nag_ip_bb (h02bbc)

1. Purpose

nag_ip_bb (h02bbc) solves 'zero-one', 'general', 'mixed' or 'all' integer linear and quadratic programming problems using a branch and bound method. The function may also be used to find either the first integer solution or the optimum integer solution. It is not intended for large sparse problems.

2. Specification

```
#include <nag.h>
#include <nagh02.h>

void nag_ip_bb(Integer n, Integer m, double a[], Integer tda,
               double bl[], double bu[], Boolean intvar[],
               double cvec[], double h[], Integer tdh,
               void (*qphess)(Integer n, Integer jthcol, double h[],
                              Integer tdh, double x[], double hx[],
                              Nag_Comm *comm),
               double x[], double *obj, Nag_H02_Opt *options,
               Nag_Comm *comm, NagError *fail)
```

3. Description

nag_ip_bb is capable of solving certain types of integer programming (IP) problems using a branch and bound (BB) method, see Taha (1987). In order to describe these types of integer programs and to briefly state the BB method, we define the following problem.

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

where A is an m by n matrix and $f(x)$ may be specified in a variety of ways depending upon the particular problem to be solved. The available forms for $f(x)$ are listed in Table 1 below. For the moment, however, we assume that $f(x) = c^T x$ so that (1) is a linear programming (LP) problem.

If, in (1), it is required that some (or all) of the variables take integer values, then the integer program is of type *mixed* (or *all*) general IP problem. If, additionally, the integer variables are restricted to take only 0-1 values (i.e., $l_j = 0$ and $u_j = 1$) then the integer program is of type mixed (or all) *zero-one* IP problem. nag_ip_bb does not treat the all integer or zero-one cases specially; therefore, since the mixed integer general IP case is the most general, we shall refer to (1), together with whatever integrality restrictions are applied, as a mixed integer linear programming (MILP) problem, with the assumption that the special cases are included in this.

The BB method applies directly to these integer programs. The general idea of BB is to solve the problem without the integrality restrictions as an LP problem (first or *root node*). If in the optimal solution an integer variable x_k takes a non-integer value x_k^*, two LP sub-problems or *nodes* are created by *branching*, imposing $x_k \le [x_k^*]$ and $x_k \ge [x_k^*] + 1$ respectively, where $[x_k^*]$ denotes the integer part of x_k^*. This method of branching continues until the first integer solution (*bound*) is obtained. The hanging nodes are then solved and investigated in order to prove the optimality of the solution. The algorithm is described in more detail in Section 7).

The same method may also be applied when the objective function $f(x)$ takes other forms. An important assumption for the method to be theoretically valid is that each sub-problem is solved to global optimality. This is the case when, for example, $f(x)$ is a quadratic function which has a positive (semi-)definite Hessian. For such $f(x)$ the sub-problems of the BB search are quadratic programming (QP) problems, which can, in principle, be solved to global optimality. With a quadratic objective function, the problem becomes a mixed integer quadratic programming (MIQP) problem.

nag_ip_bb is able to solve problems in which $f(x)$ is a linear or quadratic function, defined in a variety of ways as described in Table 1 below. The sub-problems are solved using the algorithm of nag_opt_qp (e04nfc).

Problem Type	$f(x)$	**Matrix** H
MILP	$c^T x$	Not applicable
MIQP1	$\frac{1}{2}x^T Hx$	symmetric
MIQP2	$c^T x + \frac{1}{2}x^T Hx$	symmetric
MIQP3	$\frac{1}{2}x^T H^T Hx$	m by n upper trapezoidal
MIQP4	$c^T x + \frac{1}{2}x^T H^T Hx$	m by n upper trapeziodal

Table 1

3.1. Suitability of BB Method for MIQP Problems

The BB method is applicable to an IP problem whenever the global optimum may reliably be found for each sub-problem, and this is theoretically true for an MILP problem. However, this may not be true for an MIQP problem in which the Hessian is not positive (semi-)definite; in such a case the sub-problems may have solutions which are locally but not globally optimal and, in general, it is not possible to ensure that a QP sub-problem solver will always find the global optimum when local optima are present. For problems of type MIQP3 and MIQP4, it is a consequence of the way the Hessian is defined that it must be positive (semi-)definite, but no such guarantee holds for problems of type MIQP1 or MIQP2.

nag_ip_bb does not check if the Hessian is positive (semi-)definite. This provides for the possibility that the user has special knowledge about the problem, for example that an indefinite Hessian is positive (semi-)definite on the feasible region defined by the problem constraints (in which case the problem has no local optima). Alternatively, the user may wish to use nag_ip_bb as a *heuristic*, with the understanding that if a solution is obtained, it may not be the true global optimum of the MIQP problem, or that no solution might be found even though one does exist. If the user wishes to check whether the Hessian of a problem of type MIQP1 or MIQP2 is positive (semi-)definite, and therefore whether any solution obtained can be relied upon, one way this may be achieved is to analyse its eigenvalues (for example using nag_real_symm_eigenvalues (f02aac)): the Hessian is positive semi-definite if and only if all of its eigenvalues are ≥ 0.

3.2. Maximization Problems

nag_ip_bb attempts to solve a *minimization* problem of the form (1) (together with the integrality requirements). In principle, a *maximization* problem can be solved by minimizing $-f(x)$, i.e., reversing the sign of the objective function. This is always valid in the case of an MILP problem, as long as the resulting problem is not unbounded, and simply involves reversing the signs of the coefficients of c (the elements of the input parameter array **cvec**, see Section 4). In the case of an MIQP problem some care must be taken since reversing the sign of a positive (semi-)definite Hessian will make it negative (semi-)definite and vice-versa. Recall that the theoretical validity of the BB method, applied to an MIQP problem, effectively requires that the Hessian be positive (semi-)definite on the feasible region defined by the problem constraints.

Assuming these considerations to be taken into account, a maximization problem of type MIQP1 can be solved by reversing the signs of the elements of H; type MIQP2 problems require the signs of the coefficients of c to be reversed also. Problem types MIQP3 and MIQP4 have a positive (semi-) definite Hessian by definition, so it would not normally make sense to solve these as maximization problems. Hence, nag_ip_bb does not allow the user to reverse the sign of the quadratic objective term for these problem types.

4. Parameters

n

Input: n, the number of variables.
Constraint: $\mathbf{n} > 0$.

m

Input: m, the number of general linear constraints.
Constraint: $\mathbf{m} \geq 0$.

a[m][tda]

Input: the ith row of **a** must contain the coefficients of the ith general linear constraint, for $i = 1, 2, \ldots, m$.

If **m** $= 0$ then the array **a** is not referenced and may be set to the null pointer.

tda

Input: the second dimension of the array **a** as declared in the function from which nag_ip_bb is called.
Constraint: **tda** $\geq$ **n** if **m** > 0.

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 (if any). To specify a non-existent lower bound (i.e., $l_j = -\infty$), set **bl**$[j-1] \leq -$**inf_bound**, and to specify a non-existent upper bound (i.e., $u_j = +\infty$), set **bu**$[j-1] \geq$ **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**$[j-1] = \beta$, say, where $|\beta| <$ **inf_bound**.
Constraint: **bl**$[j] \leq$ **bu**$[j]$, for $j = 0, 1, \ldots,$**n**+**m**-1.

intvar[n]

Input: indicates which are the integer variables in the problem. For example, if x_j is an integer variable then **intvar**$[j-1]$ must be set to 1, and 0 otherwise. The degenerate case, in which all elements of **intvar** are zero, is allowed. In this case, nag_ip_bb solves a single LP or QP problem (depending on the problem type as specified by the optional parameter **prob**, see Section 8.2).
Constraint: **intvar**$[j] = 0$ or 1 for $j = 0, 1, \ldots,$**n**-1.

cvec[n]

Input: the coefficients c_j of the explicit linear term of the objective function when the problem is of type MILP, MIQP2 or MIQP4. The default problem type is MILP; other problem types can be specified using the optional parameter **prob**, see Section 8.2.

If the problem is of type MIQP1 or MIQP3, **cvec** is not referenced and may be set to the null pointer.

h[n][tdh]

Input: **h** may be used to store the quadratic term H of the MIQP objective function if desired. The elements of **h** are accessed only by the function **qphess**; thus, **h** is not accessed if the problem is of the type MILP (the default) and may be set to the null pointer.

The number of rows of **h** is denoted by n_H and its default value is equal to n. (The optional parameter **hrows** may be used to specify a value of $n_H < n$; see Section 8.2).

If the problem is of type MIQP1 or MIQP2, the first n_H rows and columns of **h** must contain the leading n_H by n_H rows and columns of the symmetric Hessian matrix. Only the diagonal and upper triangular elements of the leading n_H rows and columns of **h** are referenced. The remaining elements need not be assigned.

For problems of type MIQP3 and MIQP4, the first n_H rows of **h** must contain an n_H by n upper trapezoidal factor of the Hessian matrix. The factor need not be of full rank, i.e., some of the diagonals may be zero. However, as a general rule, the larger the dimension of the leading non-singular sub-matrix of H, the fewer iterations will be required. Elements outside the upper trapezoidal part of the first n_H rows of H are assumed to be zero and need not be assigned.

In some cases, the user need not use **h** to store H explicitly (see the specification of function **qphess** below).

tdh

Input: the second dimension of the array **h** as declared in the function from which nag_ip_bb is called.
Constraint: **tdh** $\geq$ **n** or at least the value of the optional parameter **hrows** if it is set. This constraint is enforced only for problems of type MIQP in which the **qphess** parameter is null.

qphess

In general, the user need not provide a version of **qphess**, because a 'default' function is included in the NAG C Library. If the default function is required then the NAG defined null function pointer, `NULLFN`, should be supplied in the call to nag_ip_bb. The algorithm of nag_ip_bb requires only the product of H and a vector x; and in some cases the user may obtain increased efficiency by providing a version of **qphess** that avoids the need to define the elements of the matrix H explicitly.

qphess is not referenced for problems of type MILP (the default), in which case **qphess** should be replaced by `NULLFN`.

