NAG Library Function Document

nag_glopt_bnd_mcs_solve (e05jbc)

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

1 Purpose

nag_glopt_bnd_mcs_solve (e05jbc) is designed to find the global minimum or maximum of an arbitrary function, subject to simple bound-constraints using a multi-level coordinate search method. Derivatives are not required, but convergence is only guaranteed if the objective function is continuous in a neighbourhood of a global optimum. It is not intended for large problems.

The initialization function nag_glopt_bnd_mcs_init (e05jac) **must** have been called before calling nag_glopt_bnd_mcs_solve (e05jbc).

2 Specification

```
#include <nag.h>
#include <nage05.h>
void nag_glopt_bnd_mcs_solve (Integer n,
     void (*objfun)(Integer n, const double x[], double *f, Integer nstate,
          Nag_Comm *comm, Integer *inform),
     Nag_BoundType bound, Nag_MCSInitMethod initmethod, double bl[],
     double bu[], Integer sdlist, double list[], Integer numpts[],
     Integer initpt[],
     void (*monit)(Integer n, Integer ncall, const double xbest[],
          const Integer icount[], Integer ninit, const double list[],
          const Integer numpts[], const Integer initpt[], Integer nbaskt,
          const double xbaskt[], const double boxl[], const double boxu[],
          Integer nstate, Nag_Comm *comm, Integer *inform),
     double x[], double *obj, Nag_E05State *state, Nag_Comm *comm,
     NagError *fail)
```

nag_glopt_bnd_mcs_init (e05jac) **must** be called before calling nag_glopt_bnd_mcs_solve (e05jbc), or any of the option-setting or option-getting functions:

nag_glopt_bnd_mcs_optset_file (e05jcc),

nag_glopt_bnd_mcs_optset_string (e05jdc),

nag_glopt_bnd_mcs_optset_char (e05jec),

nag_glopt_bnd_mcs_optset_int (e05jfc),

nag_glopt_bnd_mcs_optset_real (e05jgc),

nag_glopt_bnd_mcs_option_check (e05jhc),

nag_glopt_bnd_mcs_optget_int (e05jkc) or

nag_glopt_bnd_mcs_optget_real (e05jlc).

You **must not** alter the number of non-fixed variables in your problem or the contents of **state** between calls of the functions:

nag_glopt_bnd_mcs_init (e05jac),

nag_glopt_bnd_mcs_solve (e05jbc),

nag_glopt_bnd_mcs_optset_file (e05jcc),

nag_glopt_bnd_mcs_optset_string (e05jdc),

nag_glopt_bnd_mcs_optset_char (e05jec),

nag_glopt_bnd_mcs_optset_int (e05jfc),

nag_glopt_bnd_mcs_optset_real (e05jgc),

nag_glopt_bnd_mcs_option_check (e05jhc),

nag_glopt_bnd_mcs_optget_int (e05jkc) or

nag_glopt_bnd_mcs_optget_real (e05jlc).

3 Description

nag_glopt_bnd_mcs_solve (e05jbc) is designed to solve modestly sized global optimization problems having simple bound-constraints only; it finds the global optimum of a nonlinear function subject to a set of bound constraints on the variables. Without loss of generality, the problem is assumed to be stated in the following form:

$$\underset{\mathbf{x}\in R^n}{\text{minimize}} F(\mathbf{x}) \quad \text{subject to} \quad \boldsymbol{\ell} \leq \mathbf{x} \leq \mathbf{u} \quad \text{and} \quad \boldsymbol{\ell} \leq \mathbf{u},$$

where $F(\mathbf{x})$ (the *objective function*) is a nonlinear scalar function (assumed to be continuous in a neighbourhood of a global minimum), and the bound vectors are elements of $\bar{R}^n$, where $\bar{R}$ denotes the extended reals $R \cup \{-\infty, \infty\}$. Relational operators between vectors are interpreted elementwise.

The optional argument **Maximize** should be set if you wish to solve maximization, rather than minimization, problems.

If certain bounds are not present, the associated elements of $\boldsymbol{\ell}$ or $\mathbf{u}$ can be set to special values that will be treated as $-\infty$ or $+\infty$. See the description of the optional argument **Infinite Bound Size**. Phrases in this document containing terms like 'unbounded values' should be understood to be taken relative to this optional argument.

Fixing variables (that is, setting $l_i = u_i$ for some i) is allowed in nag_glopt_bnd_mcs_solve (e05jbc).

A typical excerpt from a function calling nag_glopt_bnd_mcs_solve (e05jbc) is:

```
nag_glopt_bnd_mcs_init(n_r, &state, ...);
nag_glopt_bnd_mcs_optset_string(optstr, &state, ...);
nag_glopt_bnd_mcs_solve(n, objfun, ...);
```

where nag_glopt_bnd_mcs_optset_string (e05jdc) sets the optional argument and value specified in **optstr**.

The initialization function nag_glopt_bnd_mcs_init (e05jac) does not need to be called before each invocation of nag_glopt_bnd_mcs_solve (e05jbc). You should be aware that a call to the initialization function will reset each optional argument to its default value, and, if you are using repeatable randomized initialization lists (see the description of the argument **initmethod**), the random state stored in **state** will be destroyed.

You must supply a function that evaluates $F(\mathbf{x})$; derivatives are not required.

The method used by nag_glopt_bnd_mcs_solve (e05jbc) is based on MCS, the Multi-level Coordinate Search method described in Huyer and Neumaier (1999), and the algorithm it uses is described in detail in Section 11.

4 References

Huyer W and Neumaier A (1999) Global optimization by multi-level coordinate search *Journal of Global Optimization* **14** 331–355

5 Arguments

Note: for convenience the subarray notation $a(i:j,k:l)$, as described in Section 3.2.1.4 in the Essential Introduction, is used. Using this notation, the term 'column index' refers to the index j in **LIST**(i,j), say (see **list** for the definition of **LIST**).

1: **n** – Integer *Input*

On entry: n, the number of variables.

Constraint: **n** > 0.

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

objfun must evaluate the objective function $F(\mathbf{x})$ for a specified n-vector **x**.

The specification of **objfun** is:

```
void objfun (Integer n, const double x[], double *f, Integer nstate,
     Nag_Comm *comm, Integer *inform)
```

1: **n** – Integer *Input*

On entry: n, the number of variables.

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

On entry: **x**, the vector at which the objective function is to be evaluated.

3: **f** – double * *Output*

On exit: must be set to the value of the objective function at **x**, unless you have specified termination of the current problem using **inform**.

4: **nstate** – Integer *Input*

On entry: if **nstate** $= 1$ then nag_glopt_bnd_mcs_solve (e05jbc) is calling **objfun** for the first time. This argument setting allows you to save computation time if certain data must be read or calculated only once.

5: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_glopt_bnd_mcs_solve (e05jbc) you may allocate memory and initialize these pointers with various quantities for use by **objfun** when called from nag_glopt_bnd_mcs_solve (e05jbc) (see Section 3.2.1.1 in the Essential Introduction).

6: **inform** – Integer * *Output*

On exit: must be set to a value describing the action to be taken by the solver on return from **objfun**. Specifically, if the value is negative the solution of the current problem will terminate immediately; otherwise, computations will continue.

3: **bound** – Nag_BoundType *Input*

On entry: indicates whether the facility for dealing with bounds of special forms is to be used. **bound** must be set to one of the following values.

bound = Nag_Bounds
: You will supply $\boldsymbol{\ell}$ and $\mathbf{u}$ individually.

bound = Nag_NoBounds
: There are no bounds on $\mathbf{x}$.

bound = Nag_BoundsZero
: There are semi-infinite bounds $0 \leq \mathbf{x}$.

bound = Nag_BoundsEqual
: There are constant bounds $\boldsymbol{\ell} = \ell_1$ and $\mathbf{u} = u_1$.

Note that it only makes sense to fix any components of $\mathbf{x}$ when **bound** = Nag_Bounds.

Constraint: **bound** = Nag_Bounds, Nag_NoBounds, Nag_BoundsZero or Nag_BoundsEqual.

4: **initmethod** – Nag_MCSInitMethod *Input*

On entry: selects which initialization method to use.

initmethod = Nag_SimpleBdry
: Simple initialization (boundary and midpoint), with
numpts$[i-1] = 3$, **initpt**$[i-1] = 2$ and
$\mathbf{LIST}(i,j) = (\mathbf{bl}[i-1], (\mathbf{bl}[i-1] + \mathbf{bu}[i-1])/2, \mathbf{bu}[i-1])$,
for $i = 1, 2, \ldots, \mathbf{n}$ and $j = 1, 2, 3$.

initmethod = Nag_SimpleOffBdry
: Simple initialization (off-boundary and midpoint), with
numpts$[i-1] = 3$, **initpt**$[i-1] = 2$ and
$\mathbf{LIST}(i,j) =$
$((5\mathbf{bl}[i-1] + \mathbf{bu}[i-1])/6, (\mathbf{bl}[i-1] + \mathbf{bu}[i-1])/2, (\mathbf{bl}[i-1] + 5\mathbf{bu}[i-1])/6)$,
for $i = 1, 2, \ldots, \mathbf{n}$ and $j = 1, 2, 3$.

initmethod = Nag_Linesearch
: Initialization using linesearches.

initmethod = Nag_UserSet
: You are providing your own initialization list.

initmethod = Nag_Random
: Generate a random initialization list.

