NAG Library Function Document

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 <nagh.h>
void nag_ip_bb (Integer n, Integer m, const double a[], Integer tda,
     const double bl[], const double bu[], const Nag_Boolean intvar[],
     const double cvec[], const double h[], Integer tdh,
     void (*qphess)(Integer n, Integer jthcol, const double h[], Integer tdh,
          const double x[], double hx[], Nag_Comm *comm),
     double x[], double *objf, Nag_H02_Opt *options, Nag_Comm *comm,
     NagError *fail)
```

3 Description

nag_ip_bb (h02bbc) 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.

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

where A is an 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 $f(x) = c^{\mathrm{T}}x$ 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 (h02bbc) 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 $x_k \leq \left[x_k^*\right]$ and $x_k \geq \left[x_k^*\right] + 1$ respectively, where $\left[x_k^*\right]$ 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 12.

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 (h02bbc) 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	$c^{\mathrm{T}}x$	Not applicable
MIQP1	$\frac{1}{2}x^{\mathrm{T}}Hx$	symmetric
MIQP2	$c^{\mathrm{T}}x+\frac{1}{2}x^{\mathrm{T}}Hx$	symmetric
MIQP3	$\frac{1}{2}x^{\mathrm{T}}H^{\mathrm{T}}Hx$	m by n upper trapezoidal
MIQP4	$c^{\mathrm{T}}x+\frac{1}{2}x^{\mathrm{T}}H^{\mathrm{T}}Hx$	m by n upper trapezoidal

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 (h02bbc) does not check if the Hessian is positive (semi-)definite. This provides for the possibility that you have 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, you may wish to use nag_ip_bb (h02bbc) 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 you wish 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_dsyev (f08fac)): the Hessian is positive semidefinite if and only if all of its eigenvalues are ≥ 0.

3.2 Maximization Problems

nag_ip_bb (h02bbc) 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 argument 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 (h02bbc) does not allow you to reverse the sign of the quadratic objective term for these problem types.

4 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, New York

Williams H P (1993) *Model Building in Mathematical Programming* (3rd Edition) Wiley

5 Arguments

1: **n** – Integer *Input*

On entry: n, the number of variables.

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

2: **m** – Integer *Input*

On entry: m, the number of general linear constraints.

Constraint: $\mathbf{m} \geq 0$.

3: **a**[**m** × **tda**] – const double *Input*

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

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

If $\mathbf{m} = 0$, the array **a** is not referenced and may be **NULL**.

4: **tda** – Integer *Input*

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

Constraint: if $\mathbf{m} > 0$, $\mathbf{tda} \geq \mathbf{n}$

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

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

On entry: **bl** must contain the lower bounds and **bu** the upper bounds, for all the constraints in the following order. The first n elements of each array must contain the bounds on the variables, and the next m elements the bounds for the general linear constraints (if any). To specify a nonexistent lower bound (i.e., $l_j = -\infty$), set **bl**$[j-1] \leq -$**options.inf_bound**, and to specify a nonexistent upper bound (i.e., $u_j = +\infty$), set **bu**$[j-1] \geq$ **options.inf_bound**, where **options.inf_bound** is one of the optional parameters (default value 10^{20}, see Section 11.2). To specify the jth constraint as an equality, set **bl**$[j-1] =$ **bu**$[j-1] = \beta$, say, where $|\beta| <$ **options.inf_bound**.

Constraint: **bl**$[j] \leq$ **bu**$[j]$, for $j = 0, 1, \ldots, \mathbf{n} + \mathbf{m} - 1$.

7: **intvar**[**n**] – const Nag_Boolean *Input*

On entry: 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 (h02bbc) solves a single LP or QP problem (depending on the problem type as specified by the optional parameter **options.prob**, see Section 11.2).

Constraint: **intvar**$[j] = 0$ or 1, for $j = 0, 1, \ldots, \mathbf{n} - 1$.

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

On entry: 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 **options**.**prob**, see Section 11.2.

If the problem is of type MIQP1 or MIQP3, **cvec** is not referenced and may be **NULL**.

9: **h**[**n** × **tdh**] – const double *Input*

On entry: **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 **NULL**.

The number of rows of **h** is denoted by n_H and its default value is equal to n. (The optional parameter **options**.**hrows** may be used to specify a value of $n_H < n$; see Section 11.2).

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 nonsingular 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, you need not use **h** to store H explicitly (see the specification of function **qphess**).

10: **tdh** – Integer *Input*

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

Constraint: **tdh** $\geq$ **n** or at least the value of the optional parameter **options**.**hrows** if it is set. This constraint is enforced only for problems of type MIQP in which the **qphess** argument is null.

11: **qphess** – function, supplied by the user *External Function*

In general, you 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 (h02bbc). The algorithm of nag_ip_bb (h02bbc) requires only the product of H and a vector x; and in some cases you 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, const double h[], Integer tdh,
     const double x[], double hx[], Nag_Comm *comm)
```

1: **n** – Integer *Input*

On entry: n, the number of variables.

