] - x[2]) + x[1] * (3.0 * x[0] +
                                                             6.0 * x[1] +
                                                             x[2]) +
           x[2] * (x[1] - x[0] + 10.0 * x[2]);
  }
  else {
    /* Objective gradients for continuous variables */
    objgrd[0] = 8.0 * x[0] + 6.0 * x[1] - 2.0 * x[2];
    objgrd[1] = 6.0 * x[0] + 12.0 * x[1] + 2.0 * x[2];
    objgrd[2] = 2.0 * (x[1] - x[0]) + 20.0 * x[2];
    objgrd[3] = 0.0;
  }
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_mip_sqp (h02dac) Example Program Results


(objfun was just called for the first time)

(confun was just called for the first time)

Final estimate:
x[   1] =      0.3750
x[   2] =      0.0000
x[   3] =      0.5250
x[   4] =      0.1000
x[   5] =      1.0000
x[   6] =      0.0000
x[   7] =      1.0000
x[   8] =      1.0000

Requested accuracy of QP subproblems =        1e-10

Optimised value =        2.925
```

11 Optional Parameters

This section can be skipped if you wish to use the default values for all optional parameters, otherwise, the following is a list of the optional parameters available and a full description of each optional parameter is provided in Section 11.1.

Branch Bound Steps

Branching Rule

Check Gradients

Continuous Trust Radius

Descent
Descent Factor
Feasible Steps
Improved Bounds
Integer Trust Radius
Maximum Restarts
Minor Print Level
Modify Hessian
Node Selection
Non Monotone
Objective Scale Bound
Penalty
Penalty Factor
Print Level
QP Accuracy
Scale Continuous Variables
Scale Objective Function
Warm Starts

11.1 Description of the Optional Parameters

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

The summary line contains:

the keywords;

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

All options accept the value DEFAULT in order to return single options to their default states.

Keywords and character values are case insensitive, however they must be separated by at least one space.

nag_mip_opt_set (h02zkc) can be called to supply options, one call being necessary for each optional parameter. For example,

```
Call H02ZKF('Check Gradients = Yes', iopts, liopts, opts, lopts, ifail)
```

nag_mip_opt_set (h02zkc) should be consulted for a full description of the method of supplying optional parameters.

For nag_mip_sqp (h02dac) the maximum length of the argument **cvalue** used by nag_mip_opt_get (h02zlc) is 12.

Branch Bound Steps i Default $= 500$

Maximum number of branch-and-bound steps for solving the mixed integer quadratic problems.

Constraint: **Branch Bound Steps** > 1.

Branching Rule a Default $=$ Maximum

Branching rule for branch and bound search.

Branching Rule $=$ Maximum
Maximum fractional branching.

Branching Rule $=$ Minimum
Minimum fractional branching.

Check Gradients a Default = No

Perform an internal check of supplied objective and constraint gradients. It is advisable to set **Check Gradients** = Yes during code development to avoid difficulties associated with incorrect user-supplied gradients.

Continuous Trust Radius r Default = 10.0

Initial continuous trust region radius, Δ_0^c; the initial trial step $d \in R^{n_c}$ for the SQP approximation must satisfy $\|d\|_\infty \leq \Delta_0^c$.

Constraint: **Continuous Trust Radius** > 0.0.

Descent r Default = 0.05

Initial descent parameter, δ, larger values of δ allow penalty optional parameter σ to increase faster which can lead to faster convergence.

Constraint: $0.0 <$ **Descent** < 1.0.

Descent Factor r Default = 0.1

Factor for decreasing the internal descent parameter, δ, between iterations.

Constraint: $0.0 <$ **Descent Factor** < 1.0.

Feasible Steps i Default = 10

Maximum number of feasible steps without improvements, where feasibility is measured by $\|g(x)\|_\infty \leq \sqrt{\mathbf{acc}}$.

Constraint: **Feasible Steps** > 1.

Improved Bounds a Default = No

Calculate improved bounds in case of 'Best of all' node selection strategy.

Integer Trust Radius r Default = 10.0

Initial integer trust region radius, Δ_0^i; the initial trial step $e \in R^{n_i}$ for the SQP approximation must satisfy $\|e\|_\infty \leq \Delta_0^i$.

Constraint: **Integer Trust Radius** ≥ 1.0.

Maximum Restarts i Default = 2

Maximum number of restarts that allow the mixed integer SQP algorithm to return to a better solution. Setting a value higher than the default might lead to better results at the expense of function evaluations.

Constraint: $0 <$ **Maximum Restarts** ≤ 15.

Minor Print Level i Default = 0

Print level of the subproblem solver. Active only if **Print Level** $\neq 0$.

Constraint: $0 <$ **Minor Print Level** < 4.

Modify Hessian a Default = Yes

Modify the Hessian approximation in an attempt to get more accurate search directions. Calculation time is increased when the number of integer variables is large.

Node Selection a Default = Depth First

Node selection strategy for branch and bound.

Node Selection = Best of all
Large tree search; this method is the slowest as it solves all subproblem QPs independently.

Node Selection = Best of two
Uses warm starts and less memory.

Node Selection = Depth first
Uses more warm starts. If warm starts are applied, they can speed up the solution of mixed integer subproblems significantly when solving almost identical QPs.

Non Monotone i Default = 10

Maximum number of successive iterations considered for the non-monotone trust region algorithm, allowing the penalty function to increase.

Constraint: $0 <$ **Non Monotone** < 100.

Objective Scale Bound r Default = 1.0

When **Scale Objective Function** > 0 internally scale absolute function values greater than 1.0 or **Objective Scale Bound**.

Constraint: **Objective Scale Bound** > 0.0.

Penalty r Default = 1000.0

Initial penalty optional parameter, σ.

Constraint: **Penalty** ≥ 0.0.

Penalty Factor r Default = 10.0

Factor for increasing penalty optional parameter σ when the trust regions cannot be enlarged at a trial step.

Constraint: **Penalty Factor** > 1.0.

Print Level i Default = 0

Specifies the desired output level of printing.

Print Level = 0
No output.

Print Level = 1
Final convergence analysis.

Print Level = 2
One line of intermediate results per iteration.

Print Level = 3
Detailed information printed per iteration.

QP Accuracy r Default = 1.0e−10

Termination tolerance of the relaxed quadratic program subproblems.

Constraint: **QP Accuracy** > 0.0.

Scale Continuous Variables a Default = Yes

Internally scale continuous variables values.

Scale Objective Function i Default $= 1$

Internally scale objective function values.

Scale Objective Function $= 0$
No scaling.

Scale Objective Function $= 1$
Scale absolute values greater than **Objective Scale Bound**.

Warm Starts i Default $= 100$

Maximum number of warm starts within the mixed integer QP solver, see **Node Selection**.

Constraint: **Warm Starts** ≥ 0.