See **list** for the definition of **LIST**.

For more information on methods **initmethod** = Nag_Linesearch, Nag_UserSet or Nag_Random see Section 11.1.

If ‘infinite’ values (as determined by the value of the optional argument **Infinite Bound Size**) are detected by nag_glopt_bnd_mcs_solve (e05jbc) when you are using a simple initialization method (**initmethod** = Nag_SimpleBdry or Nag_SimpleOffBdry), a safeguarded initialization procedure will be attempted, to avoid overflow.

Suggested value: **initmethod** = Nag_SimpleBdry

Constraint: **initmethod** = Nag_SimpleBdry, Nag_SimpleOffBdry, Nag_Linesearch, Nag_UserSet or Nag_Random.

5: **bl**[**n**] – double *Input/Output*
6: **bu**[**n**] – double *Input/Output*

On entry: **bl** is $\boldsymbol{\ell}$, the array of lower bounds. **bu** is $\mathbf{u}$, the array of upper bounds.

If **bound** $=$ Nag_Bounds, you must set **bl**$[i-1]$ to ℓ_i and **bu**$[i-1]$ to u_i, for $i = 1, 2, \ldots,$ **n**. If a particular x_i is to be unbounded below, the corresponding **bl**$[i-1]$ should be set to $-infbnd$, where $infbnd$ is the value of the optional argument **Infinite Bound Size**. Similarly, if a particular x_i is to be unbounded above, the corresponding **bu**$[i-1]$ should be set to $infbnd$.

If **bound** $=$ Nag_NoBounds or Nag_BoundsZero, arrays **bl** and **bu** need not be set on input.

If **bound** $=$ Nag_BoundsEqual, you must set **bl**$[0]$ to ℓ_1 and **bu**$[0]$ to u_1. The remaining elements of **bl** and **bu** will then be populated by these initial values.

On exit: unless **fail.code** $=$ NE_INT, NE_INT_2, NE_NOT_INIT, NE_REAL or NE_REAL_2 on exit, **bl** and **bu** are the actual arrays of bounds used by nag_glopt_bnd_mcs_solve (e05jbc).

Constraints:

if **bound** $=$ Nag_Bounds, **bl**$[i-1] \leq$ **bu**$[i-1]$, for $i = 1, 2, \ldots,$ **n**;
if **bound** $=$ Nag_BoundsEqual, **bl**$[0] <$ **bu**$[0]$.

7: **sdlist** – Integer *Input*

On entry: must be set to, at least, the maximum over i of the number of points in coordinate i at which to split according to the initialization list **list**; that is, **sdlist** $\geq \max_i$ **numpts**$[i-1]$.

Internally, nag_glopt_bnd_mcs_solve (e05jbc) uses **list** to determine sets of points along each coordinate direction to which it fits quadratic interpolants. Since fitting a quadratic requires at least three distinct points, this puts a lower bound on **sdlist**. Furthermore, in the case of initialization by linesearches (**initmethod** $=$ Nag_Linesearch) internal storage considerations require that **sdlist** be at least 192.

Constraints:

if **initmethod** $\neq$ Nag_Linesearch, **sdlist** ≥ 3;
if **initmethod** $=$ Nag_Linesearch, **sdlist** ≥ 192;
if **initmethod** $=$ Nag_UserSet, **sdlist** $\geq \max_i\{$**numpts**$[i-1]\}$.

8: **list**[**n** $\times$ **sdlist**] – double *Input/Output*

Note: where **LIST**(i,j) appears in this document, it refers to the array element **list**$[(i-1) \times$ **sdlist** $+ j - 1]$.

Note: for convenience the subarray notation **LIST**$(i:j, k:l)$, as described in Section 3.2.1.4 in the Essential Introduction, is used. Using this notation, the term 'column index' refers to the index j in **LIST**(i,j), say.

On entry: this argument need not be set on entry if you wish to use one of the preset initialization methods (**initmethod** $\neq$ Nag_UserSet).

list is the 'initialization list': whenever a sub-box in the algorithm is split for the first time (either during the *initialization procedure* or later), for each non-fixed coordinate i the split is done at the values **LIST**$(i, 1:$ **numpts**$[i-1])$, as well as at some adaptively chosen intermediate points. The array sections **LIST**$(i, 1:$ **numpts**$[i-1])$, for $i = 1, 2, \ldots,$ **n**, must be in ascending order with each entry being distinct. In this context, 'distinct' should be taken to mean relative to the safe-range argument (see nag_real_safe_small_number (X02AMC)).

On exit: unless **fail.code** $=$ NE_ALLOC_FAIL, NE_INT, NE_INT_2, NE_NOT_INIT, NE_REAL or NE_REAL_2 on exit, the actual initialization data used by nag_glopt_bnd_mcs_solve (e05jbc). If you wish to monitor the contents of **list** you are advised to do so solely through **monit**, not through the output value here.

Constraint: if **x**$[i-1]$ is not fixed, **LIST**$(i, 1:$ **numpts**$[i-1])$ is in ascending order with each entry being distinct, for $i = 1, 2, \ldots,$ **nbl**$[i-1] \leq$ **LIST**$(i,j) \leq$ **bu**$[i-1]$, for $i = 1, 2, \ldots,$ **n** and $j = 1, 2, \ldots,$ **numpts**$[i-1]$.

9: **numpts**[**n**] – Integer *Input/Output*

On entry: this argument need not be set on entry if you wish to use one of the preset initialization methods (**initmethod** $\neq$ Nag_UserSet).

numpts encodes the number of splitting points in each non-fixed dimension.

On exit: unless **fail**.**code** = NE_ALLOC_FAIL, NE_INT, NE_INT_2, NE_NOT_INIT, NE_REAL or NE_REAL_2 on exit, the actual initialization data used by nag_glopt_bnd_mcs_solve (e05jbc).

Constraints:

if **x**[$i-1$] is not fixed, **numpts**[$i-1$] $\leq$ **sdlist**;
numpts[$i-1$] ≥ 3, for $i = 1, 2, \ldots,$ **n**.

10: **initpt**[**n**] – Integer *Input/Output*

On entry: this argument need not be set on entry if you wish to use one of the preset initialization methods (**initmethod** $\neq$ Nag_UserSet).

You must designate a point stored in **list** that you wish nag_glopt_bnd_mcs_solve (e05jbc) to consider as an 'initial point' for the purposes of the splitting procedure. Call this initial point $\mathbf{x}^*$. The coordinates of $\mathbf{x}^*$ correspond to a set of indices J_i, for $i = 1, 2, \ldots, n$, such that $\mathbf{x}_i^*$ is stored in **LIST**(i, J_i), for $i = 1, 2, \ldots, n$. You must set **initpt**[$i-1$] $= J_i$, for $i = 1, 2, \ldots, n$.

On exit: unless **fail**.**code** = NE_ALLOC_FAIL, NE_INT, NE_INT_2, NE_NOT_INIT, NE_REAL or NE_REAL_2 on exit, the actual initialization data used by nag_glopt_bnd_mcs_solve (e05jbc).

Constraint: if **x**[$i-1$] is not fixed, $1 \leq$ **initpt**[$i-1$] $\leq$ **sdlist**, for $i = 1, 2, \ldots,$ **n**.

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

monit may be used to monitor the optimization process. It is invoked upon every successful completion of the procedure in which a sub-box is considered for splitting. It will also be called just before nag_glopt_bnd_mcs_solve (e05jbc) exits if that splitting procedure was not successful.

If no monitoring is required, **monit** may be specified as **NULLFN**.

The specification of **monit** is:

```
void monit (Integer n, Integer ncall, const double xbest[],
     const Integer icount[], Integer ninit, const double list[],
     const Integer numpts[], const Integer initpt[], Integer nbaskt,
     const double xbaskt[], const double boxl[], const double boxu[],
     Integer nstate, Nag_Comm *comm, Integer *inform)
```

1: **n** – Integer *Input*

On entry: n, the number of variables.

2: **ncall** – Integer *Input*

On entry: the cumulative number of calls to **objfun**.

3: **xbest**[**n**] – const double *Input*

On entry: the current best point.

4: **icount**[**6**] – const Integer *Input*

On entry: an array of counters.

icount[0]
$nboxes$, the current number of sub-boxes.

icount[1]
$ncloc$, the cumulative number of calls to **objfun** made in local searches.

icount[2]
$nloc$, the cumulative number of points used as start points for local searches.

icount[3]
$nsweep$, the cumulative number of sweeps through levels.

icount[4]
m, the cumulative number of splits by initialization list.

icount[5]
s, the current lowest level containing non-split boxes.

5: **ninit** – Integer *Input*

On entry: the maximum over i of the number of points in coordinate i at which to split according to the initialization list **list**. See also the description of the argument **numpts**.

6: **list**[**n** × **ninit**] – const double *Input*

On entry: the initialization list.

7: **numpts**[**n**] – const Integer *Input*

On entry: the number of points in each coordinate at which to split according to the initialization list **list**.

8: **initpt**[**n**] – const Integer *Input*

On entry: a pointer to the 'initial point' in **list**. Element **initpt**[$i - 1$] is the column index in **LIST** of the ith coordinate of the initial point.