2: **jthcol** – Integer *Input*

On entry: **jthcol** specifies whether or not the vector x is a column of the identity matrix.

jthcol $= j > 0$
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**.

jthcol $= 0$
x has no special form.

3: **h**[**n** × **tdh**] – const double *Input*

On entry: the matrix H of the QP objective function. The matrix element H_{ij} is contained in **h**$[(i-1) \times tdh + j - 1]$ 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 Section 10.) The arguments **h** and **tdh** may then refer to any convenient array.

4: **tdh** – Integer *Input*

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

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

On entry: the vector x.

6: **hx**[**n**] – double *Output*

On exit: the product Hx.

7: **comm** – Nag_Comm *

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

flag – Integer *Input/Output*

On entry: **qphess** is called with **comm**→**flag** set to a non-negative number.

On exit: if **qphess** resets **comm**→**flag** to some negative number then nag_ip_bb (h02bbc) will terminate immediately with the error indicator NE_USER_STOP. If **fail** is supplied to nag_ip_bb (h02bbc), **fail.errnum** will be set to your setting of **comm**→**flag**.

first – Nag_Boolean *Input*

On entry: will be set to Nag_TRUE on the first call to **qphess** and Nag_FALSE for all subsequent calls.

nf – Integer *Input*

On entry: 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 (h02bbc) these pointers may be allocated memory and initialized with various quantities for use by **qphess** when called from nag_ip_bb (h02bbc).

Note: **qphess** should be tested separately before being used in conjunction with nag_ip_bb (h02bbc). The input arrays **h** and **x** must **not** be changed by **qphess**.

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

On entry: an initial estimate of the solution of the first sub-problem (the problem as described in Section 3).

If optional parameter **options.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 11.2 for details.

On exit: 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 **options.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 (h02bbc) was unable to investigate all of the nodes. See Section 6 for more details.

13: **objf** – double * *Output*

On exit: 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.

14: **options** – Nag_H02_Opt * *Input/Output*

On entry/exit: a pointer to a structure of type Nag_H02_Opt whose members are optional parameters for nag_ip_bb (h02bbc). These structure members offer the means of adjusting some of the argument values of the algorithm and on output will supply further details of the results. A description of the members of **options** is given below in Section 11.

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 (h02bbc).

If any of these optional parameters are required then the structure **options** should be declared and initialized by a call to nag_ip_init (h02xxc) and supplied as an argument to nag_ip_bb (h02bbc). However, if the optional parameters are not required the NAG defined null pointer, `H02_DEFAULT`, can be used in the function call.

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

Note: **comm** is a NAG defined type (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

On entry/exit: 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 11.3.1 for details. If you do not need to make use of this communication feature the null pointer `NAGCOMM_NULL` may be used in the call to nag_ip_bb (h02bbc); **comm** will then be declared internally for use in calls to user-supplied functions.

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

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

5.1 Description of Printed Output

Intermediate and final results are printed out by default. The level of printed output can be controlled with the structure member **options.print_level** (see Section 11.2).

The default, **options.print_level** = Nag_Soln_Iter, provides a single line of output at the end of each node and the final IP result. If nag_ip_bb (h02bbc) 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 (h02bbc).

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 (h02bbc), and the **options**.**crnames** member is assigned to an array of variable and constraint names (see Section 11.2 for details), the name supplied in **options**.**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 $l_j \leq -$**options**.**inf_bound**, where **options**.**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 in **bl**.)
`Upper Bound`	is the upper bound u_j specified for the variable. (`None` indicates that $u_j \geq$ **options**.**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 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, **options**.**crnames**$[j-1]$ replaced by **options**.**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.

6 Error Indicators and Warnings

NE_2_INT_ARG_LT

On entry, **tda** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These arguments must satisfy **tda** ≥ **n**.

On entry, **tdh** = ⟨*value*⟩ while **n** = ⟨*value*⟩. These arguments must satisfy **tdh** ≥ **n**.

On entry, **tdh** = ⟨*value*⟩ while **options**.**hrows** = ⟨*value*⟩. These arguments must satisfy **tdh** ≥ **options**.**hrows**.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

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

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

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

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

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

NE_BOUND

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

NE_BOUND_LCON

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

NE_CVEC_NULL

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

NE_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 argument **h**.

NE_INT_ARG_LT

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

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

NE_INTERNAL_ERROR

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

NE_INVALID_INT_RANGE_1

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

Value ⟨*value*⟩ given to **options**.**max_depth** is not valid. Correct range is **options**.**max_depth** ≥ 2.

Value ⟨*value*⟩ given to **options**.**max_df** is not valid. Correct range is **n** ≥ **options**.**max_df** ≥ 1.

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

Value ⟨*value*⟩ given to **options.max_nodes** is not valid. Correct range is **options.max_nodes** = ALL_NODES or **options.max_nodes** ≥ 1.

NE_INVALID_REAL_RANGE_F

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

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

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

NE_INVALID_REAL_RANGE_FF

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

Value ⟨*value*⟩ given to **options.rank_tol** is not valid. Correct range is 0.0 ≤ **options.rank_tol** < 1.0.