The specification of **qphess** is:

```
void qphess(Integer n, Integer jthcol, double h[], Integer tdh, double x[],
            double hx[], Nag_Comm *comm)
```

n

Input: n, the number of variables.

jthcol

Input: **jthcol** specifies whether or not the vector x is a column of the identity matrix. If **jthcol** $= j > 0$, then the vector x is the jth column of the identity matrix, and hence Hx is the jth column of H, which can sometimes be computed very efficiently and **qphess** may be coded to take advantage of this. However special code is not necessary because x is always stored explicitly in the array **x**. If **jthcol** $= 0$, x has no special form.

h[n*tdh]

Input: the matrix H of the QP objective function.
The matrix element H_{ij} is contained in $\mathbf{h}[(i-1)*tdh+j-1]$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$. In some situations, it may be desirable to compute Hx without accessing **h** – for example, if H is sparse or has special structure. (This is illustrated in the function `qphess` in the example program in Section 13.) The parameters **h** and **tdh** may then refer to any convenient array.

tdh

Input: the second dimension of the array **h** in the calling program.

x[n]

Input: the vector x.

hx[n]

Output: the product Hx.

comm

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

flag – Integer

Input: **qphess** is called with **comm -> flag** set to a non-negative number.
Output: if **qphess** resets **comm -> flag** to some negative number then nag_ip_bb will terminate immediately with the error indicator **NE_USER_STOP**. If **fail** is supplied to nag_ip_bb, **fail.errnum** will be set to the user's setting of **comm -> flag**.

first – Boolean

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

nf – Integer

Input: the number of calls made to **qphess** 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_ip_bb these pointers may be allocated memory by the user and initialised with various quantities for use by **qphess** when called from nag_ip_bb.

Note: **qphess** should be tested separately before being used in conjunction with nag_ip_bb. The input arrays **h** and **x** must **not** be changed by **qphess**.

x[n]

Input: an initial estimate of the solution of the first sub-problem (the *root node* problem as described in Section 3).

If optional parameter **branch_dir** = **Nag_Branch_InitX** (which is not the default value), then the initial values in **x** of the integer variables influence the branching procedure in the BB algorithm. Typically, an estimate of the values of the integer variables in the IP solution would be provided in this case. See Section 8.2 for details.

Output: with **fail.code** = **NE_NOERROR**, **x** contains a solution which will be an estimate of either the optimum integer solution or the first integer solution, depending on the value of optional parameter **first_soln**. If **fail.code** = **NW_MIP_MAX_NODES_INT_SOL**, **NW_MIP_MAX_DEPTH_INT_SOL**, **NW_MIP_MAX_ITER_INT_SOL**, or **NE_MIP_HESS_TOO_BIG_INT_SOL** then **x** contains a solution which may not be the optimal IP solution because nag_ip_bb was unable to investigate all of the nodes. See Section 9 for more details.

objf

Output: with **fail.code** = **NE_NOERROR**, **NW_MIP_MAX_NODES_INT_SOL**, **NW_MIP_MAX_DEPTH_INT_SOL**, **NW_MIP_MAX_ITER_INT_SOL**, or **NE_MIP_HESS_TOO_BIG_INT_SOL**, **objf** contains the value of the objective function for the IP solution.

options

Input/Output: a pointer to a structure of type Nag_H02_Opt whose members are optional parameters for nag_ip_bb. 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.

The **options** structure also allows names to be assigned to the variables and constraints of the problem, which are then used in solution output. In particular, if the problem is defined by an MPSX file, the function nag_ip_mps_read (h02buc) may be used to read the file, and to store the variable and constraint names in **options** for use by nag_ip_bb.

If any of these optional parameters are required then the structure **options** should be declared and initialised by a call to nag_ip_init (h02xxc) and supplied as an argument to nag_ip_bb. However, if the optional parameters are not required the NAG defined null pointer, `H02_DEFAULT`, can be used in the function call.

comm

Input/Output: structure containing pointers for communication to the user-supplied function, **qphess**, and the optional user-defined printing function. See the description of **qphess** 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_ip_bb; **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 initialise **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 the end of each node and the final IP result. If nag_ip_bb fails to find an IP solution, the final solution printed will be the original LP or QP (*root node*) solution. This section describes the default printout produced by nag_ip_bb.

The following line of summary output is produced at the end of every node. It gives the outcome of forcing an integer variable with a non-integer value to take a value within its specified lower and upper bounds.

`Node No` is the current node number of the BB tree being investigated.

`Parent Node` is the parent node number of the current node.

`Obj Value` is the final objective function value. If a node does not have a feasible solution then `Infeasible` is printed instead of the objective function value. If a node whose optimum solution exceeds the best integer solution so far is encountered (i.e., it does not pay to explore the sub-problem any further), then its objective function value is printed together with a `CO` (Cut Off).

`Varbl Chosen` is the index of the integer variable chosen for branching.

`Value Before` is the non-integer value of the integer variable chosen.

`Lower Bound` is the lower bound value that the integer variable is allowed to take.

`Upper Bound` is the upper bound value that the integer variable is allowed to take.

`Value After` is the value of the integer variable after the current optimization.

`Depth` is the depth of the BB tree at the current node.

The final printout includes a listing of the status of each variable and constraint.

`Varbl` gives the name of variable j, for $j = 1, 2, \ldots, n$. If an **options** structure is supplied to nag_ip_bb, and the **crnames** member is assigned to an array of variable and constraint 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 (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than the feasibility tolerance, `State` will be `++` or `--` respectively.

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

`Lower Bound` is the lower bound l_j specified for the variable. (`None` indicates that $l_j \leq -$**inf_bound**, where **inf_bound** is the optional parameter.) The bound is that imposed at the node which provided the IP solution. (If no IP solution was found, the bound is that supplied by the user in **bl**.)

`Upper Bound` is the upper bound u_j specified for the variable. (`None` indicates that $u_j \geq$ **inf_bound**.) The bound is that imposed at the node which provided the IP solution. (If no IP solution was found, the bound is that supplied by the user in **bu**.)

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

`Residual` is the difference between the variable `Value` and the nearer of its bounds l_j and u_j.

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**$[n+j-1]$, l_j and u_j replaced by l_{n+i} and u_{n+i} respectively, and with the following change in the heading:

`Constr` gives the name of the constraint.

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_ip_bb is given in Section 9. The accuracy of nag_ip_bb is considered in Section 10, where further comments may also be found.

6. Example 1

To solve the integer programming problem:
maximize

$$F(x) = 3x_1 + 4x_2$$

subject to the bounds

$$\begin{aligned} x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

and to the general constraints

$$\begin{aligned} 2x_1 + 5x_2 &\leq 15 \\ 2x_1 - 2x_2 &\leq 5 \\ 3x_1 + 2x_2 &\geq 5 \end{aligned}$$

where x_1 and x_2 are integer variables. The initial point, which is feasible, is

$$x_0 = (1, 1)^T,$$

and $F(x_0) = 7$. The optimal solution is

$$x^* = (2, 2)^T,$$

and $F(x^*) = 14$. Note that maximizing $F(x)$ is equivalent to minimizing $-F(x)$.

This example shows the simple use of nag_ip_bb 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.

6.1. Program Text

```
/* nag_ip_bb (h02bbc) Example Program
 *
 * Copyright 1998 Numerical Algorithms Group.
 *
 * Mark 5, 1998.
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nag_string.h>
#include <nagh02.h>

#ifdef NAG_PROTO
static void ex1(void);
```

```
static void ex2(void);
static void qphess(Integer n, Integer jthcol, double h[], Integer tdh,
                   double x[], double hx[], Nag_Comm *comm);
#else
static void ex1();
static void ex2();
static void qphess();
#endif

#define MAXN 10
#define MAXM 7
#define MAXBND MAXN+MAXM

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("h02bbc Example Program Results.\n");
  ex1();
  ex2();
  exit(EXIT_SUCCESS);
}

static void ex1()
{
  /* Local variables */
  double a[MAXM][MAXN], cvec[MAXN], bl[MAXBND], bu[MAXBND];
  double x[MAXN];
  double objf;
  Integer i, j, is_int;
  Integer m,  n, nbnd, tda;
  Boolean intvar[MAXN];
  static NagError fail;

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

  fail.print = TRUE;
  tda = MAXN;

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

  /* Read objective coefficients */
  Vscanf(" %*[^\n]");
  for (i = 0; i < n; ++i)
    Vscanf("%lf", &cvec[i]);

  /* Read the matrix coefficients */
  Vscanf(" %*[^\n]");
  for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
      Vscanf("%lf",&a[i][j]);

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

  /* Read which variables are integer */
  Vscanf(" %*[^\n]");
```

```
  for (i = 0; i < n; ++i)
    {
      Vscanf("%ld", &is_int);
      /* is_int = 1 if integer variable, 0 if not */
      intvar[i] = is_int ? TRUE : FALSE;
    }

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

  h02bbc(n, m, (double *)a, tda, bl, bu, intvar, cvec, (double *)0, 0,
         NULLFN, x, &objf, H02_DEFAULT, NAGCOMM_NULL, &fail);
} /* ex1 */

#ifdef NAG_PROTO
static void qphess(Integer n, Integer jthcol, double h[], Integer tdh,
                   double x[], double hx[], Nag_Comm *comm)
#else
static void qphess(n, jthcol, h, tdh, x, hx, comm)
     Integer n, jthcol, tdh;
     double h[], x[], hx[];
     Nag_Comm *comm;
#endif
{
  Integer i;

  /* In this qphess function the Hessian is defined implicitly */
  for (i = 0; i < n; ++i)
    hx[i] = 2.0*x[i];
} /* qphess */
```

6.2. Program Data

```
h02bbc Example Program Data

Data for example 1

Values of m, n
  3  2

Objective coefficients, cvec
 -3.0  -4.0

Constraint matrix a
  2.0   5.0
  2.0  -2.0
  3.0   2.0

Lower bounds
  0.0  0.0  -1.0e+20  -1.0e+20  5.0

Upper bounds
  1.0e+20  1.0e+20  15.0  5.0  1.0e+20

Integer variables (1 if integer, 0 if not)
  1  1

Initial estimate of x
  1.0  1.0
```