9: **nbaskt** – Integer *Input*

On entry: the number of points in the 'shopping basket' **xbaskt**.

10: **xbaskt**[**n** × **nbaskt**] – const double *Input*

Note: The jth candidate minimum has its ith coordinate stored in **xbaskt**[$(j - 1) \times$ **nbaskt** $+ i - 1$], for $i = 1, 2, \ldots,$ **n** and $j = 1, 2, \ldots,$ **nbaskt**.

On entry: the 'shopping basket' of candidate minima.

11: **boxl**[**n**] – const double *Input*

On entry: the array of lower bounds of the current search box.

12: **boxu**[**n**] – const double *Input*

On entry: the array of upper bounds of the current search box.

13: **nstate** – Integer *Input*

On entry: is set by nag_glopt_bnd_mcs_solve (e05jbc) to indicate at what stage of the minimization **monit** was called.

nstate $= 1$
This is the first time that **monit** has been called.

nstate $= -1$
This is the last time **monit** will be called.

nstate $= 0$
This is the first *and* last time **monit** will be called.

14: **comm** – Nag_Comm *

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

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_glopt_bnd_mcs_solve (e05jbc) you may allocate memory and initialize these pointers with various quantities for use by **monit** when called from nag_glopt_bnd_mcs_solve (e05jbc) (see Section 3.2.1.1 in the Essential Introduction).

15: **inform** – Integer * *Output*

On exit: must be set to a value describing the action to be taken by the solver on return from **monit**. Specifically, if the value is negative the solution of the current problem will terminate immediately; otherwise, computations will continue.

12: **x**[**n**] – double *Output*

On exit: if **fail.code** = NE_NOERROR, contains an estimate of the global optimum (see also Section 7).

13: **obj** – double * *Output*

On exit: if **fail.code** = NE_NOERROR, contains the function value at **x**.

If you request early termination of nag_glopt_bnd_mcs_solve (e05jbc) using **inform** in **objfun** or the analogous **inform** in **monit**, there is no guarantee that the function value at **x** equals **obj**.

14: **state** – Nag_E05State * *Communication Structure*

state contains information required by other functions in this suite. You must not modify it directly in any way.

15: **comm** – Nag_Comm *

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

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

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

nag_glopt_bnd_mcs_solve (e05jbc) returns with **fail.code** = NE_NOERROR if your termination criterion has been met: either a target value has been found to the required relative error (as determined by the values of the optional arguments **Target Objective Value**, **Target Objective Error** and **Target Objective Safeguard**), or the best function value was static for the number of sweeps through levels given by the optional argument **Static Limit**. The latter criterion is the default.

6 Error Indicators and Warnings

NE_ALLOC_FAIL

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

NE_BAD_PARAM

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

NE_DIV_COMPLETE

The division procedure completed but your target value could not be reached.
Despite every sub-box being processed **Splits Limit** times, the target value you provided in **Target Objective Value** could not be found to the tolerances given in **Target Objective Error** and **Target Objective Safeguard**. You could try reducing **Splits Limit** or the objective tolerances.

NE_INF_INIT_LIST

A finite initialization list could not be computed internally. Consider reformulating the bounds on the problem, try providing your own initialization list, use the randomization option (**initmethod** $=$ Nag_Random) or vary the value of **Infinite Bound Size**.

The user-supplied initialization list contained infinite values, as determined by the optional argument **Infinite Bound Size**.

NE_INLIST_CLOSE

An error occurred during initialization. It is likely that points from the initialization list are very close together. Try relaxing the bounds on the variables or use a different initialization method.

NE_INT

On entry, **initmethod** $=$ Nag_Linesearch and **sdlist** $= \langle$*value*$\rangle$.
Constraint: if **initmethod** $=$ Nag_Linesearch then **sdlist** ≥ 192.

On entry, **initmethod** $= \langle$*value*$\rangle$ and **sdlist** $= \langle$*value*$\rangle$.
Constraint: if **initmethod** $\neq$ Nag_Linesearch then **sdlist** ≥ 3.

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

On entry, user-supplied section **LIST**$(i, 1:$ **numpts**$[i-1])$ contained $ndist$ distinct elements, and $ndist <$ **numpts**$[i-1]$: $ndist = \langle$*value*$\rangle$, **numpts**$[i-1] = \langle$*value*$\rangle$, $i = \langle$*value*$\rangle$.

The number of non-fixed variables $n_r = 0$.
Constraint: $n_r > 0$.

NE_INT_2

A value of **Splits Limit** ($smax$) smaller than $n_r + 3$ was set: $smax = \langle$*value*$\rangle$, $n_r = \langle$*value*$\rangle$.

On entry, user-supplied **initpt**$[i-1] = \langle$*value*$\rangle$, $i = \langle$*value*$\rangle$.
Constraint: if **x**$[i-1]$ is not fixed then **initpt**$[i-1] \geq 1$, for $i = 1, 2, \ldots,$ **n**.

On entry, user-supplied **initpt**$[i-1] = \langle$*value*$\rangle$, $i = \langle$*value*$\rangle$ and **sdlist** $= \langle$*value*$\rangle$.
Constraint: if **x**$[i-1]$ is not fixed then **initpt**$[i-1] \leq$ **sdlist**, for $i = 1, 2, \ldots,$ **n**.

On entry, user-supplied **numpts**$[i-1] = \langle$*value*$\rangle$, $i = \langle$*value*$\rangle$.
Constraint: if **x**$[i-1]$ is not fixed then **numpts**$[i-1] \geq 3$, for $i = 1, 2, \ldots,$ **n**.

On entry, user-supplied **numpts**$[i-1] = \langle$*value*$\rangle$, $i = \langle$*value*$\rangle$ and **sdlist** $= \langle$*value*$\rangle$.
Constraint: if **x**$[i-1]$ is not fixed then **numpts**$[i-1] \leq$ **sdlist**, for $i = 1, 2, \ldots,$ **n**.

On entry, user-supplied section **LIST**$(i, 1:$ **numpts**$[i-1])$ was not in ascending order: **numpts**$[i-1] = \langle$*value*$\rangle$, $i = \langle$*value*$\rangle$.

NE_INTERNAL_ERROR

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

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

NE_LINESEARCH_ERROR

An error occurred during linesearching. It is likely that your objective function is badly scaled: try rescaling it. Also, try relaxing the bounds or use a different initialization method. If the problem persists, please contact NAG quoting error code $\langle value \rangle$.

NE_MONIT_TERMIN

User-supplied monitoring function requested termination.

NE_NO_LICENCE

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

NE_NOT_INIT

Initialization function nag_glopt_bnd_mcs_init (e05jac) has not been called.

NE_OBJFUN_TERMIN

User-supplied objective function requested termination.

NE_REAL

On entry, **bound** $=$ Nag_BoundsEqual and **bl**$[0] =$ **bu**$[0] = \langle value \rangle$.
Constraint: if **bound** $=$ Nag_BoundsEqual then **bl**$[0] <$ **bu**$[0]$.

NE_REAL_2

On entry, **bound** $=$ Nag_Bounds or Nag_BoundsEqual and **bl**$[i-1] = \langle value \rangle$, **bu**$[i-1] = \langle value \rangle$ and $i = \langle value \rangle$.
Constraint: if **bound** $=$ Nag_Bounds then **bl**$[i-1] \leq$ **bu**$[i-1]$, for $i = 1, 2, \ldots,$ **n**; if **bound** $=$ Nag_BoundsEqual then **bl**$[0] <$ **bu**$[0]$.

On entry, user-supplied **LIST**$(i,j) = \langle value \rangle$, $i = \langle value \rangle$, $j = \langle value \rangle$, and **bl**$[i-1] = \langle value \rangle$.
Constraint: if **x**$[i-1]$ is not fixed then **LIST**$(i,j) \geq$ **bl**$[i-1]$, for $i = 1, 2, \ldots,$ **n** and $j = 1, 2, \ldots,$ **numpts**$[i-1]$.

On entry, user-supplied **LIST**$(i,j) = \langle value \rangle$, $i = \langle value \rangle$, $j = \langle value \rangle$, and **bu**$[i-1] = \langle value \rangle$.
Constraint: if **x**$[i-1]$ is not fixed then **LIST**$(i,j) \leq$ **bu**$[i-1]$, for $i = 1, 2, \ldots,$ **n** and $j = 1, 2, \ldots,$ **numpts**$[i-1]$.

NE_TOO_MANY_FEVALS

The function evaluations limit was exceeded.
Approximately **Function Evaluations Limit** function calls have been made without your chosen termination criterion being satisfied.

7 Accuracy

If **fail.code** $=$ NE_NOERROR on exit, then the vector returned in the array **x** is an estimate of the solution **x** whose function value satisfies your termination criterion: the function value was static for **Static Limit** sweeps through levels, or

$$F(\mathbf{x}) - objval \leq \max(objerr \times |objval|, objsfg),$$

where *objval* is the value of the optional argument **Target Objective Value**, *objerr* is the value of the optional argument **Target Objective Error**, and *objsfg* is the value of the optional argument **Target Objective Safeguard**.