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 **options.max_df**. No further nodes were examined. An IP solution was found but it may not be optimal.
The value of the argument **options.max_df** is too small. Rerun nag_ip_bb (h02bbc) 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 **options.max_df**. No further nodes were examined. No IP solution was found amongst the nodes examined.
The value of the argument **options.max_df** is too small. Rerun nag_ip_bb (h02bbc) with a larger value.

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 **options.max_df**.
The value of the argument **options.max_df** is too small. Rerun nag_ip_bb (h02bbc) with a larger value.

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 **options.feas_tol**). If the data for the constraints are accurate only to the absolute precision σ, you 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 9).

NE_MIP_ROOT_MAX_ITER

The maximum number of iterations (determined by optional parameter **options.max_iter**) was reached before normal termination occurred for the original LP or QP problem (see Section 9).

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

NE_MIP_ROOT_UNBOUNDED

The root node of the BB tree appears to be unbounded.
See Section 12 for advice.

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_NOT_APPEND_FILE

Cannot open file ⟨*string*⟩ for appending.

NE_NOT_CLOSE_FILE

Cannot close file ⟨*string*⟩.

NE_OPT_NOT_INIT

Options structure not initialized.

NE_PRIORITY_NULL

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

NE_USER_STOP

User requested termination, user flag value = ⟨*value*⟩.
This exit occurs if you set **comm**→**flag** to a negative value in **qphess**. If **fail** is supplied the value of **fail**.**errnum** will be the same as your setting of **comm**→**flag**.

NE_WRITE_ERROR

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

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 **options**.**max_depth**) has been reached.
Increase **options**.**max_depth** and rerun nag_ip_bb (h02bbc). 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 **options**.**max_depth**) has been reached before any integer solution has been found.
Increase **options**.**max_depth** and rerun nag_ip_bb (h02bbc).

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 9 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 9 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 **options**.**max_nodes**) investigated in the BB tree.
Increase **options**.**max_nodes** and rerun nag_ip_bb (h02bbc). 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 **options**.**max_nodes** and rerun nag_ip_bb (h02bbc).

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 **options**.**int_tol**.
It may be appropriate to increase **options**.**int_tol** and rerun nag_ip_bb (h02bbc).

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 **options**.**feas_tol** and rerunning the program. If the problem recurs even after this change, see Section 9.

7 Accuracy

nag_ip_bb (h02bbc) implements a numerically stable active set strategy and returns solutions that are as accurate as the condition of the problem warrants on the machine.

8 Parallelism and Performance

nag_ip_bb (h02bbc) is not threaded in any implementation.

9 Further Comments

The root node may not have an optimum solution, i.e., nag_ip_bb (h02bbc) 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, you are 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 Example

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

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

where

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

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

$$u = \begin{pmatrix} 4 & 3 & 2 & 8 & 2 & 2 \end{pmatrix}^{\mathrm{T}}.$$

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 (h02bbc) illustrating the use of some of the optional parameters.

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

nag_ip_bb (h02bbc) 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 **options**.**int_obj_bound** is initialized 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 **options**.**nodsel** = Nag_Deep_Search to modify the way nag_ip_bb (h02bbc) selects nodes during the tree search. The results from this show that the value assigned to **options**.**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 **options**.**branch_dir** = Nag_Branch_InitX to instruct nag_ip_bb (h02bbc) to branch according to the values of the integer variables provided in the initial **x** argument. In this case **x** contains the optimal IP solution from the last call of nag_ip_bb (h02bbc). The results show that these values allow nag_ip_bb (h02bbc) to find and confirm the optimal IP solution quickly.

The final two calls to nag_ip_bb (h02bbc) show its use in solving an MIQP problem. First, nag_ip_bb (h02bbc) is called with the **intvar** argument 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 (h02bbc)). Let x^* be the solution to this LP problem. Then, retaining the same constraints, the linear objective is replaced by the quadratic objective

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

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 reinitialized by nag_ip_free (h02xzc). Then optional parameter **options**.**prob** = Nag_MIQP2 and **options**.**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 (h02bbc) is called to solve the MIQP problem, and finally nag_ip_free (h02xzc) is called to free the memory in **options**.

10.1 Program Text

```
/* nag_ip_bb (h02bbc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 *
 */

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

#ifdef __cplusplus
extern "C"
{
#endif
  static void NAG_CALL qphess(Integer n, Integer jthcol, const double h[],
                              Integer tdh, const double x[], double hx[],
                              Nag_Comm *comm);
#ifdef __cplusplus
}
#endif

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

int main(void)
{
  static double ruser[1] = { -1.0 };
  Integer exit_status = 0;
  Integer i, j, m, n, nbnd, tda;
  char **crnames = 0, *names = 0;
  double *a = 0, *bl = 0, *bu = 0, *cvec = 0, objf, red_bnd, *x = 0;
  Nag_Boolean *intvar = 0, *intvar2 = 0;
  char nag_enum_arg[40];
  Nag_Comm comm;
  Nag_H02_Opt options;
  NagError fail;

  INIT_FAIL(fail);

  printf("nag_ip_bb (h02bbc) Example Program Results\n");

  /* For communication with user-supplied functions: */
  comm.user = ruser;

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

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