6.3. Program Results

```
h02bbc Example Program Results.

Example 1: default options used.

Parameters to h02bbc
--------------------
```

```
Linear constraints............  3    Number of variables...........  2
Number of integer variables...  2

prob.................... Nag_MILP
feas_tol................ 1.05e-08    machine precision....... 1.11e-16
inf_bound............... 1.00e+20    max_iter................       50
first_soln..............    FALSE    max_depth...............       10
max_nodes...............ALL_NODES    int_tol................. 1.00e-05
int_obj_bound........... 1.00e+20    soln_tol................ 1.05e-08
nodsel..........Nag_MinObj_Search    varsel..............Nag_First_Int
branch_dir........Nag_Branch_Left    crnames..............not supplied
print_level.........Nag_Soln_Iter
outfile.................   stdout

Memory allocation:
lower...................      Nag
upper...................      Nag
state...................      Nag
lambda..................      Nag

  Node Parent     Obj     Varbl      Value      Lower      Upper      Value Depth
   No   Node     Value   Chosen     Before      Bound      Bound      After
    1       -1.750e+01
    2      1 -1.620e+01        1  3.93e+00  0.00e+00  3.00e+00   3.00e+00     1
    3      1 Infeasible        1  3.93e+00  4.00e+00      None   3.93e+00     1
    4      2 -1.300e+01        2  1.80e+00  0.00e+00  1.00e+00   1.00e+00     2
   *** Integer Solution ***

    5      2 -1.550e+01        2  1.80e+00  2.00e+00      None   2.00e+00     2
    6      5 -1.480e+01        1  2.50e+00  0.00e+00  2.00e+00   2.00e+00     3
    7      5 Infeasible        1  2.50e+00  3.00e+00  3.00e+00   2.50e+00     3
    8      6 -1.400e+01        2  2.20e+00  2.00e+00  2.00e+00   2.00e+00     4
   *** Integer Solution ***

    9      6 -1.200e+01 CO     2  2.20e+00  3.00e+00      None   3.00e+00     4

Final solution:

Varbl   State    Value       Lower Bound  Upper Bound    Lagr Mult    Residual

V 1       UL   2.00000e+00    0.0000e+00   2.0000e+00   -3.000e+00   0.000e+00
V 2       EQ   2.00000e+00    2.0000e+00   2.0000e+00   -4.000e+00   0.000e+00

Constr  State    Value       Lower Bound  Upper Bound    Lagr Mult    Residual

C 1       FR   1.40000e+01      None       1.5000e+01    0.000e+00   1.000e+00
C 2       FR   0.00000e+00      None       5.0000e+00    0.000e+00   5.000e+00
C 3       FR   1.00000e+01    5.0000e+00      None       0.000e+00   5.000e+00

Exit from branch and bound tree search after 9 nodes.

Optimal IP solution found.

Final IP objective value =  -1.4000000e+01
```

7. Further Description

This section provides further information about the BB algorithm used by nag_ip_bb. 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.

Further descriptions of the BB algorithm may be found in Dakin (1965) and Mitra (1973).

7.1. Overview

As outlined in Section 3, the essence of the BB algorithm is to form a 'tree' of sub-problems which are relatively easy to solve. The initial sub-problem, the *root node* of the tree, is a *relaxation* of the IP problem, in that it is the IP problem with the integer restrictions removed. When that has been solved, two *child* sub-problems or *nodes* are formed by selecting an integer variable x_k which in the solution to the relaxed problem takes a non-integer value x_k^*, and *branching* on that variable, i.e., imposing $x_k \leq [x_k^*]$ for one node and $x_k \geq [x_k^*] + 1$ for the other, where $[x_k^*]$ denotes the integer part of x_k^*. One of these nodes is then solved. At this point, either a further branching operation is carried out from the node just solved, creating two new unsolved nodes (one of which is solved next), or the remaining unsolved child node is solved. Continuing in this way, the tree is developed – at each stage selecting an unsolved node to solve, or a solved node to branch from. The selection of the node and, in the case of a branching operation, the selection of the variable to branch on, is considered further in Section 7.2.

The mechanism for forming the nodes on branching simply involves adjusting the lower or upper bound on the branching variable. Note that as the tree is descended, each child node inherits any bound adjustments made to its parent node, and so a child node is always more constrained than its parent.

If the procedure described above is continued, eventually a child must be created for which all of its integer variables are fixed at integer values, or which is infeasible. If the latter is true then the search down that branch of the tree may be terminated since any children of that node must also be infeasible (the child is always more constrained than the parent). If the former is true then we have an integer feasible solution for the IP problem, which may or may not be the optimum integer solution. For some applications of IP, it is sufficient to obtain any integer feasible solution and the search may terminate here, but usually the search must be continued, either to find a better integer solution, or to confirm that the optimal integer solution has been found. In nag_ip_bb the optional parameter **first_soln** may be set to **TRUE** to request termination at the first integer solution (the default value is **FALSE**; see Section 8.2).

Assuming that the optimal integer solution is required, the rest of the tree must be searched. The efficiency of the method relies on not having to examine every node of the tree which could, potentially, be formed by applying the procedure as described above. The method incorporates features which have the effect of eliminating certain portions of the tree from the search. As already explained, the search is terminated along a particular branch on encountering an infeasible node. Similarly, once an integer solution has been found, this can be used to eliminate parts of the search tree as follows. Suppose an integer feasible solution x^+ has been found, with an associated objective function value $f(x^+)$. Now suppose during the search of the remainder of the tree, a node is encountered, whose objective function value exceeds $f(x^+)$. In this case there is no need to examine any further down that branch of the tree since any children of that node will also have objective function values which exceed $f(x^+)$. The quantity $f(x^+)$ therefore acts as a *bound* on the optimal integer solution. This bound may be refined as better integer solutions are found. Finally, if an integer solution is found before all integer variables have been fixed by the branching process, simply because the unfixed integer variables happen to have integer values at the solution of a particular node, there is again no need to search further along that branch of the tree. Termination of the search at a node, whether through finding an integer solution there, detecting infeasibility, or bounding it based on a known integer solution, is known as *fathoming* the node.

7.2. Selection of Node and Branching Variable

Since each branching operation generates two unsolved nodes (sub-problems), at a typical stage of the algorithm there will be a number of nodes which are either unsolved or which have been solved but have not yet been branched from. Therefore, when a node has been solved there is a choice to be made as to which node should be solved next, and this will either be an existing, unsolved node, or one which will be created by a branching operation.

If a node is selected to be branched from, there is a further choice to be made and that is the integer variable to be branched on.

Within nag_ip_bb these choices are controlled by the optional parameters **nodsel**, which controls node selection, and **varsel**, which controls branching variable selection. The default node selection behaviour is to choose the node with lowest objective value, if it has been solved, or lowest parent objective value if it is unsolved. By default the branching variable chosen is that with the smallest index in x, selected from those integer variables taking non-integer values at the solution of the sub-problem being branched from. Details of the available options are given in Section 8.2.

These choices can help to improve the efficiency of the BB algorithm since they particularly influence how quickly the first integer feasible solution is obtained and its quality. A good integer solution obtained early in the search can eliminate a large portion of the remaining tree, by means of the bounding operation described in Section 7.1). Unfortunately, there is no single strategy for making such choices which can be applied successfully to all IP problems – the best strategy is highly problem dependent and is usually obtained by experimentation.

7.3. Further Reducing the Size of the BB Search Tree

In addition to considering variations in the node and variable selection strategies, the user may also consider setting some other parameters to help to reduce the number of nodes searched. Recall from Section 7.1) that once the algorithm has found an IP solution, the objective function value associated with this is used as a bound to eliminate parts of the tree. Similarly, if the user knows from the outset a strict upper bound on the optimal solution, perhaps as a result of solving a related, more constrained problem, or obtained through analytical means, this may be supplied to nag_ip_bb as the optional parameter **int_obj_bound**. This will be used by nag_ip_bb in the same way as a bound obtained by finding an IP solution except that it can be used to eliminate parts of the tree even before an integer solution is found.

Another parameter which the user might consider setting to reduce the size of the tree is **soln_tol**. Again this is related to the bounding process, and applies when an integer solution has been found. When searching the remainder of the tree, instead of setting the bound to $f(x^+)$, the objective function value associated with the integer solution most recently found, nag_ip_bb sets the bound to $f(x^+) -$ **soln_tol**. This means that integer solutions with objective values within **soln_tol** of any integer solution already found, can not themselves be found. The idea here is to allow the user to avoid further search for solutions which are not substantially better (as measured by **soln_tol**) than the best solution found so far. Of course, a sensible choice for the value of **soln_tol** relies on the user's knowledge of the problem and requirements on the solution.

Further details of the optional parameters **int_obj_bound** and **soln_tol** are given in Section 8.2.

Finally, a very important factor which can have a large impact on the size of the search tree is the way the problem is modelled. Often, there is more than one way to formulate a problem as an IP model. A general aim is that the feasible region of the relaxed IP problem should be as close as possible to that of the IP problem itself. This has the effect of generating tight bounds in the BB procedure. Note that in order to achieve this aim, it may be necessary to introduce further constraints, which do not alter the IP solution but which help to reduce the feasible region of the sub-problems. This is in contrast to standard LP, for example, in which fewer constraints are generally considered to be associated with an easier problem. There is of course a balance to be struck since adding constraints to an IP problem will make the sub-problems harder to solve, despite, it is hoped, reducing the size of the tree. See Williams (1993) for more information on formulating IP models.

8. Optional Parameters

A number of optional input and output parameters to nag_ip_bb are available through the structure argument **options**, type Nag_H02_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, `H02_DEFAULT`, in place of **options** when calling nag_ip_bb; the default settings will then be used for all parameters.

Before assigning values to **options** directly the structure **must** be initialised by a call to the function nag_ip_init (h02xxc). 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_ip_read (h02xyc) in which case initialisation 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 initialisation.

If assignment of functions and memory to pointers in the **options** structure is required, then this must be done directly in the calling program; they cannot be assigned using nag_ip_read (h02xyc).

8.1. Optional Parameter Checklist and Default Values

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

`Nag_MIP_ProbType prob`	**Nag_MILP**
`Boolean list`	**TRUE**
`Nag_PrintType print_level`	**Nag_Soln_Iter**
`char outfile[80]`	`stdout`
`void (*print_fun)()`	NULL
`Integer max_iter`	max(50,5(**n**+**m**))
`Integer max_nodes`	`ALL_NODES`
`Boolean first_soln`	**FALSE**
`Integer max_depth`	max(10,3**n**/2)
`double int_tol`	10^{-5}
`double int_obj_bound`	10^{20}
`double soln_tol`	$\sqrt{\epsilon}$
`Nag_Node_Selection nodsel`	`Nag_MinObj_Search`
`Nag_Var_Selection varsel`	`Nag_First_Int`
`Nag_Branch_Direction branch_dir`	`Nag_Branch_Left`
`double *priority`	NULL
`double feas_tol`	$\sqrt{\epsilon}$
`double inf_bound`	10^{20}
`double rank_tol`	100ϵ
`Integer hrows`	0 or **n**
`Integer max_df`	**n**
`char **crnames`	NULL
`double *lower`	size **n**+**m**
`double *upper`	size **n**+**m**
`double *lambda`	size **n**+**m**
`Integer *state`	size **n**+**m**

8.2. Description of Optional Parameters

prob – Nag_MIP_ProbType Default = **Nag_MILP**

Input: specifies the type of objective function to be minimized during the optimality phase. The following are the five possible values of **prob** and the size of the arrays **h** and **cvec** that are required to define the objective function:

Nag_MILP **h** not referenced, **cvec**[**n**];

Nag_MIQP1 **h**[**n**][**tdh**] symmetric, **cvec** not referenced;

Nag_MIQP2 **h**[**n**][**tdh**] symmetric, **cvec**[**n**];

Nag_MIQP3 **h**[**n**][**tdh**] upper trapezoidal, **cvec** not referenced;

Nag_MIQP4 **h**[**n**][**tdh**] upper trapezoidal, **cvec**[**n**].