8 Parallelism and Performance

nag_glopt_bnd_mcs_solve (e05jbc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_glopt_bnd_mcs_solve (e05jbc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

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

9 Further Comments

For each invocation of nag_glopt_bnd_mcs_solve (e05jbc), local workspace arrays of fixed length are allocated internally. The total size of these arrays amounts to $13n_r + smax - 1$ Integer elements, where $smax$ is the value of the optional argument **Splits Limit** and n_r is the number of non-fixed variables, and $(2 + n_r)\mathbf{sdlist} + 2\mathbf{n} + 22n_r + 3n_r^2 + 1$ double elements. In addition, if you are using randomized initialization lists (see the description of the argument **initmethod**), a further 21 Integer elements are allocated internally.

In order to keep track of the regions of the search space that have been visited while looking for a global optimum, nag_glopt_bnd_mcs_solve (e05jbc) internally allocates arrays of increasing sizes depending on the difficulty of the problem. Two of the main factors that govern the amount allocated are the number of sub-boxes (call this quantity $nboxes$) and the number of points in the 'shopping basket' (the argument **nbaskt** on entry to **monit**). Safe, pessimistic upper bounds on these two quantities are so large as to be impractical. In fact, the worst-case number of sub-boxes for even the most simple initialization list (when $\mathbf{ninit} = 3$ on entry to **monit**) grows like ${n_r}^{n_r}$. Thus nag_glopt_bnd_mcs_solve (e05jbc) does not attempt to estimate in advance the final values of $nboxes$ or **nbaskt** for a given problem. There are a total of 5 Integer arrays and $4 + n_r + \mathbf{ninit}$ double arrays whose lengths depend on $nboxes$, and there are a total of 2 Integer arrays and $3 + \mathbf{n} + n_r$ double arrays whose lengths depend on **nbaskt**. nag_glopt_bnd_mcs_solve (e05jbc) makes a fixed initial guess that the maximum number of sub-boxes required will be 10000 and that the maximum number of points in the 'shopping basket' will be 1000. If ever a greater amount of sub-boxes or more room in the 'shopping basket' is required, nag_glopt_bnd_mcs_solve (e05jbc) performs reallocation, usually doubling the size of the inadequately-sized arrays. Clearly this process requires periods where the original array and its extension exist in memory simultaneously, so that the data within can be copied, which compounds the complexity of nag_glopt_bnd_mcs_solve (e05jbc)'s memory usage. It is possible (although not likely) that if your problem is particularly difficult to solve, or of a large size (hundreds of variables), you may run out of memory.

One array that could be dynamically resized by nag_glopt_bnd_mcs_solve (e05jbc) is the 'shopping basket' (**xbaskt** on entry to **monit**). If the initial attempt to allocate $1000n_r$ *double*s for this array fails, **monit** will not be called on exit from nag_glopt_bnd_mcs_solve (e05jbc).

nag_glopt_bnd_mcs_solve (e05jbc) performs better if your problem is well-scaled. It is worth trying (by guesswork perhaps) to rescale the problem if necessary, as sensible scaling will reduce the difficulty of the optimization problem, so that nag_glopt_bnd_mcs_solve (e05jbc) will take less computer time.

10 Example

This example finds the global minimum of the 'peaks' function in two dimensions

$$F(x,y) = 3(1-x)^2 \exp\left(-x^2 - (y+1)^2\right) - 10\left(\frac{x}{5} - x^3 - y^5\right) \exp\left(-x^2 - y^2\right) - \frac{1}{3}\exp\left(-(x+1)^2 - y^2\right)$$

on the box $[-3, 3] \times [-3, 3]$.

The function F has several local minima and one global minimum in the given box. The global minimum is approximately located at $(0.23, -1.63)$, where the function value is approximately -6.55.

We use default values for all the optional arguments, and we instruct nag_glopt_bnd_mcs_solve (e05jbc) to use the simple initialization list corresponding to **initmethod** = Nag_SimpleBdry. In particular, this will set for us the initial point $(0, 0)$ (see Section 10.3).

10.1 Program Text

```
/* nag_glopt_bnd_mcs_solve (e05jbc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 25, 2014.
 */
#include <stdio.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nage05.h>

#ifdef __cplusplus
extern "C" {
#endif
static void NAG_CALL objfun(Integer n, const double x[], double *f,
                            Integer nstate, Nag_Comm *comm, Integer *inform);
static void NAG_CALL monit(Integer n, Integer ncall, const double xbest[],
                           const Integer icount[], Integer sdlist,
                           const double list[], const Integer numpts[],
                           const Integer initpt[], Integer nbaskt,
                           const double xbaskt[], const double boxl[],
                           const double boxu[], Integer nstate, Nag_Comm *comm,
                           Integer *inform);
static void NAG_CALL output_current_box(const double boxl[],
                                        const double boxu[]);
#ifdef __cplusplus
}
#endif

int main(void)
{
  /* Scalars */
  double            obj;
  Integer           exit_status=0, i, n=2, plot, sdlist;
  Nag_BoundType     boundenum;
  Nag_MCSInitMethod initmethodenum;
  /* Arrays */
  static double     ruser[2] = {-1.0, -1.0};
  char              bound[16], initmethod[18];
  double            *bl = 0, *bu = 0, *list = 0, *x = 0;
  Integer           *initpt = 0, *numpts = 0;
  Integer           iuser[1];
  /* Nag Types */
  Nag_E05State      state;
  NagError          fail;
  Nag_Comm          comm;

  INIT_FAIL(fail);

  printf("nag_glopt_bnd_mcs_solve (e05jbc) Example Program Results\n");

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

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

```
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%*[^\n] ", &sdlist);
#else
  scanf("%"NAG_IFMT"%*[^\n] ", &sdlist);
#endif

  if (n <= 0 || sdlist <= 0)
    goto END;

  if (!(bl = NAG_ALLOC(n, double)) ||
      !(bu = NAG_ALLOC(n, double)) ||
      !(list = NAG_ALLOC(n*sdlist, double)) ||
      !(x = NAG_ALLOC(n, double)) ||
      !(initpt = NAG_ALLOC(n, Integer)) ||
      !(numpts = NAG_ALLOC(n, Integer)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Read in bound (and bl and bu if necessary) */
#ifdef _WIN32
  scanf_s("%15s%*[^\n] ", bound, _countof(bound));
#else
  scanf("%15s%*[^\n] ", bound);
#endif

  /* nag_enum_name_to_value (x04nac).
   * Converts NAG enum member name to value
   */
  boundenum = (Nag_BoundType) nag_enum_name_to_value(bound);

  if (boundenum == Nag_Bounds)
    /* Read in the whole of each bound */
    {
      for (i = 0; i < n; ++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 < n; ++i)
#ifdef _WIN32
        scanf_s("%lf", &bu[i]);
#else
        scanf("%lf", &bu[i]);
#endif
#ifdef _WIN32
      scanf_s("%*[^\n] ");
#else
      scanf("%*[^\n] ");
#endif
    }
  else if (boundenum == Nag_BoundsEqual)
    /* Bounds are uniform: read in only the first entry of each */
    {
#ifdef _WIN32
      scanf_s("%lf%*[^\n] ", &bl[0]);
#else
      scanf("%lf%*[^\n] ", &bl[0]);
#endif
#ifdef _WIN32
      scanf_s("%lf%*[^\n] ", &bu[0]);
#else
```

```
      scanf("%lf%*[^\n] ", &bu[0]);
#endif
    }

  /* Read in initmethod */
#ifdef _WIN32
  scanf_s("%17s%*[^\n] ", initmethod, _countof(initmethod));
#else
  scanf("%17s%*[^\n] ", initmethod);
#endif

  /* nag_enum_name_to_value (x04nac).
   * Converts NAG enum member name to value
   */
  initmethodenum = (Nag_MCSInitMethod) nag_enum_name_to_value(initmethod);

  /* Read in plot. Its value determines whether monit displays
   * information on the current search box
   */
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%*[^\n] ", &plot);
#else
  scanf("%"NAG_IFMT"%*[^\n] ", &plot);
#endif

  /* Communicate plot through to monit */
  iuser[0] = plot;

  /* Call nag_glopt_bnd_mcs_init (e05jac) to initialize
   * nag_glopt_bnd_mcs_solve (e05jbc). */
  /* Its first argument is a legacy argument and has no significance. */
  nag_glopt_bnd_mcs_init(0, &state, &fail);

  if (fail.code != NE_NOERROR)
    {
      printf("Initialization of nag_glopt_bnd_mcs_solve (e05jbc) failed.\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }

  /* Solve the problem. */
  /* nag_glopt_bnd_mcs_solve (e05jbc).
   * Global optimization by multilevel coordinate search, simple bounds.
   */
  nag_glopt_bnd_mcs_solve(n, objfun, boundenum, initmethodenum, bl, bu,
                          sdlist, list, numpts, initpt, monit, x, &obj,
                          &state, &comm, &fail);

  if (fail.code != NE_NOERROR)
    {
      printf("Error message from nag_glopt_bnd_mcs_solve (e05jbc).\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }

  printf("Final objective value = %11.5f\n", obj);
  printf("Global optimum x = ");
  for (i = 0; i < n; ++i)
    printf("%9.5f", x[i]);
  printf("\n");