```
  scanf("%" NAG_IFMT "%" NAG_IFMT "", &m, &n);
#endif
  nbnd = n + m;
  if (n >= 1 && m >= 0) {
    if (!(a = NAG_ALLOC(m * n, double)) ||
        !(cvec = NAG_ALLOC(n, double)) ||
        !(bl = NAG_ALLOC(nbnd, double)) ||
        !(bu = NAG_ALLOC(nbnd, double)) ||
        !(x = NAG_ALLOC(n, double)) ||
        !(intvar = NAG_ALLOC(n, Nag_Boolean)) ||
        !(intvar2 = NAG_ALLOC(n, Nag_Boolean)) ||
        !(crnames = NAG_ALLOC(nbnd, char *)) ||
        !(names = NAG_ALLOC(nbnd * 9, char)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
    tda = n;
  }
  else {
    printf("Invalid n or m.\n");
    exit_status = 1;
    return exit_status;
  }
  /* Read names */
#ifdef _WIN32
  scanf_s(" %*[^\n]");
#else
  scanf(" %*[^\n]");
#endif
  nbnd = n + m;
  for (i = 0; i < nbnd; ++i) {
#ifdef _WIN32
    scanf_s("%8s", &names[9 * i], 9);
#else
    scanf("%8s", &names[9 * i]);
#endif
    crnames[i] = &names[9 * i];
  }
  /* Read objective coefficients */
#ifdef _WIN32
  scanf_s(" %*[^\n]");
#else
  scanf(" %*[^\n]");
#endif
  for (i = 0; i < n; ++i)
#ifdef _WIN32
    scanf_s("%lf", &cvec[i]);
#else
    scanf("%lf", &cvec[i]);
#endif

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

  /* Read the bounds */
#ifdef _WIN32
  scanf_s(" %*[^\n]");
#else
```

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

  /* Read which variables are integer */
#ifdef _WIN32
  scanf_s(" %*[^\n]");
#else
  scanf(" %*[^\n]");
#endif
  for (i = 0; i < n; ++i) {
#ifdef _WIN32
    scanf_s("%39s", nag_enum_arg, (unsigned)_countof(nag_enum_arg));
#else
    scanf("%39s", nag_enum_arg);
#endif
    /* intvar = Nag_TRUE if integer variable, Nag_FALSE if not */
    intvar[i] = (Nag_Boolean) nag_enum_name_to_value(nag_enum_arg);
  }

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

  /* nag_ip_init (h02xxc).
   * Initialize option structure to null values
   */
  nag_ip_init(&options); /* Initialize options structure */
  options.crnames = crnames;
  options.list = Nag_FALSE;
  options.print_level = Nag_NoPrint;
  /* nag_ip_bb (h02bbc), see above. */
  fflush(stdout);
  nag_ip_bb(n, m, a, tda, bl, bu, intvar, cvec, (double *) 0, 0,
            NULLFN, x, &objf, &options, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_bb (h02bbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

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

  /* Read amount to reduce lower bound on constraint 1 by */
#ifdef _WIN32
```

```
  scanf_s(" %*[^\n]");
#else
  scanf(" %*[^\n]");
#endif
#ifdef _WIN32
  scanf_s("%lf", &red_bnd);
#else
  scanf("%lf", &red_bnd);
#endif
  bl[n] -= red_bnd;

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

  options.list = Nag_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.list = Nag_FALSE;
  options.print_level = Nag_NoPrint;

  printf("***Set options.list = Nag_FALSE\n");
  printf("***Set options.int_obj_bound = %16.7e\n", options.int_obj_bound);
  printf("***Set options.nodsel = Nag_Deep_Search\n");
  printf("***Set options.print_level = Nag_NoPrint\n");

  /* nag_ip_bb (h02bbc), see above. */
  fflush(stdout);
  nag_ip_bb(n, m, a, tda, bl, bu, intvar, cvec, (double *) 0, 0,
            NULLFN, x, &objf, &options, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_bb (h02bbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  printf("\n***IP objective value = %16.7e\n", objf);

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

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

  /* nag_ip_bb (h02bbc), see above. */
  fflush(stdout);
  nag_ip_bb(n, m, a, tda, bl, bu, intvar, cvec, (double *) 0, 0,
            NULLFN, x, &objf, &options, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_bb (h02bbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  printf("\n***IP objective value = %16.7e\n", objf);
  /* nag_ip_free (h02xzc).
   * Free NAG allocated memory from option structures
   */
  nag_ip_free(&options, "", &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_free (h02xzc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* 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 */
```

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

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

  options.list = Nag_FALSE;
  options.print_level = Nag_NoPrint;

  /* nag_ip_bb (h02bbc), see above. */
  nag_ip_bb(n, m, a, tda, bl, bu, intvar2, cvec, (double *) 0, 0,
            NULLFN, x, &objf, &options, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_bb (h02bbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  printf("***LP objective value = %16.7e\n", objf);