Constraint: **options.prob** = **Nag_MILP**, **Nag_MIQP1**, **Nag_MIQP2**, **Nag_MIQP3** or **Nag_MIQP4**.

list – Boolean Default = **TRUE**

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

print_level – Nag_PrintType Default = **Nag_Soln_Iter**

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

Nag_NoPrint No output.

Nag_Soln The final IP solution.

Nag_Soln_Root The root node and final IP solution.

Nag_Iter	One line of output for each node investigated.
Nag_Soln_Iter	The final IP solution and one line of output for each node.
Nag_Soln_Root_Iter	The root node and final IP solution and one line of output for each node.

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

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.

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

Input: the limit on the number of iterations for each node.
Constraint: **options.max_iter** ≥ 0.

max_nodes – Integer Default = `ALL_NODES`

Input: the maximum number of nodes that are to be searched in order to find a solution (optimum integer solution). If **max_nodes** is not set, or is set equal to the symbol `ALL_NODES`, and the optional parameter **first_soln** = **FALSE** (the default), then the BB tree search is continued until all the nodes have been investigated.
Constraint: **options.max_nodes** > 0 or
options.max_nodes = `ALL_NODES`.

first_soln – Boolean Default = **FALSE**

Input: specifies whether to terminate the BB tree search after the first integer solution (if any) is obtained. If **first_soln** = **TRUE** then the BB tree search is terminated at node k say, which contains the first integer solution. For optional parameter **max_nodes** $\neq$ `ALL_NODES` this applies only if $k \leq$ **max_nodes**.

max_depth – Integer Default = max(10,3**n**/2)

Input: the maximum depth of the BB tree used for branch and bound.
Constraint: **options.max_depth** ≥ 2.

int_tol – double Default = 10^{-5}

Input: the integer feasibility tolerance; i.e., an integer variable is considered to take an integer value if its violation does not exceed **int_tol**. For example, if the integer variable x_j is of order unity then x_j is considered to be integer if (1−**int_tol**) $\leq x_j \leq$ (1+**int_tol**).
Constraint: **options.int_tol** > 0.0.

int_obj_bound – double Default = 10^{20}

Input: specifies an initial bound on the optimum integer solution. The user should supply a value for this parameter only if a valid strict upper bound for the IP problem is available. Supplying too small a value will result in nag_ip_bb not finding an IP solution. If a valid value is provided then this may help to reduce the number of nodes searched in the BB tree (see Section 7.3).

The default value, 10^{20}, is equivalent to no such bound being available.

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

Input: specifies a tolerance on the optimal IP solution, i.e., an IP solution returned by nag_ip_bb as optimal may have an objective function value which is as much as **soln_tol** greater than that associated with the true optimal IP solution. By setting **soln_tol** to a non-zero value, the size of the BB search tree may be reduced at the expense of obtaining a (possibly) inferior solution (see Section 7.3).

This parameter only takes effect after the first IP solution has been found. It therefore has no effect if optional parameter **first_soln** = **TRUE** and need not be taken into account when setting optional parameter **int_obj_bound**.
Constraint: **options.soln_tol** ≥ 0.0.

nodsel – Nag_Node_Selection Default = **Nag_MinObj_Search**

Input: specifies how nodes are selected during the BB tree search (see Section 7.2). The selection is made from those nodes which are still 'active', i.e., those which either have not yet been solved, or which have been solved but not yet branched from. If the node selected has not been solved then it will be solved next; otherwise, it is branched from and one of the resulting child nodes will be solved next. In the latter case, the choice of which child node is solved first is determined by the value of optional parameter **branch_dir** (see below). The possible values of **nodsel** and their meanings are described below.

Nag_MinObj_Search selects the node with smallest objective function value. A node which has not yet been solved is assigned its parent's objective function value as the basis for its selection.

Nag_Deep_Search selects the deepest node in the BB tree. When selecting a node for branching and there is more than one candidate at the deepest level, preference is given to the node which was solved earliest. This type of node selection is affected by the value of **branch_dir** (see below).

Nag_Broad_Search selects the shallowest node in the tree. This has the effect of searching across the tree (rather than down as for **Nag_Deep_Search**).

Nag_DeepMinObj_Search as **Nag_Deep_Search** until the first integer solution is found and as **Nag_MinObj_Search** thereafter.

Nag_DeepBroad_Search as **Nag_Deep_Search** until the first integer solution is found and as **Nag_Broad_Search** thereafter.

Constraint: **options.nodsel** = **Nag_MinObj_Search**, **Nag_Deep_Search**, **Nag_Broad_Search**, **Nag_DeepMinObj_Search** or **Nag_DeepBroad_Search**.

varsel – Nag_Var_Selection Default = **Nag_First_Int**

Input: specifies how nag_ip_bb selects the variable to branch on, when an unbranched node has been chosen according to optional parameter **nodsel**. Let x^* denote the solution associated with the selected node. Integer variables are scanned in order of their index in x, and any which are integral to within the optional tolerance parameter **int_tol** are ignored. The following values of **varsel** are available.

Nag_First_Int select the first integer variable x_i such that x_i^* is non-integer.

Nag_Nearest_Half select the integer variable x_i such that $|x_i^* - [x_i^*]|$ is nearest to 0.5, where $[x_i^*]$ denotes the integer part of x_i^*. That is, x_i is the integer variable such that x_i^* is farthest from having an integer value.

Nag_Use_Priority branch on the integer variable selected according to the set of priorities provided by the user in optional parameter **priority** (see below).

Constraint: **options.varsel** = **Nag_First_Int**, **Nag_Nearest_Half** or **Nag_Use_Priority**.

branch_dir – Nag_Branch_Direction Default = **Nag_Branch_Left**

Input: specifies which node to solve first when two nodes are created by a branching operation. This option is unlikely to have much effect when optional parameter **nodsel** = **Nag_MinObj_Search** or **Nag_Broad_Search**, since the overall order in which parts of the tree are examined will remain the same. However, when **nodsel** = **Nag_Deep_Search**, **branch_dir** will influence the path taken by nag_ip_bb as the tree is descended. Similarly, this parameter will affect the initial deep search when **nodsel** = **Nag_DeepMinObj_Search** or **Nag_DeepBroad_Search**. The following values of **branch_dir** are available.

Nag_Branch_Left solve the 'left' node first, i.e., that which was formed by reducing the upper bound on the branching variable.

Nag_Branch_Right solve the 'right' node first, i.e., that which was formed by increasing the lower bound on the branching variable.

Nag_Branch_InitX branch according to the initial values of the integer variables, as supplied in the parameter **x** to nag_ip_bb. Let x^0 be the initial solution as supplied by the user, and let i be the index of the integer variable currently being branched on. Then if z_i^0 is the nearest integer to x_i^0 which satisfies the initial bounds on x, nag_ip_bb will first branch towards z_i^0 and solve this sub-problem. This value of **branch_dir** would be appropriate, in conjunction with a deep search (as defined by **nodsel**), if the user can provide in **x** a good estimate of an integer solution to the IP problem.

Constraint: **options.branch_dir** = **Nag_Branch_Left**, **Nag_Branch_Right** or **Nag_Branch_InitX**.

priority – double * **Default** = **NULL**

Input: if **varsel** = **Nag_Use_Priority** then for each integer variable x_i, **priority**$[i-1]$ must contain the priority the variable should be given when nag_ip_bb selects a variable to branch on (x_i is an integer variable if **intvar**$[i-1]$ = **TRUE**, for $i = 1, 2, \ldots, n$). For example, if x_k and x_l are integer variables and **priority**$[l-1]$ > **priority**$[k-1]$, then variable x_l will be selected in preference to x_k. Variables with equal priorities are selected according to their indices (i.e., x_k is selected if $k < l$ and **priority**$[k-1]$ = **priority**$[l-1]$).

With some problems of type MILP, setting **priority** to **cvec** might be effective, since the objective coefficient of a variable could be regarded as a measure of the importance of the variable in the problem.

If x_i is not an integer variable (i.e., **intvar**$[i-1]$ = **FALSE**), **priority**$[i-1]$ is not referenced. If optional parameter **nodsel** ≠ **Nag_Use_Priority** then **priority** is not referenced.

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

Input: the maximum acceptable absolute violation in each constraint at a 'feasible' point (feasibility tolerance); i.e., a constraint is considered satisfied if its violation does not exceed **feas_tol**.
Constraint: **options.feas_tol** > 0.0.

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.

rank_tol – double Default = 100ϵ

This parameter is not used for problems of type MILP.

Input: **rank_tol** enables the user to control the condition number of the triangular matrix factor R which arises in solving a QP subproblem (see Section 7 of the documentation for nag_opt_qp (e04nfc) for details). If ρ_i denotes the function $\rho_i = \max\{|R_{11}|, |R_{22}|, \ldots, |R_{ii}|\}$, the dimension of R is defined to be smallest index i such that $|R_{i+1,i+1}| \leq$ **rank_tol** $\times |\rho_{i+1}|$.
Constraint: $0.0 \leq$ **options.rank_tol** < 1.0.

hrows – Integer Default = 0 or **n**

Input: specifies n_H, the number of rows of the quadratic term H of the QP objective function. For the default MILP problem type, **hrows** is not used and its value is set to zero. For MIQP problem types, the default value of **hrows** is **n**, the number of variables. However, a value of **hrows** less than **n** is appropriate for problems of type MIQP3 or MIQP4 when H is an upper

trapezoidal matrix with n_H rows. Similarly, **hrows** may be used to define the dimension of a leading block of non-zeros in the Hessian matrices for problems of type MIQP1 or MIQP2, in which case the last $\mathbf{n} - n_H$ rows and columns of H are assumed to be zero.
Constraint: $0 \leq$ **options.hrows** $\leq$ **n**.

max_df – Integer Default = **n**

Input: places a limit on the storage allocated for the triangular factor R of the reduced Hessian H_r of QP sub-problems (see Section 7 of the documentation for nag_opt_qp (e04nfc) for details). Ideally, **max_df** should be set slightly larger than the value of n_r (the number of rows and columns of H_r) expected at the solution. It need not be larger than $m_n + 1$, where m_n is the number of variables that appear nonlinearly in the quadratic objective function. For many problems it can be much smaller than m_n.