 END:
  NAG_FREE(bl);
  NAG_FREE(bu);
  NAG_FREE(list);
  NAG_FREE(x);
  NAG_FREE(initpt);
  NAG_FREE(numpts);
  return exit_status;
}

static void NAG_CALL objfun(Integer n, const double x[], double *f,
                            Integer nstate, Nag_Comm *comm, Integer *inform)
{
  /* Routine to evaluate objective function */

  if (comm->user[0] == -1.0)
    {
      printf("(User-supplied callback objfun, first invocation.)\n");
      comm->user[0] = 0.0;
    }

  /* This is a two-dimensional objective function.
   * As an example of using the inform mechanism,
   * terminate if any other problem size is supplied.
   */
  if (n!=2)
    {
      *inform = -1;
      return;
    }

  *inform = 0;

  if (*inform >= 0)
  /* Here we’re prepared to evaluate objfun at the current x */
    {
      if (nstate == 1)
      /* This is the first call to objfun */
        {
          printf("\n(objfun was just called for the first time)\n");
        }

      *f = (
        3.0*pow((1.0-x[0]), 2)*exp(-pow(x[0], 2)-pow((x[1]+1), 2))
        - (10.0*(x[0]/5.0-pow(x[0], 3)-pow(x[1], 5))*
           exp(-pow(x[0], 2)-pow(x[1], 2)))
        - 1.0/3.0*exp(-pow((x[0]+1.0), 2)-pow(x[1], 2))
        );
    }
}

static void NAG_CALL monit(Integer n, Integer ncall, const double xbest[],
                           const Integer icount[], Integer sdlist,
                           const double list[], const Integer numpts[],
                           const Integer initpt[], Integer nbaskt,
                           const double xbaskt[], const double boxl[],
                           const double boxu[], Integer nstate, Nag_Comm *comm,
                           Integer *inform)
{
  /* Scalars */
  Integer i, j;
  Integer plot;

#define LIST(I, J) list[(I-1)*sdlist + (J-1)]
#define XBASKT(I, J) xbaskt[(I-1)*nbaskt + (J-1)]

  if (comm->user[1] == -1.0)
    {
      printf("(User-supplied callback monit, first invocation.)\n");
      comm->user[1] = 0.0;
    }

  *inform = 0;

  if (*inform >= 0)
  /* We are going to allow the iterations to continue */
    {
      /* Extract plot from the communication structure */
      plot = comm->iuser[0];

      if (nstate == 0 || nstate == 1)
      /* When nstate == 1, monit is called for the first time.
       * When nstate == 0, monit is called for the first AND last time.
       * Display a welcome message */
        {
          printf("\n*** Begin monitoring information ***\n\n");

          printf("Values controlling initial splitting of a box:\n");
          for (i = 1; i <= n; ++i)
            {
              printf("**\n");
              printf("In dimension %5"NAG_IFMT"\n", i);
              printf("Extent of initialization list in this dimension ="
                     "%5"NAG_IFMT"\n", numpts[i - 1]);
              printf("Initialization points in this dimension:\n");
              printf("LIST(i, 1:numpts[i - 1]) =");
              for (j = 1; j <= numpts[i - 1]; ++j)
                printf("%9.5f", LIST(i, j));
              printf("\n");
              printf("Initial point in this dimension: LIST(i,%5"NAG_IFMT")\n",
                     initpt[i - 1]);
            }

          if (plot != 0 && n == 2)
            printf("<Begin displaying search boxes>\n\n");
        }

      if (plot != 0 && n == 2)
        {
          /* Display the coordinates of the edges of the current search box */
          output_current_box(boxl, boxu);
        }

      if (nstate <= 0)
      /* monit is called for the last time */
        {
          if (plot != 0 && n == 2)
            printf("<End displaying search boxes>\n\n");
          printf("Total sub-boxes = %5"NAG_IFMT"\n", icount[0]);
          printf("Total function evaluations = %5"NAG_IFMT"\n", ncall);
          printf("Total function evaluations used in local search = "
                 "%5"NAG_IFMT"\n", icount[1]);
          printf("Total points used in local search = %5"NAG_IFMT"\n",
                  icount[2]);
          printf("Total sweeps through levels = %5"NAG_IFMT"\n", icount[3]);
          printf("Total splits by init. list = %5"NAG_IFMT"\n", icount[4]);
          printf("Lowest level with nonsplit boxes = %5"NAG_IFMT"\n",
                  icount[5]);
          printf("Number of candidate minima in the ’shopping basket’"
                  " = %5"NAG_IFMT"\n", nbaskt);
          printf("Shopping basket:\n");

          for (i = 1; i <= n; ++i)
            {
              printf("xbaskt(%3"NAG_IFMT",:) =", i);
              for (j = 1; j <= nbaskt; ++j)
                printf("%9.5f", XBASKT(i, j));
              printf("\n");
            }

          printf("Best point:\n");
          printf("xbest =");
          for (i = 0; i < n; ++i)
            printf("%9.5f", xbest[i]);
          printf("\n");

          printf("\n*** End monitoring information ***\n\n");
        }
    }
```

```
}

static void NAG_CALL output_current_box(const double boxl[],
                                        const double boxu[])
{
  printf("%20.15f %20.15f\n", boxl[0], boxl[1]);
  printf("%20.15f %20.15f\n\n", boxl[0], boxu[1]);
  printf("%20.15f %20.15f\n", boxl[0], boxl[1]);
  printf("%20.15f %20.15f\n\n", boxu[0], boxl[1]);
  printf("%20.15f %20.15f\n", boxl[0], boxu[1]);
  printf("%20.15f %20.15f\n\n", boxu[0], boxu[1]);
  printf("%20.15f %20.15f\n", boxu[0], boxl[1]);
  printf("%20.15f %20.15f\n\n", boxu[0], boxu[1]);
}
```

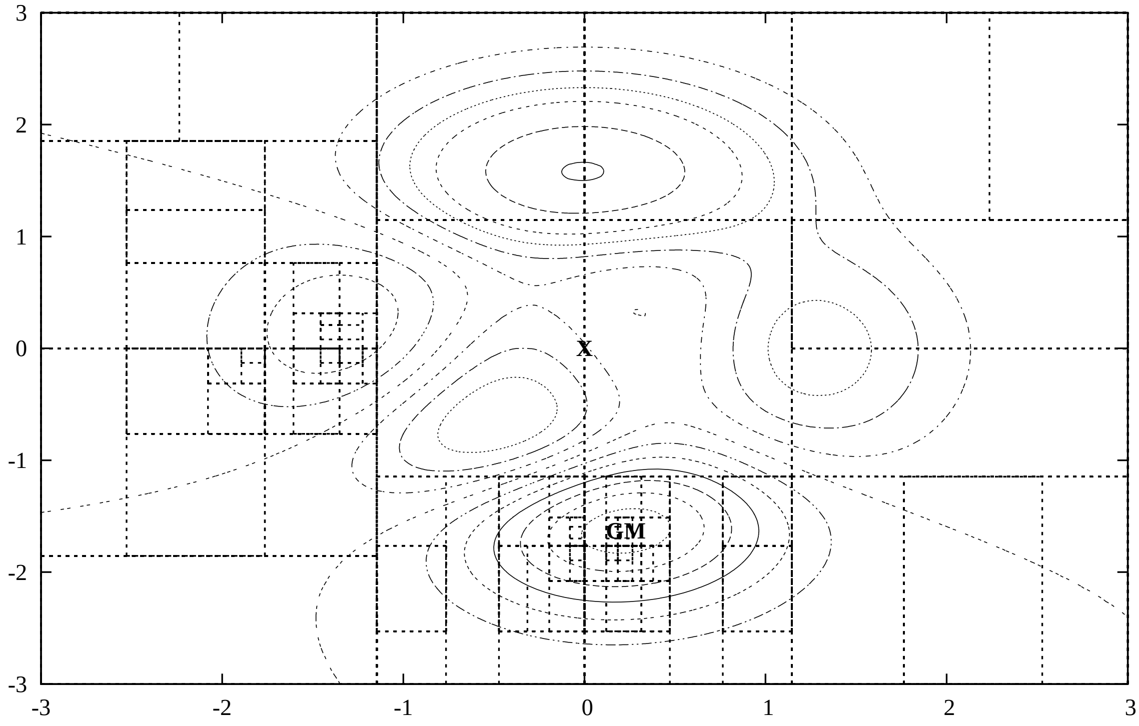
10.2 Program Data

```
nag_glopt_bnd_mcs_solve (e05jbc) Example Program Data
  3                                                          : sdlist
  Nag_Bounds                                                 : bound
  -3.0   -3.0                                                : Lower bounds bl
  3.0   3.0                                                  : Upper bounds bu
  Nag_SimpleBdry                                             : initmethod
  0                                                          : plot
```

10.3 Program Results

```
nag_glopt_bnd_mcs_solve (e05jbc) Example Program Results
(User-supplied callback objfun, first invocation.)

(objfun was just called for the first time)
(User-supplied callback monit, first invocation.)