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

  /* Re-initialize options structure */
  /* nag_ip_free (h02xzc), see above. */
  fflush(stdout);
  nag_ip_free(&options, "", &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_free (h02xzc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  /* nag_ip_init (h02xxc), see above. */
  nag_ip_init(&options);
  options.crnames = crnames;

  options.list = Nag_FALSE;
  options.print_level = Nag_NoPrint;
  options.prob = Nag_MIQP2;

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

  /* nag_ip_bb (h02bbc), see above. */
  fflush(stdout);
  nag_ip_bb(n, m, a, tda, bl, bu, intvar, cvec, (double *) 0, 0,
            qphess, x, &objf, &options, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_bb (h02bbc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  printf("***MIQP2 objective value = %16.7e\n", objf);

  /* nag_ip_free (h02xzc), see above. */
  nag_ip_free(&options, "", &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ip_free (h02xzc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

END:
  NAG_FREE(a);
  NAG_FREE(cvec);
  NAG_FREE(bl);
  NAG_FREE(bu);
  NAG_FREE(x);
  NAG_FREE(intvar);
```

```
  NAG_FREE(intvar2);
  NAG_FREE(crnames);
  NAG_FREE(names);

  return exit_status;
}

static void NAG_CALL qphess(Integer n, Integer jthcol, const double h[],
                            Integer tdh, const double x[], double hx[],
                            Nag_Comm *comm)
{
  Integer i;
  if (comm->user[0] == -1.0) {
    printf("(User-supplied callback qphess, first invocation.)\n");
    fflush(stdout);
    comm->user[0] = 0.0;
  }
  /* In this qphess function the Hessian is defined implicitly */
  if (jthcol == 0) {
    for (i = 0; i < n; ++i)
      hx[i] = 2.0 * x[i];
  }
  else {
    for (i = 0; i < n; ++i)
      hx[i] = (i == jthcol - 1 ? 2.0 : 0.0);
  }
} /* qphess */
```

10.2 Program Data

```
nag_ip_bb (h02bbc) Example Program Data
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 (Nag_TRUE if integer, Nag_FALSE if not)
  Nag_TRUE  Nag_TRUE  Nag_TRUE  Nag_TRUE  Nag_TRUE  Nag_TRUE

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
```

10.3 Program Results

```
nag_ip_bb (h02bbc) Example Program Results

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

```
***Set options.print_level = Nag_NoPrint

***IP objective value =    9.4000000e+01


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

***Set options.branch_dir = Nag_Branch_InitX

***IP objective value =    9.4000000e+01


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

***LP objective value =    9.0812500e+01


Finally, solve a related MIQP problem.
--------------------------------------
(User-supplied callback qphess, first invocation.)
***MIQP2 objective value =   -3.8125000e+01
```

11 Optional Parameters

A number of optional input and output arguments to nag_ip_bb (h02bbc) are available through the structure argument **options**, type Nag_H02_Opt. a argument may be selected by assigning an appropriate value to the relevant structure member; those arguments not selected will be assigned default values. If no use is to be made of any of the optional parameters you should use the NAG defined null pointer, `H02_DEFAULT`, in place of **options** when calling nag_ip_bb (h02bbc) the default settings will then be used for all arguments.

Before assigning values to **options** directly the structure **must** be initialized by a call to the function nag_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 initialization of the **options** structure will be performed automatically if not already done. Any subsequent direct assignment to the **options** structure must **not** be preceded by initialization.

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

11.1 Optional Parameter Checklist and Default Values

For easy reference, the following list shows the members of **options** which are valid for nag_ip_bb (h02bbc) 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`	Nag_TRUE
`Nag_PrintType print_level`	**Nag_Soln_Iter**
`char outfile[80]`	`stdout`
`void (*print_fun)()`	**NULL**
`Integer max_iter`	$\max(50, 5(\mathbf{n}+\mathbf{m}))$
`Integer max_nodes`	`ALL_NODES`
`Boolean first_soln`	Nag_FALSE
`Integer max_depth`	$\max(10, 3\mathbf{n}/2)$
`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 $\mathbf{n}+\mathbf{m}$
`double *upper`	size $\mathbf{n}+\mathbf{m}$
`double *lambda`	size $\mathbf{n}+\mathbf{m}$
`Integer *state`	size $\mathbf{n}+\mathbf{m}$

11.2 Description of the Optional Parameters

prob – Nag_MIP_ProbType Default = Nag_MILP

On entry: specifies the type of objective function to be minimized during the optimality phase. The following are the five possible values of **options**.**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** + **tdh**] symmetric, **cvec** not referenced;

Nag_MIQP2 **h**[(**n**) × **tdh** + **tdh**] symmetric, **cvec**[**n**];

Nag_MIQP3 **h**[(**n**) × **tdh** + **tdh**] upper trapezoidal, **cvec** not referenced;

Nag_MIQP4 **h**[(**n**) × **tdh** + **tdh**] upper trapezoidal, **cvec**[**n**].