For quadratic problems, a minimizer may lie on any number of constraints, so that n_r may vary between 1 and n. The default value is therefore normally **n** but if the optional parameter **hrows** is specified then the default value of **max_df** is set to the value in **hrows**.
Constraint: $1 \leq$ **options.max_df** $\leq$ **n**.

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 variables and constraints of the problem. Thus, **crnames**$[j-1]$ contains the name of the the jth variable, $j = 1, 2, \ldots,$**n**, and **crnames**[**n**$+i-1$] contains the names of the ith 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_ip_mps_read (h02buc) prior to calling nag_ip_bb. In this case, nag_ip_mps_read (h02buc) may optionally be used to allocate memory to **crnames** and to read the variable and constraint names defined in the MPSX file into **crnames**. In this case, the memory freeing function nag_ip_free (h02xzc) should be used to free the memory pointed to by **crnames** on return from nag_ip_bb. Users should **not** use the standard C function `free()` for this purpose.

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

Input: **n**+**m** values of memory will be automatically allocated by nag_ip_bb and this is the recommended method of use of **options.lower**. However a user may supply memory from the calling program.
Output: the lower bounds imposed at the point returned in **x**. If no IP solution was found **lower** contains the same bounds as supplied by the user in **bl**. The first **n** elements contain the lower bounds on the variables, and the next **m** elements contain the lower bounds for the general linear constraints (if any).

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

Input: **n**+**m** values of memory will be automatically allocated by nag_ip_bb and this is the recommended method of use of **options.upper**. However a user may supply memory from the calling program.
Output: the upper bounds imposed at the point returned in **x**. If no IP solution was found **upper** contains the same bounds as supplied by the user in **bu**. The first **n** elements contain the upper bounds on the variables, and the next **m** elements contain the upper bounds for the general linear constraints (if any).

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

Input: **n**+**m** values of memory will be automatically allocated by nag_ip_bb and this is the recommended method of use of **options.state**. However a user may supply memory from the calling program.
Output: the status of the constraints in the working set at the point returned in **x**. The significance of each possible value of **state**$[j]$ is as follows:

state$[j]$	**Meaning**
-2	The constraint violates its lower bound by more than the feasibility tolerance.
-1	The constraint violates its upper bound by more than the feasibility tolerance.

0 The constraint is satisfied to within the feasibility tolerance, but is not in the working set.

1 This inequality constraint is included in the working set at its lower bound.

2 This inequality constraint is included in the working set at its upper bound.

3 This constraint is included in the working set as an equality. This value of **state** can occur only when **bl**[j] = **bu**[j].

4 This corresponds to optimality being declared with **x**[j] being temporarily fixed at its current value. This value of **state** can only occur if the optimal solution is not unique.

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

Input: **n+m** values of memory will be automatically allocated by nag_ip_bb 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 Lagrange multipliers for each constraint with respect to the current working set at the point returned in **x**. The first **n** elements contain the multipliers (reduced costs) for the bound constraints on the variables, and the next **m** elements contain the multipliers (shadow costs) for the general linear constraints (if any). If **state**[j] = 0, **lambda**[j] is zero. If x is optimal, **lambda**[j] should be non-negative if **state**[j] = 1, non-positive if **state**[j] = 2 and zero if **state**[j] = 4.

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_ip_bb are listed, whereas the printout of results is governed by the value of **print_level**. The default of **print_level** = **Nag_Soln_Iter** provides intermediate and final results.

If **print_level** = **Nag_Iter**, **Nag_Soln_Iter** or **Nag_Soln_Root_Iter**, the following line of summary output is produced at the end of every node. It gives the outcome of forcing an integer variable with a non-integer value to take a value within its specified lower and upper bounds.

`Node No` is the current node number of the BB tree being investigated.

`Parent Node` is the parent node number of the current node.

`Obj Value` is the final objective function value. If a node does not have a feasible solution then `Infeasible` is printed instead of the objective function value. If a node whose optimum solution exceeds the best integer solution so far is encountered (i.e., it does not pay to explore the sub-problem any further), then its objective function value is printed together with a `CO` (Cut Off).

`Varbl Chosen` is the index of the integer variable chosen for branching.

`Value Before` is the non-integer value of the integer variable chosen.

`Lower Bound` is the lower bound value that the integer variable is allowed to take.

`Upper Bound` is the upper bound value that the integer variable is allowed to take.

`Value After` is the value of the integer variable after the current optimization.

`Depth` is the depth of the BB tree at the current node.

If **print_level** = **Nag_Soln_Root** or **Nag_Soln_Root_Iter**, the root node solution is output before the BB search is commenced. If **print_level** = **Nag_Soln**, **Nag_Soln_Iter**, **Nag_Soln_Root** or **Nag_Soln_Root_Iter** the final IP solution or, if none was found, the root node solution is output.

The following describes the printout for each variable and constraint for both root node and final IP solution printout.

`Varbl` gives the name of variable j, for $j = 1, 2, \ldots, n$. If an **options** structure is supplied to nag_ip_bb, and the **crnames** member is assigned to an array of variable and constraint 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 (`FR` if neither bound is in the working set, `EQ` if a fixed variable, `LL` if on its lower bound, `UL` if on its upper bound, `TF` if temporarily fixed at its current value). If `Value` lies outside the upper or lower bounds by more than the feasibility tolerance, `State` will be `++` or `--` respectively.

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

`Lower Bound` is the lower bound l_j specified for the variable. (`None` indicates that $l_j \leq -$**inf_bound**, where **inf_bound** is the optional parameter.) For root node printout, $l_j =$ **bl**$[j-1]$; for IP solution printout, l_j is the lower bound imposed at the node which provided the IP solution.

`Upper Bound` is the upper bound u_j specified for the variable. (`None` indicates that $u_j \geq$ **inf_bound**.) For root node printout, $u_j =$ **bu**$[j-1]$; for IP solution printout, u_j is the upper bound imposed at the node which provided the IP solution.

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

`Residual` is the difference between the variable `Value` and the nearer of its bounds l_j and u_j.

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**$[n+j-1]$, l_j and u_j replaced by l_{n+i} and u_{n+i} respectively, and with the following change in the heading:

`Constr` gives the name of constraint i, $i = 1, 2, \ldots, m$. If an **options** structure is supplied to nag_ip_bb, and the **crnames** member is assigned to an array of variable and constraint names (see Section 8.2 for details), the name supplied in **crnames**$[n+i-1]$ is assigned to the constraint. Otherwise, a default name is assigned to the constraint.