*** Begin monitoring information ***

Values controlling initial splitting of a box:
**
In dimension     1
Extent of initialization list in this dimension =    3
Initialization points in this dimension:
LIST(i, 1:numpts[i - 1]) = -3.00000  0.00000  3.00000
Initial point in this dimension: LIST(i,    2)
**
In dimension     2
Extent of initialization list in this dimension =    3
Initialization points in this dimension:
LIST(i, 1:numpts[i - 1]) = -3.00000  0.00000  3.00000
Initial point in this dimension: LIST(i,    2)
Total sub-boxes =   228
Total function evaluations =   197
Total function evaluations used in local search =    88
Total points used in local search =    13
Total sweeps through levels =    12
Total splits by init. list =     5
Lowest level with nonsplit boxes =     7
Number of candidate minima in the 'shopping basket' =     2
Shopping basket:
xbaskt(  1,:) = -1.34740  0.22828
xbaskt(  2,:) =  0.20452 -1.62553
Best point:
xbest =  0.22828 -1.62553

*** End monitoring information ***

Final objective value =    -6.55113
Global optimum x =   0.22828 -1.62553
```

Example Program
The Peaks Function *F* and Search Boxes
The global minimum is denoted by **GM**, while our start point is labelled with **X**



Note: *the remainder of this document is intended for more advanced users. Section 11 contains a detailed description of the algorithm. This information may be needed in order to understand Section 12, which describes the optional arguments that can be set by calls to nag_glopt_bnd_mcs_optset_file (e05jcc), nag_glopt_bnd_mcs_optset_string (e05jdc), nag_glopt_bnd_mcs_optset_char (e05jec), nag_glopt_bnd_mcs_optset_int (e05jfc) and/or nag_glopt_bnd_mcs_optset_real (e05jgc).*

11 Algorithmic Details

Here we summarise the main features of the MCS algorithm used in nag_glopt_bnd_mcs_solve (e05jbc), and we introduce some terminology used in the description of the function and its arguments. We assume throughout that we will only do any work in coordinates i in which x_i is free to vary. The MCS algorithm is fully described in Huyer and Neumaier (1999).

11.1 Initialization and Sweeps

Each sub-box is determined by a basepoint **x** and an *opposite point* **y**. We denote such a sub-box by $B[\mathbf{x},\mathbf{y}]$. The basepoint is allowed to belong to more than one sub-box, is usually a boundary point, and is often a vertex.

An *initialization procedure* produces an initial set of sub-boxes. Whenever a sub-box is split along a coordinate i for the first time (in the initialization procedure or later), the splitting is done at three or more user-defined values $\left\{x_i^j\right\}_j$ at which the objective function is sampled, and at some adaptively chosen intermediate points. At least four children are generated. More precisely, we assume that we are given

$$\ell_i \leq x_i^1 < x_i^2 < \cdots < x_i^{L_i} \leq u_i, \quad L_i \geq 3, \qquad \text{for } i = 1, 2, \ldots, n$$

and a vector $\mathbf{p}$ that, for each i, locates within $\left\{x_i^j\right\}_j$ the ith coordinate of an *initial point* $\mathbf{x}^0$; that is, if $x_i^0 = x_i^j$ for some $j = 1, 2, \ldots, L_i$, then $p_i = j$. A good guess for the global optimum can be used as $\mathbf{x}^0$.

The initialization points and the vectors $\boldsymbol{\ell}$ and $\mathbf{p}$ are collectively called the *initialization list* (and sometimes we will refer to just the initialization points as 'the initialization list', whenever this causes no confusion). The initialization data may be input by you, or they can be set to sensible default values by nag_glopt_bnd_mcs_solve (e05jbc): if you provide them yourself, **LIST**(i,j) should contain x_i^j, **numpts**$[i-1]$ should contain L_i, and **initpt**$[i-1]$ should contain p_i, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, L_i$; if you wish nag_glopt_bnd_mcs_solve (e05jbc) to use one of its preset initialization methods, you could choose one of two simple, three-point methods (see Figure 1). If the list generated by one of these methods contains infinite values, attempts are made to generate a safeguarded list using the function subint(x,y) (which is also used during the splitting procedure, and is described in Section 11.2). If infinite values persist, nag_glopt_bnd_mcs_solve (e05jbc) exits with **fail.code** = NE_INF_INIT_LIST. There is also the option to generate an initialization list with the aid of linesearches (by setting **initmethod** = Nag_Linesearch). Starting with the absolutely smallest point in the root box, linesearches are made along each coordinate. For each coordinate, the local minimizers found by the linesearches are put into the initialization list. If there were fewer than three minimizers, they are augmented by close-by values. The final preset initialization option (**initmethod** = Nag_Random) generates a randomized list, so that independent multiple runs may be made if you suspect a global optimum has not been found. Each call to the initialization function nag_glopt_bnd_mcs_init (e05jac) resets the initial-state vector for the Wichmann–Hill base-generator that is used. Depending on whether you set the optional argument **Repeatability** to ON or OFF, the random state is initialized to give a repeatable or non-repeatable sequence. Then, a random integer between 3 and **sdlist** is selected, which is then used to determine the number of points to be generated in each coordinate; that is, **numpts** becomes a constant vector, set to this value. The components of **list** are then generated, from a uniform distribution on the root box if the box is finite, or else in a safeguarded fashion if any bound is infinite. The array **initpt** is set to point to the best point in **list**.

Given an initialization list (preset or otherwise), nag_glopt_bnd_mcs_solve (e05jbc) evaluates F at $\mathbf{x}^0$, and sets the initial estimate of the global minimum, $\mathbf{x}^*$, to $\mathbf{x}^0$. Then, for $i = 1, 2, \ldots, n$, the objective function F is evaluated at $L_i - 1$ points that agree with $\mathbf{x}^*$ in all but the ith coordinate. We obtain pairs $\left(\hat{\mathbf{x}}^j, f_i^j\right)$, for $j = 1, 2, \ldots, L_i$, with: $\mathbf{x}^* = \hat{\mathbf{x}}^{j_1}$, say; with, for $j \neq j_1$,

$$\hat{x}_k^j = \begin{cases} x_k^* & \text{if } k \neq i; \\ x_k^j & \text{otherwise;} \end{cases}$$

and with

$$f_i^j = F\left(\hat{\mathbf{x}}^j\right).$$

The point having the smallest function value is renamed $\mathbf{x}^*$ and the procedure is repeated with the next coordinate.

Once nag_glopt_bnd_mcs_solve (e05jbc) has a full set of initialization points and function values, it can generate an initial set of sub-boxes. Recall that the *root box* is $B[\mathbf{x}, \mathbf{y}] = [\boldsymbol{\ell}, \mathbf{u}]$, having basepoint $\mathbf{x} = \mathbf{x}^0$. The opposite point $\mathbf{y}$ is a corner of $[\boldsymbol{\ell}, \mathbf{u}]$ farthest away from $\mathbf{x}$, in some sense. The point $\mathbf{x}$ need not be a vertex of $[\boldsymbol{\ell}, \mathbf{u}]$, and $\mathbf{y}$ is entitled to have infinite coordinates. We loop over each coordinate i, splitting the current box along coordinate i into $2L_i - 2$, $2L_i - 1$ or $2L_i$ sub-intervals with exactly one of the $\hat{x}_i^j$ as endpoints, depending on whether two, one or none of the $\hat{x}_i^j$ are on the boundary. Thus, as well as splitting at $\hat{x}_i^j$, for $j = 1, 2, \ldots, L_i$, we split at additional points z_i^j, for $j = 2, 3, \ldots, L_i$. These additional z_i^j are such that

$$z_i^j = \hat{x}_i^{j-1} + q^m\left(\hat{x}_i^j - \hat{x}_i^{j-1}\right), \quad j = 2, \ldots, L_i,$$

where q is the golden-section ratio $\left(\sqrt{5} - 1\right)/2$, and the exponent m takes the value 1 or 2, chosen so that the sub-box with the smaller function value gets the larger fraction of the interval. Each child sub-box gets as basepoint the point obtained from $\mathbf{x}^*$ by changing x_i^* to the x_i^j that is a boundary point of the corresponding ith coordinate interval; this new basepoint therefore has function value f_i^j. The opposite point is derived from $\mathbf{y}$ by changing y_i to the other end of that interval.

nag_glopt_bnd_mcs_solve (e05jbc) can now rank the coordinates based on an estimated variability of F. For each i we compute the union of the ranges of the quadratic interpolant through any three consecutive $\hat{x}_i^j$, taking the difference between the upper and lower bounds obtained as a measure of the variability of F in coordinate i. A vector π is populated in such a way that coordinate i has the π_ith highest estimated variability. For tiebreaks, when the $\mathbf{x}^*$ obtained after splitting coordinate i belongs to two sub-boxes, the one that contains the minimizer of the quadratic models is designated the current sub-box for coordinate $i+1$.

Boxes are assigned levels in the following manner. The root box is given level 1. When a sub-box of level s is split, the child with the smaller fraction of the golden-section split receives level $s+2$; all other children receive level $s+1$. The box with the better function value is given the larger fraction of the splitting interval and the smaller level because then it is more likely to be split again more quickly. We see that after the initialization procedure the first level is empty and the non-split boxes have levels $2, \ldots, n_r + 2$, so it is meaningful to choose $s_{\max}$ much larger than n_r. Note that the internal structure of nag_glopt_bnd_mcs_solve (e05jbc) demands that $s_{\max}$ be at least $n_r + 3$.

Examples of initializations in two dimensions are given in Figure 1. In both cases the initial point is $\mathbf{x}^0 = (\boldsymbol{\ell} + \mathbf{u})/2$; on the left the initialization points are

$$\mathbf{x}^1 = \boldsymbol{\ell}, \quad \mathbf{x}^2 = (\boldsymbol{\ell} + \mathbf{u})/2, \quad \mathbf{x}^3 = \mathbf{u},$$

while on the right the points are

$$\mathbf{x}^1 = (5\boldsymbol{\ell} + \mathbf{u})/6, \quad \mathbf{x}^2 = (\boldsymbol{\ell} + \mathbf{u})/2, \quad \mathbf{x}^3 = (\boldsymbol{\ell} + 5\mathbf{u})/6.$$

In Figure 1, basepoints and levels after initialization are displayed. Note that these initialization lists correspond to **initmethod** = Nag_SimpleBdry and **initmethod** = Nag_SimpleOffBdry, respectively.

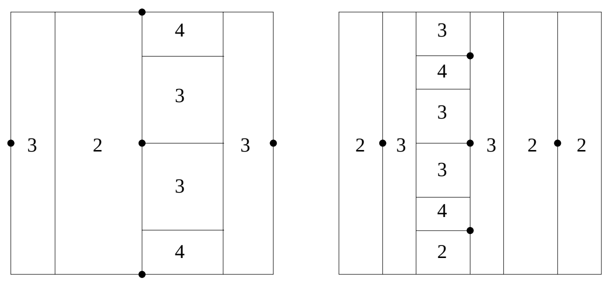


Figure 2
Examples of the initialization procedure

After initialization, a series of *sweeps* through levels is begun. A sweep is defined by three steps:

(i) scan the list of non-split sub-boxes. Fill a *record list* **b** according to $b_s = 0$ if there is no box at level s, and with b_s pointing to a sub-box with the lowest function value among all sub-boxes with level s otherwise, for $0 < s < s_{\max}$;

(ii) the sub-box with label b_s is a candidate for splitting. If the sub-box is not to be split, according to the rules described in Section 11.2, increase its level by 1 and update b_{s+1} if necessary. If the sub-box is split, mark it so, insert its children into the list of sub-boxes, and update **b** if any child with level s' yields a strict improvement of F over those sub-boxes at level s';

(iii) increment s by 1. If $s = s_{\max}$ then displaying monitoring information and start a new sweep; else if $b_s = 0$ then repeat this step; else display monitoring information and go to the previous step.

Clearly, each sweep ends after at most $s_{\max} - 1$ visits of the third step.

11.2 Splitting

Each sub-box is stored by nag_glopt_bnd_mcs_solve (e05jbc) as a set of information about the history of the sub-box: the label of its parent, a label identifying which child of the parent it is, etc. Whenever a sub-box $B[\mathbf{x}, \mathbf{y}]$ of level $s < s_{\max}$ is a candidate for splitting, as described in Section 11.1, we recover $\mathbf{x}$,

y, and the number, n_j, of times coordinate j has been split in the history of B. Sub-box B could be split in one of two ways.