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

list – Nag_Boolean Default = Nag_TRUE

On entry: if **options**.**list** = Nag_TRUE the argument settings in the call to nag_ip_bb (h02bbc) will be printed.

print_level – Nag_PrintType Default = Nag_Soln_Iter

On entry: the level of results printout produced by nag_ip_bb (h02bbc). 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 11.3.

Constraint: **options.print_level** = Nag_NoPrint, Nag_Soln, Nag_Soln_Root, Nag_Iter, Nag_Soln_Iter or Nag_Soln_Root_Iter.

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

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

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

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

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

See Section 11.3.1 below for further details.

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

On entry: the limit on the number of iterations for each node.

Constraint: **options.max_iter** ≥ 0.

max_nodes – Integer Default = `ALL_NODES`

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

Constraints:

options.max_nodes > 0 or;
options.max_nodes = `ALL_NODES`.

first_soln – Nag_Boolean Default = Nag_FALSE

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

max_depth – Integer Default = $\max(10, 3\mathbf{n}/2)$

On entry: the maximum depth of the BB tree used for branch and bound.

Constraint: **options.max_depth** ≥ 2.

int_tol – double Default = 10^{-5}

On entry: the integer feasibility tolerance; i.e., an integer variable is considered to take an integer value if its violation does not exceed **options.int_tol**. For example, if the integer variable x_j is of order unity then x_j is considered to be integer if $(1 - \mathbf{options.int\_tol}) \leq x_j \leq (1 + \mathbf{options.int\_tol})$.

Constraint: **options.int_tol** > 0.0.

int_obj_bound – double Default = 10^{20}

On entry: specifies an initial bound on the optimum integer solution. You should supply a value for this argument only if a valid strict upper bound for the IP problem is available. Supplying too small a value will result in nag_ip_bb (h02bbc) 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 12.3).

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

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

On entry: specifies a tolerance on the optimal IP solution, i.e., an IP solution returned by nag_ip_bb (h02bbc) as optimal may have an objective function value which is as much as **options**.**soln_tol** greater than that associated with the true optimal IP solution. By setting **options**.**soln_tol** to a nonzero value, the size of the BB search tree may be reduced at the expense of obtaining a (possibly) inferior solution (see Section 12.3).

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

Constraint: **options**.**soln_tol** ≥ 0.0.

nodsel – Nag_Node_Selection Default = Nag_MinObj_Search

On entry: specifies how nodes are selected during the BB tree search (see Section 12.2). 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 **options**.**branch_dir** (see below). The possible values of **options**.**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 **options**.**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

On entry: specifies how nag_ip_bb (h02bbc) selects the variable to branch on, when an unbranched node has been chosen according to optional parameter **options**.**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 argument **options**.**int_tol** are ignored. The following values of **options**.**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 $\left\|x_i^* - \left[x_i^*\right]\right\|$ is nearest to 0.5, where $\left[x_i^*\right]$ 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 in optional parameter **options**.**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

On entry: 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 **options.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 **options.nodsel** = Nag_Deep_Search, **options.branch_dir** will influence the path taken by nag_ip_bb (h02bbc) as the tree is descended. Similarly, this argument will affect the initial deep search when **options.nodsel** = Nag_DeepMinObj_Search or Nag_DeepBroad_Search. The following values of **options.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 argument **x** to nag_ip_bb (h02bbc). Let x^0 be the initial solution as supplied by you, 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 (h02bbc) will first branch towards z_i^0 and solve this sub-problem. This value of **options.branch_dir** would be appropriate, in conjunction with a deep search (as defined by **options.nodsel**), if you 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**

On entry: if **options.varsel** = Nag_Use_Priority then for each integer variable x_i, **options.priority**$[i-1]$ must contain the priority the variable should be given when nag_ip_bb (h02bbc) selects a variable to branch on (x_i is an integer variable if **intvar**$[i-1]$ = Nag_TRUE, for $i = 1, 2, \ldots, n$). For example, if x_k and x_l are integer variables and **options.priority**$[l-1] >$ **options.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 **options.priority**$[k-1] =$ **options.priority**$[l-1]$).

With some problems of type MILP, setting **options.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**$[i-1]$ = Nag_FALSE), **options.priority**$[i-1]$ is not referenced. If optional parameter **options.varsel** $\neq$ Nag_Use_Priority then **options.priority** is not referenced.

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

On entry: 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 **options.feas_tol**.

Constraint: **options.feas_tol** > 0.0.

inf_bound – double Default = 10^{20}

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

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

rank_tol – double Default = 100ϵ

This argument is not used for problems of type MILP.

On entry: **options.rank_tol** enables you to control the condition number of the triangular matrix factor R which arises in solving a QP subproblem (see Section 12 in 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 $\left|R_{i+1,i+1}\right| \leq$ **options.rank_tol** $\times |\rho_{i+1}|$.