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_ip_bb 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);
```

This section may 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_ip_bb. 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->node_prt** = **TRUE** then the results from the most recently solved node are provided through **st**. Note that **print_fun** will be called with **comm->node_prt** = **TRUE** only if **print_level** = **Nag_Iter**, **Nag_Soln_Iter** or **Nag_Soln_Root_Iter**. The following members of **st** are set:

node_num – Integer
 the current node number of the BB tree being investigated.

parent_node – Integer
 the parent node number of the current node.

node_status – Nag_NodeStatus
the status of the current node. The possible values of **node_status** and their meanings are as follows:

Nag_NS_NotBranched	the node has been solved but the branch cannot yet be eliminated from the search.
Nag_NS_Integer	an integer solution was found at this node. There is no need to search this branch further.
Nag_NS_Bounded	the objective value exceeds the upper bound on the optimal IP solution. There is no need to search this branch further.
Nag_NS_Infeasible	the problem was infeasible at this node. There is no need to search this branch further.
Nag_NS_Terminated	the iteration limit was exceeded at this node. The search has to be terminated prematurely for this branch.

objf – double
if **st->node_status** = **Nag_NS_NotBranched**, **Nag_NS_Integer** or **Nag_NS_Bounded**, then **objf** holds the objective value.

branch_index – Integer
the index in x of the variable chosen for branching.

x_lo – double
the lower bound on the branching variable.

x_up – double
the upper bound on the branching variable.

x_before – double
the non-integer value of the branching variable before the node was solved.

x_after – double
the value of the branching variable after the node was solved.

depth – Integer
the depth of the BB tree at the current node.

If **comm->rootnode_sol_prt** = **TRUE** then the solution of the root node is provided through **st**. Note that **print_fun** will be called with **comm->rootnode_sol_prt** = **TRUE** only if **print_level** = **Nag_Soln_Root** or **Nag_Soln_Root_Iter**. The following members of **st** are set:

endstate – Nag_EndState
the state of termination of the sub-problem solver at the root node. Some of these states result in immediate termination of the algorithm. If this is the case, then no valid solution is available. The other states allow the algorithm to proceed with the BB tree search. Possible values of endstate and their correspondence, if any, to the exit value of **fail.code** from nag_ip_bb are:

Value of **endstate**	Value of **fail.code**
Nag_Optimal	(BB search may proceed)
Nag_Deadpoint	(BB search may proceed)
Nag_Weakmin	(BB search may proceed)
Nag_Unbounded	**NE_MIP_ROOT_UNBOUNDED**
Nag_Infeasible	**NE_MIP_ROOT_INFEAS**
Nag_Too_Many_Iter	**NE_MIP_ROOT_MAX_ITER**
Nag_Hess_Too_Big	**NE_MIP_ROOT_HESS_TOO_BIG**

n – Integer
the number of variables.

m – Integer
the number of linear constraints.

objf – double
the value of the objective function.

x – double
the components **x**$[j-1]$ of the solution x, for $j = 1, 2, \ldots,$**st->n**.

ax – double
if **st->m** > 0, **ax**$[j-1]$ contains the components of the linear constraint vector, for $j = 1, 2, \ldots,$**st->m**.

state – Integer *
contains the status of the **st->n** variables and **st->m** general linear constraints. See Section 8.2 for a description of the possible status values.

lambda – double *
contains the **st->n** + **st->m** values of the Lagrange multipliers.

bl – double *
contains the **st->n** + **st->m** lower bounds on the variables.

bu – double *
contains the **st->n** + **st->m** upper bounds on the variables.

If **comm->sol_prt** = **TRUE** then the final IP solution is provided through **st**. Note that **print_fun** will be called with **comm->sol_prt** = **TRUE** only if **print_level** = **Nag_Soln**, **Nag_Soln_Root**, **Nag_Soln_Iter**, or **Nag_Soln_Root_Iter**. If no IP solution was found then the root node solution is available. The **endstate** member of **st** should be examined to determine the status of the solution. The following members of **st** are set:

endstate – Nag_EndState
the state of termination of nag_ip_bb. Possible values of **endstate** and their correspondence to the exit value of **fail.code** are shown below.

Value of **endstate**	Value of **fail.code**
Nag_MIP_Best_ISol or **Nag_MIP_Stop_First_ISol**	**NE_NOERROR**
Nag_MIP_No_ISol	**NE_MIP_NO_INT_SOL**
Nag_MIP_Root_Unbounded	**NE_MIP_ROOT_UNBOUNDED**
Nag_MIP_Root_Infeasible	**NE_MIP_ROOT_INFEAS**
Nag_MIP_Root_Max_Itn	**NE_MIP_ROOT_MAX_ITER**
Nag_MIP_Root_Big_Hess	**NE_MIP_ROOT_HESS_TOO_BIG**
Nag_MIP_Max_Itn_ISol	**NE_MIP_MAX_ITER_INT_SOL**
Nag_MIP_Max_Itn_No_ISol	**NE_MIP_MAX_ITER_NO_INT_SOL**
Nag_MIP_Big_Hess_ISol	**NE_MIP_HESS_TOO_BIG_INT_SOL**
Nag_MIP_Big_Hess_No_ISol	**NE_MIP_HESS_TOO_BIG_NO_INT_SOL**
Nag_MIP_Max_Nodes_ISol	**NE_MIP_MAX_NODES_INT_SOL**
Nag_MIP_Max_Nodes_No_ISol	**NE_MIP_MAX_NODES_NO_INT_SOL**
Nag_MIP_Max_Depth_ISol	**NE_MIP_MAX_DEPTH_INT_SOL**
Nag_MIP_Max_Depth_No_ISol	**NE_MIP_MAX_DEPTH_NO_INT_SOL**

n – Integer
the number of variables.

m – Integer
the number of linear constraints.

nnodes – Integer
the number of nodes examined during the BB tree search.

objf – double
the value of the objective function.

x – double
: the components $\mathbf{x}[j-1]$ of the solution x, for $j = 1, 2, \ldots,$**st->n**.

ax – double
: if **st->m** > 0, $\mathbf{ax}[j-1]$ contains the components of the linear constraint vector, for $j = 1, 2, \ldots,$**st->m**.

state – Integer *
: contains the status of the **st->n** variables and **st->m** general linear constraints. See Section 8.2 for a description of the possible status values.

lambda – double *
: contains the **st->n** + **st->m** values of the Lagrange multipliers.

bl – double *
: contains the **st->n** + **st->m** lower bounds on the variables.

bu – double *
: contains the **st->n** + **st->m** upper bounds on the variables.

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

rootnode_sol_prt – Boolean
: will be **TRUE** when the print function is called with the solution of the root node.

node_prt – Boolean
: will be **TRUE** when the print function is called with the result of the most recently solved node.

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

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_ip_bb 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_USER_STOP
: User requested termination, user flag value = ⟨*value*⟩.

This exit occurs if the user sets **comm->flag** to a negative value in **qphess**. If **fail** is supplied the value of **fail.errnum** will be the same as the user's setting of **comm->flag**.

NE_INT_ARG_LT
: On entry, **n** must not be less than 1: **n** = ⟨*value*⟩.
On entry, **m** must not be less than 0: **m** = ⟨*value*⟩.

NE_2_INT_ARG_LT
: On entry, **tda** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These parameters must satisfy **tda** $\geq$ **n**.
On entry, **tdh** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These parameters must satisfy **tdh** $\geq$ **n**.
On entry, **tdh** = ⟨*value*⟩ while **options.hrows** = ⟨*value*⟩. These parameters must satisfy **tdh** $\geq$ **hrows**.

NE_OPT_NOT_INIT
: Options structure not initialized.

NE_BAD_PARAM
: On entry parameter **options.prob** had an illegal value.
On entry parameter **options.print_level** had an illegal value.
On entry parameter **options.nodsel** had an illegal value.
On entry parameter **options.varsel** had an illegal value.
On entry parameter **options.branch_dir** had an illegal value.

NE_INVALID_INT_RANGE_1

Value ⟨*value*⟩ given to **options.max_iter** is not valid. Correct range is **max_iter** ≥ 0.
Value ⟨*value*⟩ given to **options.max_nodes** is not valid. Correct range is **max_nodes** = **ALL_NODES** or **max_nodes** ≥ 1.
Value ⟨*value*⟩ given to **options.max_depth** is not valid. Correct range is **max_depth** ≥ 2.
Value ⟨*value*⟩ given to **options.hrows** is not valid. Correct range is **n** $\geq$ **hrows** ≥ 0.
Value ⟨*value*⟩ given to **options.max_df** is not valid. Correct range is **n** $\geq$ **max_df** ≥ 1.

NE_INVALID_REAL_RANGE_FF

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

NE_INVALID_REAL_RANGE_F

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

NE_CVEC_NULL

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

NE_H_NULL

options.prob = ⟨*value*⟩, **qphess** is NULL but argument **h** is also NULL. If the default function for **qphess** is to be used for this problem then an array must be supplied in parameter **h**.

NE_PRIORITY_NULL

options.varsel = **Nag_Use_Priority** but **options.priority** is NULL.

NE_NAME_TOO_LONG

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

NE_BOUND

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

NE_BOUND_LCON

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

NE_ALLOC_FAIL

Memory allocation failed.

NE_MIP_ROOT_UNBOUNDED

The root node of the BB tree appears to be unbounded.

See Section 10 for advice.

NE_MIP_ROOT_INFEAS

The root node of the BB tree is infeasible.

A feasible point could not be found for the original LP or QP problem, i.e., it was not possible to satisfy all the constraints to within the feasibility tolerance (determined by optional parameter **feas_tol**). If the data for the constraints are accurate only to the absolute precision σ, the user should ensure that the value of the feasibility tolerance is greater than σ. For example, if all elements of A are of order unity and are accurate only to three decimal places, the feasibility tolerance should be at least 10^{-3} (see Section 10).

NE_MIP_ROOT_MAX_ITER

The maximum number of iterations, ⟨*value*⟩, was performed before normal termination occurred for the root node of the BB tree.

The maximum number of iterations (determined by optional parameter **max_iter**) was reached before normal termination occurred for the original LP or QP problem (see Section 10).

NW_MIP_NO_INT_SOL

No feasible IP solution was found, i.e., it was not possible to satisfy all the integer variables to within optional parameter **int_tol**.

It may be appropriate to increase **int_tol** and rerun nag_ip_bb.

NW_MIP_MAX_ITER_INT_SOL

The IP solution found may not be the optimum. The search had to be terminated in at least one branch of the BB tree because the iteration limit was reached.

It was not possible to solve at least one node of the BB tree, which means that the tree search could not be completed. An IP solution was found but a better one may be present in the unsearched portion of the tree. See Section 10 for more information.

NW_MIP_MAX_ITER_NO_INT_SOL

No IP solution was found but the search had to be terminated in at least one branch of the BB tree because the iteration limit was reached.

It was not possible to solve at least one node of the BB tree, which means that the tree search could not be completed. No IP solution was found but one may be present in the unsearched portion of the tree. See Section 10 for more information.

NW_MIP_MAX_NODES_INT_SOL

The IP solution found is the best for the number of nodes (as determined by optional parameter **max_nodes**) investigated in the BB tree.

Increase **max_nodes** and rerun nag_ip_bb. The IP objective obtained should be assigned to **options.int_obj_bound** to aid the BB tree search in the repeated run.

NW_MIP_MAX_NODES_NO_INT_SOL

No integer solution was found for the number of nodes (as determined by **options.max_nodes**) investigated in the BB tree.

Increase **max_nodes** and rerun nag_ip_bb.

NW_MIP_MAX_DEPTH_INT_SOL

An IP solution was found but the search has been terminated because the maximum allowed tree depth (as determined by optional parameter **max_depth**) has been reached.

Increase **max_depth** and rerun nag_ip_bb. The IP objective obtained should be assigned to **options.int_obj_bound** to aid the BB tree search in the repeated run.

NW_MIP_MAX_DEPTH_NO_INT_SOL

The maximum allowed tree depth (as determined by optional parameter **max_depth**) has been reached before any integer solution has been found.

Increase **max_depth** and rerun nag_ip_bb.

NE_MIP_ROOT_HESS_TOO_BIG

Reduced Hessian exceeds assigned dimension at root node.
options.max_df = ⟨*value*⟩.

This error can only occur with MIQP problems. Whilst attempting to solve the root node, the QP algorithm needed to expand the reduced Hessian when it was already at its maximum dimension, as specified by the optional parameter **max_df**.

The value of the parameter **max_df** is too small. Rerun nag_ip_bb with a larger value.

NE_MIP_HESS_TOO_BIG_INT_SOL

Reduced Hessian exceeds assigned dimension during BB tree search.
options.max_df = ⟨*value*⟩. An IP solution was found.

This error can only occur with MIQP problems. Whilst attempting to solve a node during the BB tree search, the QP algorithm needed to expand the reduced Hessian when it was already at its maximum dimension, as specified by the optional parameter **max_df**. No further nodes were examined. An IP solution was found but it may not be optimal.

The value of the parameter **max_df** is too small. Rerun nag_ip_bb with a larger value. The IP objective obtained should be assigned to **options.int_obj_bound** to aid the BB tree search in the repeated run.

NE_MIP_HESS_TOO_BIG_NO_INT_SOL

Reduced Hessian exceeds assigned dimension during BB tree search.
options.max_df = ⟨*value*⟩. No IP solution was found.

This error can only occur with MIQP problems. Whilst attempting to solve a node during the BB tree search, the QP algorithm needed to expand the reduced Hessian when it was already at its maximum dimension, as specified by the optional parameter **max_df**. No further nodes were examined. No IP solution was found amongst the nodes examined.

The value of the parameter **max_df** is too small. Rerun nag_ip_bb with a larger value.

NW_OVERFLOW_WARN

Serious ill-conditioning in the working set after adding constraint ⟨*value*⟩. Overflow may occur in subsequent iterations.

If overflow occurs preceded by this warning then serious ill-conditioning has probably occurred in the working set when adding a constraint during the solution of a node in the BB tree. It may be possible to avoid the difficulty by increasing the magnitude of the optional parameter **feas_tol** and rerunning the program. If the problem recurs even after this change, see Section 10.

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

The root node may not have an optimum solution, i.e., nag_ip_bb terminates with **fail.code** = **NE_MIP_ROOT_UNBOUNDED**, **NE_MIP_ROOT_INFEAS**, **NE_MIP_ROOT_MAX_ITER**, **NE_MIP_ROOT_HESS_TOO_BIG** or overflow may occur. In this case, the user is recommended to relax the integer restrictions of the problem and try to find the optimum LP or QP solution by using nag_opt_lp (e04mfc) (for LP) or nag_opt_qp (e04nfc) (for QP) instead.

In the BB method, it is possible for a node to terminate without finding a solution. For example, this may occur due to the number of iterations exceeding the maximum allowed. Therefore the BB tree search for that particular branch cannot be continued and if an IP solution is found, the final solution reported is not necessarily the optimum IP solution (**fail.code** = **NW_MIP_MAX_ITER_INT_SOL**). Similarly, if no IP solution is found, it is not necessarily the case that no IP solution exists (**fail_code** = **NW_MIP_MAX_ITER_NO_INT_SOL**).

10.1 Accuracy

The function 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

Dakin R J (1965) A tree search algorithm for mixed integer programming problems *Comput. J.* **8** 250–255.

Mitra G (1973) Investigation of some branch and bound strategies for the solution of mixed integer linear programs *Math. Programming* **4** 155–170.

Taha H A (1987) *Operations Research: An Introduction.* Macmillan.

Williams H P (1993) *Model Building in Mathematical Programming.* John Wiley.

12. See Also

nag_opt_lp (e04mfc)
nag_opt_qp (e04nfc)
nag_ip_mps_read (h02buc)
nag_ip_init (h02xxc)
nag_ip_read (h02xyc)
nag_ip_free (h02xzc)

13. Example 2

One of the applications of integer programming is to the so-called diet problem. Given the nutritional content of a selection of foods, the cost of each food, the amount available of each food and the consumer's minimum daily energy requirements, the problem is to find the cheapest combination. This gives rise to the following problem:

minimize

$$c^T x \text{ subject to } Ax \geq b, \ 0 \leq x \leq u,$$

where

$$c = (3 \quad 24 \quad 13 \quad 9 \quad 20 \quad 19)^T, \quad x = (x_1, x_2, x_3, x_4, x_5, x_6)^T \quad \text{is integer,}$$

$$A = \begin{pmatrix} 110 & 205 & 160 & 160 & 420 & 260 \\ 4 & 32 & 13 & 8 & 4 & 14 \\ 2 & 12 & 54 & 285 & 22 & 80 \end{pmatrix}, \quad b = \begin{pmatrix} 2000 \\ 55 \\ 800 \end{pmatrix} \quad \text{and}$$

$$u = (4 \quad 3 \quad 2 \quad 8 \quad 2 \quad 2)^T.$$

The rows of A correspond to energy, protein and calcium and the columns of A correspond to oatmeal, chicken, eggs, milk, pie and bacon respectively.

The following program solves the above problem to obtain the optimal integer solution and then examines the effect of decreasing the energy required to 1970 units. The example involves a number of calls to nag_ip_bb illustrating the use of some of the optional parameters.

The data is read and the options structure initialised. All options are left at their default values except: the **crnames** member is assigned to the local `char *` array, `crnames`, the elements of which point to strings containing the variable and constraint names; and **print_level** is set to **Nag_Soln**.

nag_ip_bb is called to obtain the optimal IP solution of the problem, and then the lower bound on the minimum energy constraint (i.e., the first general constraint) is reduced. Since the problem is now less constrained than the original IP problem, the objective function value returned in `objf` from the original problem provides an upper bound for the objective of the optimal IP solution of the modified problem. Optional parameter **int_obj_bound** is initialised to this value with a small number added to ensure that it is a strict upper bound on the optimal objective of the modified problem. Also, the optional parameter **nodsel** is set to **Nag_Deep_Search** to modify the way nag_ip_bb selects nodes during the tree search. The results from this show that the value assigned to **int_obj_bound** allow a number of nodes to be cut off (indicated by `CO` in the printout) before the first IP solution is found.

Next, the effect of supplying branching directions is illustrated. The optional parameter **branch_dir** is set to **Nag_Branch_InitX** to instruct nag_ip_bb to branch according to the values of the integer variables provided in the initial **x** parameter. In this case **x** contains the optimal IP solution from the last call of nag_ip_bb. The results show that these values allow nag_ip_bb to find and confirm the optimal IP solution quickly.

The final two calls to nag_ip_bb show its use in solving an MIQP problem. First, nag_ip_bb is called with the **intvar** parameter set to an array `intvar2` which specifies all variables to be non-integer.

This solves the root LP problem of the adjusted diet problem (as solved in the previous three calls to nag_ip_bb). Let x^* be the solution to this LP problem. Then, retaining the same constraints, the linear objective is replaced by the quadratic objective

$$\sum_{i=1}^{n}(x_i - x_i^*)^2 - \sum_{i=1}^{n}(x_i^*)^2 = x^T x - 2(x^*)^T x$$

which measures, to within a constant, the sum of squares deviation of x from x^*. That is, the problem is to find the IP solution which most closely approximates (in the least-squares sense) the LP solution. Before solving this problem, the memory assigned to the pointers in the **options** structure is freed by nag_ip_free (h02xzc) and the structure is reinitialised by nag_ip_free (h02xzc). Then optional parameter **prob** is set to **Nag_MIQP2** and **crnames** is assigned as before; otherwise, default options are used. The quadratic term of the objective is supplied via the function `qphess` which does not require explicit storage for the matrix H. nag_ip_bb is called to solve the MIQP problem, and finally nag_ip_free (h02xzc) is called to free the memory in **options**.

13.1. Program Text