(i) **Splitting by rank**

If $s > 2n_r\big(\min\big(n_j + 1\big)\big)$, the box is always split. The *splitting index* is set to a coordinate i such that $n_i = \min\big(n_j\big)$.

(ii) **Splitting by expected gain**

If $s \leq 2n_r\big(\min\big(n_j + 1\big)\big)$, the sub-box could be split along a coordinate where a maximal gain in function value is expected. This gain is estimated according to a local separable quadratic model obtained by fitting to $2n_r + 1$ function values. If the expected gain is too small the sub-box is not split at all, and its level is increased by 1.

Eventually, a sub-box that is not eligible for splitting by expected gain will reach level $2n_r\big(\min\big(n_j + 1\big)\big) + 1$ and then be split by rank, as long as $s_{\max}$ is large enough. As $s_{\max} \to \infty$, the rule for splitting by rank ensures that each coordinate is split arbitrarily often.

Before describing the details of each splitting method, we introduce the procedure for correctly handling splitting at adaptive points and for dealing with unbounded intervals. Suppose we want to split the ith coordinate interval $\square\{x_i, y_i\}$, where we define $\square\{x_i, y_i\} = [\min(x_i, y_i), \max(x_i, y_i)]$, for $x_i \in R$ and $y_i \in \bar{R}$, and where **x** is the basepoint of the sub-box being considered. The descendants of the sub-box should shrink sufficiently fast, so we should not split too close to x_i. Moreover, if y_i is large we want the new *splitting value* to **not** be too large, so we force it to belong to some smaller interval $\square\{\xi', \xi''\}$, determined by

$$\xi'' = \text{subint}(x_i, y_i), \quad \xi' = x_i + (\xi'' - x_i)/10,$$

where the function subint is defined by

$$\text{subint}(x, y) = \begin{cases} \text{sign}(y) & \text{if } 1000|x| < 1 \text{ and } |y| > 1000; \\ 10\text{sign}(y)|x| & \text{if } 1000|x| \geq 1 \text{ and } |y| > 1000|x|; \\ y & \text{otherwise.} \end{cases}$$

11.2.1 Splitting by rank

Consider a sub-box B with level $s > 2n_r\big(\min\big(n_j + 1\big)\big)$. Although the sub-box has reached a high level, there is at least one coordinate along which it has not been split very often. Among the i such that $n_i = \min\big(n_j\big)$ for B, select the splitting index to be the coordinate with the lowest π_i (and hence highest variability rank). 'Splitting by rank' refers to the ranking of the coordinates by n_i and π_i.

If $n_i = 0$, so that B has never been split along coordinate i, the splitting is done according to the initialization list and the adaptively chosen golden-section split points, as described in Section 11.1. Also as covered there, new basepoints and opposite points are generated. The children having the smaller fraction of the golden-section split (that is, those with larger function values) are given level $\min\{s + 2, s_{\max}\}$. All other children are given level $s + 1$.

Otherwise, B ranges between x_i and y_i in the ith coordinate direction. The splitting value is selected to be $z_i = x_i + 2(\text{subint}(x_i, y_i) - x_i)/3$; we are not attempting to split based on a large reduction in function value, merely in order to reduce the size of a large interval, so z_i may not be optimal. Sub-box B is split at z_i and the golden-section split point, producing three parts and requiring only one additional function evaluation, at the point $\mathbf{x}'$ obtained from **x** by changing the ith coordinate to z_i. The child with the smaller fraction of the golden-section split is given level $\min\{s + 2, s_{\max}\}$, while the other two parts are given level $s + 1$. Basepoints are assigned as follows: the basepoint of the first child is taken to be **x**, and the basepoint of the second and third children is the point $\mathbf{x}'$. Opposite points are obtained by changing y_i to the other end of the ith coordinate-interval of the corresponding child.

11.2.2 Splitting by expected gain

When a sub-box B has level $s \leq 2n_r\big(\min\big(n_j + 1\big)\big)$, we compute the optimal splitting index and splitting value from a local separable quadratic used as a simple local approximation of the objective function. To fit this curve, for each coordinate we need two additional points and their function values. Such data may be recoverable from the history of B: whenever the ith coordinate was split in the history of B, we obtained values that can be used for the current quadratic interpolation in coordinate i.

We loop over i; for each coordinate we pursue the history of B back to the root box, and we take the first two points and function values we find, since these are expected to be closest to the current basepoint $\mathbf{x}$. If the current coordinate has not yet been split we use the initialization list. Then we generate a local separable model $e(\xi)$ for $F(\xi)$ by interpolation at $\mathbf{x}$ and the $2n_r$ additional points just collected:

$$e(\xi) = F(\mathbf{x}) + \sum_{i=1}^{n} e_i(\xi_i).$$

We define the *expected gain* $\hat{e}_i$ in function value when we evaluate at a new point obtained by changing coordinate i in the basepoint, for each i, based on two cases:

(i) $n_i = 0$. We compute the expected gain as

$$\hat{e}_i = \min_{1 \leq j \leq L_i} \left\{ f_i^j \right\} - f_i^{p_i}.$$

Again, we split according to the initialization list, with the new basepoints and opposite points being as before.

(ii) $n_i > 0$. Now, the ith component of our sub-box ranges from x_i to y_i. Using the quadratic partial correction function

$$e_i(\xi_i) = \alpha_i(\xi_i - x_i) + \beta_i(\xi_i - x_i)^2$$

we can approximate the maximal gain expected when changing x_i only. We will choose the splitting value from $\square\{\xi', \xi''\}$. We compute

$$\hat{e}_i = \min_{\xi_i \in \square\{\xi', \xi''\}} e_i(\xi_i)$$

and call z_i the minimizer in $\square\{\xi', \xi''\}$.

If the expected best function value f_{exp} satisfies

$$f_{\text{exp}} = F(\mathbf{x}) + \min_{1 \leq i \leq n} \hat{e}_i < f_{\text{best}}, \qquad (1)$$

where f_{best} is the current best function value (including those function values obtained by local optimization), we expect the sub-box to contain a better point and so we split it, using as splitting index the component with minimal $\hat{e}_i$. Equation (1) prevents wasting function calls by avoiding splitting sub-boxes whose basepoints have bad function values. These sub-boxes will eventually be split by rank anyway.

We now have a splitting index and a splitting value z_i. The sub-box is split at z_i as long as $z_i \neq y_i$, and at the golden-section split point; two or three children are produced. The larger fraction of the golden-section split receives level $s + 1$, while the smaller fraction receives level $\min\{s + 2, s_{\text{max}}\}$. If it is the case that $z_i \neq y_i$ and the third child is larger than the smaller of the two children from the golden-section split, the third child receives level $s + 1$. Otherwise it is given the level $\min\{s + 2, s_{\text{max}}\}$. The basepoint of the first child is set to $\mathbf{x}$, and the basepoint of the second (and third if it exists) is obtained by changing the ith coordinate of $\mathbf{x}$ to z_i. The opposite points are again derived by changing y_i to the other end of the ith coordinate interval of B.

If equation (1) does not hold, we expect no improvement. We do not split, and we increase the level of B by 1.

11.3 Local Search

The local optimization algorithm used by nag_glopt_bnd_mcs_solve (e05jbc) uses linesearches along directions that are determined by minimizing quadratic models, all subject to bound constraints. Triples of vectors are computed using *coordinate searches* based on linesearches. These triples are used in *triple search* procedures to build local quadratic models for F. A trust-region-type approach to minimize these models is then carried out, and more information about the coordinate search and the triple search can be found in Huyer and Neumaier (1999).

The local search starts by looking for better points without being too local, by making a triple search using points found by a coordinate search. This yields a new point and function value, an approximation of the gradient of the objective, and an approximation of the Hessian of the objective. Then the quadratic model for F is minimized over a small box, with the solution to that minimization problem then being used as a linesearch direction to minimize the objective. A measure r is computed to quantify the predictive quality of the quadratic model.

The third stage is the checking of termination criteria. The local search will stop if more than $loclim$ visits to this part of the local search have occurred, where $loclim$ is the value of the optional argument **Local Searches Limit**. If that is not the case, it will stop if the limit on function calls has been exceeded (see the description of the optional argument **Function Evaluations Limit**). The final criterion checks if no improvement can be made to the function value, or whether the approximated gradient $\mathbf{g}$ is small, in the sense that

$$|\mathbf{g}|^{\mathrm{T}} \max(|\mathbf{x}|, |\mathbf{x}_{\mathrm{old}}|) < loctol(f_0 - f).$$

The vector $\mathbf{x}_{\mathrm{old}}$ is the best point at the start of the current loop in this iterative local-search procedure, the constant $loctol$ is the value of the optional argument **Local Searches Tolerance**, f is the objective value at $\mathbf{x}$, and f_0 is the smallest function value found by the initialization procedure.

Next, nag_glopt_bnd_mcs_solve (e05jbc) attempts to move away from the boundary, if any components of the current point lie there, using linesearches along the offending coordinates. Local searches are terminated if no improvement could be made.

The fifth stage carries out another triple search, but this time it does not use points from a coordinate search, rather points lying within the trust-region box are taken.

The final stage modifies the trust-region box to be bigger or smaller, depending on the quality of the quadratic model, minimizes the new quadratic model on that box, and does a linesearch in the direction of the minimizer. The value of r is updated using the new data, and then we go back to the third stage (checking of termination criteria).

The Hessians of the quadratic models generated by the local search may not be positive definite, so nag_glopt_bnd_mcs_solve (e05jbc) uses the general nonlinear optimizer nag_opt_sparse_nlp_solve (e04vhc) to minimize the models.

12 Optional Arguments

Several optional arguments in nag_glopt_bnd_mcs_solve (e05jbc) define choices in the problem specification or the algorithm logic. In order to reduce the number of formal arguments of nag_glopt_bnd_mcs_solve (e05jbc) these optional arguments have associated *default values* that are appropriate for most problems. Therefore, you need only specify those optional arguments whose values are to be different from their default values.

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

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

Defaults
Function Evaluations Limit
Infinite Bound Size
List
Local Searches
Local Searches Limit
Local Searches Tolerance
Maximize
Minimize

Nolist
Repeatability
Splits Limit
Static Limit
Target Objective Error
Target Objective Safeguard
Target Objective Value

Optional arguments may be specified by calling one, or more, of the functions nag_glopt_bnd_mcs_optset_file (e05jcc), nag_glopt_bnd_mcs_optset_string (e05jdc), nag_glopt_bnd_mcs_optset_char (e05jec), nag_glopt_bnd_mcs_optset_int (e05jfc) and nag_glopt_bnd_mcs_optset_real (e05jgc) before a call to nag_glopt_bnd_mcs_solve (e05jbc).

nag_glopt_bnd_mcs_optset_file (e05jcc) reads options from an external options file, with `Begin` and `End` as the first and last lines respectively, and with each intermediate line defining a single optional argument. For example,