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

hrows – Integer Default $= 0$ or **n**

On entry: specifies n_H, the number of rows of the quadratic term H of the QP objective function. For the default MILP problem type, **options.hrows** is not used and its value is set to zero. For MIQP problem types, the default value of **options.hrows** is **n**, the number of variables. However, a value of **options.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, **options.hrows** may be used to define the dimension of a leading block of nonzeros 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**

On entry: places a limit on the storage allocated for the triangular factor R of the reduced Hessian H_r of QP sub-problems (see Section 12 in nag_opt_qp (e04nfc) for details). Ideally, **options.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 **options.hrows** is specified then the default value of **options.max_df** is set to the value in **options.hrows**.

Constraint: $1 \leq$ **options.max_df** $\leq$ **n**.

crnames – char ** Default $=$ **NULL**

On entry: if **options.crnames** is not **NULL** then it must point to an array of $\mathbf{n} + \mathbf{m}$ character strings, with maximum string length 8, containing the names of the variables and constraints of the problem. Thus, **options.crnames**$[j-1]$ contains the name of the jth variable, $j = 1, 2, \ldots,$ **n**, and **options.crnames**$[\mathbf{n} + i - 1]$ contains the names of the ith constraint, $i = 1, 2, \ldots,$ **m**. If supplied, the names are used in the solution output (see Section 5.1 and Section 11.3).

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 (h02bbc). In this case, nag_ip_mps_read (h02buc) may optionally be used to allocate memory to **options.crnames** and to read the variable and constraint names defined in the MPSX file into **options.crnames**. In this case, the memory freeing function nag_ip_free (h02xzc) should be used to free the memory pointed to by **options.crnames** on return from nag_ip_bb (h02bbc). You should **not** use the standard C function `free()` for this purpose.

lower – double Default $= \mathbf{n} + \mathbf{m}$

On entry: $\mathbf{n} + \mathbf{m}$ values of memory will be automatically allocated by nag_ip_bb (h02bbc) and this is the recommended method of use of **options.lower**. However you may supply memory from the calling program.

On exit: the lower bounds imposed at the point returned in **x**. If no IP solution was found **options.lower** contains the same bounds as supplied 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 $= \mathbf{n} + \mathbf{m}$

On entry: $\mathbf{n} + \mathbf{m}$ values of memory will be automatically allocated by nag_ip_bb (h02bbc) and this is the recommended method of use of **options.upper**. However you may supply memory from the calling program.

On exit: the upper bounds imposed at the point returned in **x**. If no IP solution was found **options.upper** contains the same bounds as supplied 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 $= \mathbf{n} + \mathbf{m}$

On entry: $\mathbf{n} + \mathbf{m}$ values of memory will be automatically allocated by nag_ip_bb (h02bbc) and this is the recommended method of use of **options**.**state**. However you may supply memory from the calling program.

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

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

lambda – double Default $= \mathbf{n} + \mathbf{m}$

On entry: $\mathbf{n} + \mathbf{m}$ values of memory will be automatically allocated by nag_ip_bb (h02bbc) and this is the recommended method of use of **options**.**lambda**. However you may supply memory from the calling program.

On exit: the values of the Lagrange multipliers for each constraint with respect to the current working set 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 **options**.**state**$[j] = 0$, **options**.**lambda**$[j]$ is zero. If x is optimal, **options**.**lambda**$[j]$ should be non-negative if **options**.**state**$[j] = 1$, non-positive if **options**.**state**$[j] = 2$ and zero if **options**.**state**$[j] = 4$.

11.3 Description of Printed Output

The level of printed output can be controlled with the structure members **options**.**list** and **options**.**print_level** (see Section 11.2).

If **options**.**list** = Nag_TRUE then the argument values to nag_ip_bb (h02bbc) are listed, whereas the printout of results is governed by the value of **options**.**print_level**. The default of **options**.**print_level** = Nag_Soln_Iter provides intermediate and final results.

If **options**.**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 **options**.**print_level** = Nag_Soln_Root or Nag_Soln_Root_Iter, the root node solution is output before the BB search is commenced. If **options**.**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 (h02bbc), and the **options**.**crnames** member is assigned to an array of variable and constraint names (see Section 11.2 for details), the name supplied in **options**.**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 $l_j \leq -$**options**.**inf_bound**, where **options**.**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 $u_j \geq$ **options**.**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, **options**.**crnames**$[j-1]$ replaced by **options**.**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 (h02bbc), and the **options**.**crnames** member is assigned to an array of variable and constraint names (see Section 11.2 for details), the name supplied in **options**.**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; you can print the final solution when nag_ip_bb (h02bbc) returns to the calling program.

11.3.1 Output of results via a user-defined printing function

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

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

This section may be skipped if you wish to use the default printing facilities.

When a user-defined function is assigned to **options**.**print_fun** this will be called in preference to the internal print function of nag_ip_bb (h02bbc). Calls to the user-defined function are again controlled by means of the **options**.**print_level** member. Information is provided through **st** and **comm**, the two structure arguments to **options**.**print_fun**.