```
static void ex2()
{
  /* Local variables */
  double a[MAXM][MAXN], cvec[MAXN], bl[MAXBND], bu[MAXBND];
  double x[MAXN];
  double objf, red_bnd;
  Integer i, j, is_int;
  Integer m,  n, nbnd, tda;
  char *crnames[MAXBND];
  char names[9*MAXBND];
  Boolean intvar[MAXN], intvar2[MAXN];
  static NagError fail;
  Nag_H02_Opt options;

  Vprintf("\nExample 2: some options set.\n");
  Vscanf(" %*[^\n]"); /* Skip heading */

  fail.print = TRUE;
  tda = MAXN;

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

  /* Read names */
  Vscanf(" %*[^\n]");
  nbnd = n+m;
  for (i = 0; i < nbnd; ++i)
    {
      Vscanf("%s", &names[9*i]);
      crnames[i] = &names[9*i];
    }
  /* Read objective coefficients */
  Vscanf(" %*[^\n]");
  for (i = 0; i < n; ++i)
    Vscanf("%lf", &cvec[i]);

  /* Read the matrix coefficients */
  Vscanf(" %*[^\n]");
  for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
      Vscanf("%lf",&a[i][j]);

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

```
/* Read which variables are integer */
Vscanf(" %*[^\n]");
for (i = 0; i < n; ++i)
  {
    Vscanf("%ld", &is_int);
    /* is_int = 1 if integer variable, 0 if not */
    intvar[i] = is_int ? TRUE : FALSE;
  }

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

h02xxc(&options); /* Initialise options structure */
options.crnames = crnames;
options.print_level = Nag_Soln;
h02bbc(n, m, (double *)a, tda, bl, bu, intvar, cvec, (double *)0, 0,
       (void*)0, x, &objf, &options, NAGCOMM_NULL, &fail);

/* Now solve a related problem obtained by reducing lower
   bound on a constraint */

/* Read amount to reduce lower bound on constraint 1 by */
Vscanf(" %*[^\n]");
Vscanf("%lf", &red_bnd);
bl[n] -= red_bnd;

Vprintf("\nSolve modified problem - use different tree search.\n");
Vprintf("----------------------------------------------------\n");

options.list = FALSE;
if (red_bnd > 0.0)
  {
    /* We have a valid bound for the objective since this problem
       is less constrained than first one */
    options.int_obj_bound = objf + 1.0e-3;
  }
options.nodsel = Nag_Deep_Search;
options.print_level = Nag_Iter;

Vprintf("***Set options.list = FALSE\n");
Vprintf("***Set options.int_obj_bound = %15.7e\n", options.int_obj_bound);
Vprintf("***Set options.nodsel = Nag_Deep_Search\n");
Vprintf("***Set options.print_level = Nag_Iter\n");

h02bbc(n, m, (double *)a, tda, bl, bu, intvar, cvec, (double *)0, 0,
       (void*)0, x, &objf, &options, NAGCOMM_NULL, &fail);
Vprintf("\n***IP objective value = %15.7e\n", objf);

Vprintf("\n\nIllustrate effect of supplying branching directions.\n");
Vprintf("-----------------------------------------------------\n\n");

options.branch_dir = Nag_Branch_InitX;
Vprintf("***Set options.branch_dir = Nag_Branch_InitX\n");

h02bbc(n, m, (double *)a, tda, bl, bu, intvar, cvec, (double *)0, 0,
       (void*)0, x, &objf, &options, NAGCOMM_NULL, &fail);
Vprintf("\n***IP objective value = %15.7e\n", objf);
h02xzc(&options, "", NAGERR_DEFAULT);

/* Finally, illustrate solution of an MIQP problem
   - we find the IP solution which is closest in
   least-squares sense to the root node LP solution
   of BB tree */

Vprintf("\n\nObtain solution of root LP problem.\n");
Vprintf("-----------------------------------\n\n");
```

```
  /* Set all variables non-integer to obtain LP solution */
  for (i = 0; i < n; ++i)
    intvar2[i] = 0;

  options.print_level = Nag_NoPrint;
  Vprintf("***Printout suppressed: options.print_level = Nag_NoPrint\n");

  h02bbc(n, m, (double *)a, tda, bl, bu, intvar2, cvec, (double *)0, 0,
         (void*)0, x, &objf, &options, NAGCOMM_NULL, &fail);
  Vprintf("***LP objective value = %15.7e\n", objf);

  /* Set linear part of solution */
  for (i = 0; i < n; ++i)
    cvec[i] = -2.0*x[i];

  /* Re-initialise options structure */
  h02xzc(&options, "", NAGERR_DEFAULT);
  h02xxc(&options);
  options.crnames = crnames;

  options.list = TRUE;
  options.prob = Nag_MIQP2;

  Vprintf("\n\nFinally, solve a related MIQP problem.\n");
  Vprintf("-------------------------------------\n");

  h02bbc(n, m, (double *)a, tda, bl, bu, intvar, cvec, (double *)0, 0,
         qphess, x, &objf, &options, NAGCOMM_NULL, &fail);

  h02xzc(&options, "", NAGERR_DEFAULT);
} /* ex2 */
```