If **comm**→**node_prt** = Nag_TRUE then the results from the most recently solved node are provided through **st**. Note that **options**.**print_fun** will be called with **comm**→**node_prt** = Nag_TRUE only if **options**.**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 **st**→**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** = Nag_TRUE then the solution of the root node is provided through **st**. Note that **options**.**print_fun** will be called with **comm**→**rootnode_sol_prt** = Nag_TRUE only if **options**.**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 **st→endstate** and their correspondence, if any, to the exit value of **fail** from nag_ip_bb (h02bbc) are:

Value of **st→endstate**	Value of **fail**
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 $\mathbf{x}[j-1]$ of the solution x, for $j = 1, 2, \ldots, \mathbf{st}\rightarrow\mathbf{n}$.

ax – double

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

state – Integer

Contains the status of the **st→n** variables and **st→m** general linear constraints. See Section 11.2 for a description of the possible status values.

lambda – double

Contains the $\mathbf{st}\rightarrow\mathbf{n} + \mathbf{st}\rightarrow\mathbf{m}$ values of the Lagrange multipliers.

bl – double

Contains the $\mathbf{st}\rightarrow\mathbf{n} + \mathbf{st}\rightarrow\mathbf{m}$ lower bounds on the variables.

bu – double

Contains the $\mathbf{st}\rightarrow\mathbf{n} + \mathbf{st}\rightarrow\mathbf{m}$ upper bounds on the variables.

If **comm→sol_prt** = Nag_TRUE then the final IP solution is provided through **st**. Note that **options**.**print_fun** will be called with **comm→sol_prt** = Nag_TRUE only if **options**.**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 **st→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 (h02bbc). Possible values of **st→endstate** and their correspondence to the exit value of **fail** are shown below.

Value of **st→endstate**	Value of **fail**
Nag_MIP_Best_ISol or	

Nag_MIP_Stop_First_ISol	NE_NOERROR
Nag_MIP_No_ISol	NW_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	NW_MIP_MAX_ITER_INT_SOL
Nag_MIP_Max_Itn_No_ISol	NW_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	NW_MIP_MAX_NODES_INT_SOL
Nag_MIP_Max_Nodes_No_ISol	NW_MIP_MAX_NODES_NO_INT_SOL
Nag_MIP_Max_Depth_ISol	NW_MIP_MAX_DEPTH_INT_SOL
Nag_MIP_Max_Depth_No_ISol	NW_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 $j = 1, 2, \ldots, \mathbf{st}\rightarrow\mathbf{n}$.

ax – double

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

state – Integer

Contains the status of the $\mathbf{st}\rightarrow\mathbf{n}$ variables and $\mathbf{st}\rightarrow\mathbf{m}$ general linear constraints. See Section 11.2 for a description of the possible status values.

lambda – double

Contains the $\mathbf{st}\rightarrow\mathbf{n} + \mathbf{st}\rightarrow\mathbf{m}$ values of the Lagrange multipliers.

bl – double

Contains the $\mathbf{st}\rightarrow\mathbf{n} + \mathbf{st}\rightarrow\mathbf{m}$ lower bounds on the variables.

bu – double

Contains the $\mathbf{st}\rightarrow\mathbf{n} + \mathbf{st}\rightarrow\mathbf{m}$ upper bounds on the variables.

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

rootnode_sol_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with the solution of the root node.

node_prt – Nag_Boolean

Will be Nag_TRUE when the print function is called with the result of the most recently solved node.

sol_prt – Nag_Boolean

Will be Nag_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 either before entry to nag_ip_bb (h02bbc) or during a call to **qphess** or **options**.**print_fun**. The type Pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.

12 Further Description

This section provides further information about the BB algorithm used by nag_ip_bb (h02bbc).

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

12.1 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 $x_k \leq \left[x_k^*\right]$ for one node and $x_k \geq \left[x_k^*\right] + 1$ for the other, where $\left[x_k^*\right]$ 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 12.2.

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 (h02bbc) the optional parameter **options**.**first_soln** may be set to Nag_TRUE to request termination at the first integer solution (the default value is Nag_FALSE; see Section 11.2).

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.

12.2 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 (h02bbc) these choices are controlled by the optional parameters **options**.**nodsel**, which controls node selection, and **options**.**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 11.2.

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 12.1). 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.

12.3 Further Reducing the Size of the BB Search Tree

In addition to considering variations in the node and variable selection strategies, you may also consider setting some other arguments to help to reduce the number of nodes searched. Recall from Section 12.1) 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 you know 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 (h02bbc) as the optional parameter **options**.**int_obj_bound**. This will be used by nag_ip_bb (h02bbc) 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 argument which you might consider setting to reduce the size of the tree is **options**.**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 (h02bbc) sets the bound to $f(x^+) -$ **options**.**soln_tol**. This means that integer solutions with objective values within **options**.**soln_tol** of any integer solution already found, cannot themselves be found. The idea here is to allow you to avoid further search for solutions which are not substantially better (as measured by **options**.**soln_tol**) than the best solution found so far. Of course, a sensible choice for the value of **options**.**soln_tol** relies on your knowledge of the problem and requirements on the solution.

Further details of the optional parameters **options**.**int_obj_bound** and **options**.**soln_tol** are given in Section 11.2.

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.