13.2. Program Data

```
Data for example 2

Values of m, n
  3  6

Variable and constraint names
OATMEAL  CHICKEN  EGGS  MILK  PIE  BACON
ENERGY  PROTEIN  CALCIUM

Objective coefficients, cvec
 3.0  24.0  13.0  9.0  20.0  19.0

Constraint matrix a
 110.0   205.0   160.0   160.0   420.0   260.0
   4.0    32.0    13.0     8.0     4.0    14.0
   2.0    12.0    54.0   285.0    22.0    80.0

Lower bounds
 0.0  0.0  0.0  0.0  0.0  0.0 2000.0  55.0  800.0

Upper bounds
 4.0  3.0  2.0  8.0  2.0  2.0  1.0e+20  1.0e+20  1.0e+20

Integer variables (1 if integer, 0 if not)
  1  1  1  1  1  1

Initial estimate of x
  0.0  0.0  0.0  0.0  0.0  0.0

Reduction in first constraint lower bound for re-run
 30.0
```

13.3. Program Results

```
Example 2: some options set.

Parameters to h02bbc
--------------------

Linear constraints............  3    Number of variables...........  6
Number of integer variables...  6

prob.................... Nag_MILP
feas_tol................ 1.05e-08    machine precision....... 1.11e-16
inf_bound............... 1.00e+20    max_iter................       50
first_soln.............     FALSE    max_depth...............       10
max_nodes...............ALL_NODES    int_tol................. 1.00e-05
int_obj_bound........... 1.00e+20    soln_tol................ 1.05e-08
nodsel..........Nag_MinObj_Search    varsel..............Nag_First_Int
branch_dir........Nag_Branch_Left    crnames..................supplied
print_level...........   Nag_Soln
outfile.................   stdout

Memory allocation:
lower...................      Nag
upper...................      Nag
state...................      Nag
lambda..................      Nag

Final solution:

Varbl   State    Value       Lower Bound  Upper Bound    Lagr Mult    Residual

OATMEAL   EQ   4.00000e+00   4.0000e+00   4.0000e+00    3.000e+00   0.000e+00
CHICKEN   LL   0.00000e+00   0.0000e+00   3.0000e+00    2.400e+01   0.000e+00
EGGS      LL   0.00000e+00   0.0000e+00   2.0000e+00    1.300e+01   0.000e+00
MILK      LL   5.00000e+00   5.0000e+00   8.0000e+00    9.000e+00   0.000e+00
PIE       EQ   2.00000e+00   2.0000e+00   2.0000e+00    2.000e+01   0.000e+00
BACON     LL   0.00000e+00   0.0000e+00   2.0000e+00    1.900e+01   0.000e+00

Constr  State    Value       Lower Bound  Upper Bound    Lagr Mult    Residual

ENERGY    FR   2.08000e+03   2.0000e+03      None       0.000e+00   8.000e+01
PROTEIN   FR   6.40000e+01   5.5000e+01      None       0.000e+00   9.000e+00
CALCIUM   FR   1.47700e+03   8.0000e+02      None       0.000e+00   6.770e+02

Exit from branch and bound tree search after 27 nodes.

Optimal IP solution found.

Final IP objective value =   9.7000000e+01


Solve modified problem - use different tree search.
---------------------------------------------------
***Set options.list = FALSE
***Set options.int_obj_bound =   9.7001000e+01
***Set options.nodsel = Nag_Deep_Search
***Set options.print_level = Nag_Iter

  Node Parent     Obj     Varbl     Value     Lower     Upper      Value Depth
   No   Node     Value   Chosen    Before     Bound     Bound      After
    1         9.081e+01
    2      1  9.165e+01       4  4.31e+00  0.00e+00  4.00e+00   4.00e+00     1
    3      1  9.176e+01       4  4.31e+00  5.00e+00  8.00e+00   5.00e+00     1
    4      2  9.206e+01       6  1.92e-01  0.00e+00  0.00e+00   0.00e+00     2
    5      2  9.519e+01       6  1.92e-01  1.00e+00  2.00e+00   1.00e+00     2
    6      4  9.385e+01       3  3.13e-01  0.00e+00  0.00e+00   0.00e+00     3
    7      4  9.481e+01       3  3.13e-01  1.00e+00  2.00e+00   1.00e+00     3
    8      6 Infeasible       2  2.44e-01  0.00e+00  0.00e+00   2.44e-01     4
    9      6  1.033e+02 CO    2  2.44e-01  1.00e+00  3.00e+00   1.00e+00     4
   10      7  9.606e+01       4  3.31e+00  0.00e+00  3.00e+00   3.00e+00     4
```

```
   11     7  9.576e+01        4  3.31e+00  4.00e+00  4.00e+00   4.00e+00     4
   12    10  9.785e+01 CO     3  1.31e+00  1.00e+00  1.00e+00   1.00e+00     5
   13    10  9.881e+01 CO     3  1.31e+00  2.00e+00  2.00e+00   2.00e+00     5
   14    11  1.116e+02 CO     5  1.74e+00  0.00e+00  1.00e+00   1.00e+00     5
   15    11  9.800e+01 CO     5  1.74e+00  2.00e+00  2.00e+00   2.00e+00     5
   16     5  1.039e+02 CO     4  2.69e+00  0.00e+00  2.00e+00   2.00e+00     3
   17     5  9.562e+01        4  2.69e+00  3.00e+00  4.00e+00   3.00e+00     3
   18    17  1.023e+02 CO     5  1.88e+00  0.00e+00  1.00e+00   1.00e+00     4
   19    17  9.664e+01        5  1.88e+00  2.00e+00  2.00e+00   2.00e+00     4
   20    19  9.838e+01 CO     1  3.55e+00  0.00e+00  3.00e+00   3.00e+00     5
   21    19  9.800e+01 CO     1  3.55e+00  4.00e+00  4.00e+00   4.00e+00     5
   22     3  9.444e+01        5  1.74e+00  0.00e+00  1.00e+00   1.00e+00     2
   23     3  9.400e+01        5  1.74e+00  2.00e+00  2.00e+00   2.00e+00     2
   *** Integer Solution ***


***IP objective value =   9.4000000e+01


Illustrate effect of supplying branching directions.
----------------------------------------------------

***Set options.branch_dir = Nag_Branch_InitX

  Node Parent     Obj     Varbl     Value     Lower     Upper      Value Depth
   No   Node     Value   Chosen    Before     Bound     Bound      After
    1        9.081e+01
    2     1  9.176e+01        4  4.31e+00  5.00e+00  8.00e+00   5.00e+00     1
    3     1  9.165e+01        4  4.31e+00  0.00e+00  4.00e+00   4.00e+00     1
    4     2  9.400e+01        5  1.74e+00  2.00e+00  2.00e+00   2.00e+00     2
   *** Integer Solution ***

    5     2  9.444e+01 CO     5  1.74e+00  0.00e+00  1.00e+00   1.00e+00     2
    6     3  9.206e+01        6  1.92e-01  0.00e+00  0.00e+00   0.00e+00     2
    7     3  9.519e+01 CO     6  1.92e-01  1.00e+00  2.00e+00   1.00e+00     2
    8     6  9.385e+01        3  3.13e-01  0.00e+00  0.00e+00   0.00e+00     3
    9     6  9.481e+01 CO     3  3.13e-01  1.00e+00  2.00e+00   1.00e+00     3
   10     8 Infeasible        2  2.44e-01  0.00e+00  0.00e+00   2.44e-01     4
   11     8  1.033e+02 CO     2  2.44e-01  1.00e+00  3.00e+00   1.00e+00     4

***IP objective value =   9.4000000e+01


Obtain solution of root LP problem.
-----------------------------------

***Printout suppressed: options.print_level = Nag_NoPrint
***LP objective value =   9.0812500e+01


Finally, solve a related MIQP problem.
--------------------------------------

Parameters to h02bbc
--------------------

Linear constraints............  3    Number of variables...........  6
Number of integer variables...  6

prob....................Nag_MIQP2
feas_tol................ 1.05e-08    machine precision....... 1.11e-16
inf_bound............... 1.00e+20    max_iter................       50
rank_tol................ 1.11e-14    max_df..................        6
hrows...................        6
first_soln.............     FALSE    max_depth...............       10
max_nodes...............ALL_NODES    int_tol................. 1.00e-05
int_obj_bound........... 1.00e+20    soln_tol................ 1.05e-08
nodsel..........Nag_MinObj_Search    varsel..............Nag_First_Int
branch_dir........Nag_Branch_Left    crnames..................supplied
print_level.........Nag_Soln_Iter
```

```
outfile.................   stdout

Memory allocation:
lower...................      Nag
upper...................      Nag
state...................      Nag
lambda..................      Nag

  Node Parent     Obj     Varbl      Value      Lower      Upper       Value Depth
   No   Node     Value   Chosen     Before      Bound      Bound       After
    1       -3.860e+01
    2      1 -3.848e+01       4  4.31e+00  0.00e+00  4.00e+00   4.00e+00     1
    3      1 -3.813e+01       4  4.31e+00  5.00e+00  8.00e+00   5.00e+00     1
   *** Integer Solution ***

    4      2 -3.847e+01       2  7.58e-02  0.00e+00  0.00e+00   0.00e+00     2
    5      2 -3.750e+01 CO    2  7.58e-02  1.00e+00  3.00e+00   1.00e+00     2
    6      4 -3.846e+01       3  8.58e-02  0.00e+00  0.00e+00   0.00e+00     3
    7      4 -3.750e+01 CO    3  8.58e-02  1.00e+00  2.00e+00   1.00e+00     3
    8      6 Infeasible       6  1.92e-01  0.00e+00  0.00e+00   1.92e-01     4
    9      6 -3.750e+01 CO    6  1.92e-01  1.00e+00  2.00e+00   1.00e+00     4

Final solution:

Varbl   State    Value       Lower Bound  Upper Bound    Lagr Mult    Residual

OATMEAL   FR   4.00000e+00    0.0000e+00   4.0000e+00    0.000e+00   0.000e+00
CHICKEN   FR   0.00000e+00    0.0000e+00   3.0000e+00    0.000e+00   0.000e+00
EGGS      FR   0.00000e+00    0.0000e+00   2.0000e+00    0.000e+00   0.000e+00
MILK      LL   5.00000e+00    5.0000e+00   8.0000e+00    1.375e+00   0.000e+00
PIE       FR   2.00000e+00    0.0000e+00   2.0000e+00    0.000e+00   0.000e+00
BACON     FR   0.00000e+00    0.0000e+00   2.0000e+00    0.000e+00   0.000e+00

Constr  State    Value       Lower Bound  Upper Bound    Lagr Mult    Residual

ENERGY    FR   2.08000e+03    1.9700e+03      None       0.000e+00   1.100e+02
PROTEIN   FR   6.40000e+01    5.5000e+01      None       0.000e+00   9.000e+00
CALCIUM   FR   1.47700e+03    8.0000e+02      None       0.000e+00   6.770e+02

Exit from branch and bound tree search after 9 nodes.

Optimal IP solution found.

Final IP objective value =  -3.8125000e+01
```