```
Begin
  Static Limit = 50
End
```

The call

```
e05jcc (fileid, &state, &fail);
```

can then be used to read the file on the descriptor **fileid** as returned by a call of nag_open_file (x04acc). The value **fail.code** = NE_NOERROR is returned on successful exit. nag_glopt_bnd_mcs_optset_file (e05jcc) should be consulted for a full description of this method of supplying optional arguments.

nag_glopt_bnd_mcs_optset_string (e05jdc), nag_glopt_bnd_mcs_optset_char (e05jec), nag_glopt_bnd_mcs_optset_int (e05jfc) or nag_glopt_bnd_mcs_optset_real (e05jgc) can be called to supply options directly, one call being necessary for each optional argument. nag_glopt_bnd_mcs_optset_string (e05jdc), nag_glopt_bnd_mcs_optset_char (e05jec), nag_glopt_bnd_mcs_optset_int (e05jfc) or nag_glopt_bnd_mcs_optset_real (e05jgc) should be consulted for a full description of this method of supplying optional arguments.

All optional arguments not specified by you are set to their default values. Valid values of optional arguments specified by you are unaltered by nag_glopt_bnd_mcs_solve (e05jbc) and so remain in effect for subsequent calls to nag_glopt_bnd_mcs_solve (e05jbc), unless you explicitly change them.

12.1 Description of the Optional Arguments

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

The summary line contains:

a parameter value, where the letters a, i and r denote options that take character, integer and real values respectively, and where the letter a denotes an option that takes an ON or OFF value;

the default value, where the symbol ϵ is a generic notation for ***machine precision*** (see nag_machine_precision (X02AJC)), the symbol $r_{\max}$ stands for the largest positive model number (see nag_real_largest_number (X02ALC)), n_r represents the number of non-fixed variables, and the symbol d stands for the maximum number of decimal digits that can be represented (see nag_decimal_digits (X02BEC)).

Option names are case-insensitive and must be provided in full; abbreviations are not recognized.

Defaults

This special keyword is used to reset all optional arguments to their default values, and any random state stored in **state** will be destroyed.

Any option value given with this keyword will be ignored. This optional argument cannot be queried or got.

Function Evaluations Limit i Default $= 100n_r^2$

This puts an approximate limit on the number of function calls allowed. The total number of calls made is checked at the top of an internal iteration loop, so it is possible that a few calls more than nf may be made.

Constraint: $nf > 0$.

Infinite Bound Size r Default $= r_{\max}^{\frac{1}{4}}$

This defines the 'infinite' bound $infbnd$ in the definition of the problem constraints. Any upper bound greater than or equal to $infbnd$ will be regarded as ∞ (and similarly any lower bound less than or equal to $-infbnd$ will be regarded as $-\infty$).

Constraint: $r_{\max}^{\frac{1}{4}} \leq infbnd \leq r_{\max}^{\frac{1}{2}}$.

Local Searches a Default $=$ ON

If you want to try to accelerate convergence of nag_glopt_bnd_mcs_solve (e05jbc) by starting local searches from candidate minima, you will require $lcsrch$ to be ON.

Constraint: $lcsrch =$ ON or OFF.

Local Searches Limit i Default $= 50$

This defines the maximal number of iterations to be used in the trust-region loop of the local-search procedure.

Constraint: $loclim > 0$.

Local Searches Tolerance r Default $= 2\epsilon$

The value of $loctol$ is the multiplier used during local searches as a stopping criterion for when the approximated gradient is small, in the sense described in Section 11.3.

Constraint: $loctol \geq 2\epsilon$.

Minimize Default
Maximize

These keywords specify the required direction of optimization. Any option value given with these keywords will be ignored.

Nolist Default
List

These options control the echoing of each optional argument specification as it is supplied. **List** turns printing on, **Nolist** turns printing off. The output is sent to `stdout`.

Any option value given with these keywords will be ignored. This optional argument cannot be queried or got.

Repeatability a Default $=$ OFF

For use with random initialization lists (**initmethod** $=$ Nag_Random). When set to ON, an internally-initialized random state is stored in **state** for use in subsequent calls to nag_glopt_bnd_mcs_solve (e05jbc).

Constraint: $repeat =$ ON or OFF.

Splits Limit i Default $= \lfloor d(n_r + 2)/3 \rfloor$

Along with the initialization list **list**, this defines a limit on the number of times the root box will be split along any single coordinate direction. If **Local Searches** is OFF you may find the default value to be too small.

Constraint: $smax > n_r + 2$.

Static Limit i Default $= 3n_r$

As the default termination criterion, computation stops when the best function value is static for $stclim$ sweeps through levels. This argument is ignored if you have specified a target value to reach in **Target Objective Value**.

Constraint: $stclim > 0$.

Target Objective Error r Default $= \epsilon^{\frac{1}{4}}$

If you have given a target objective value to reach in $objval$ (the value of the optional argument **Target Objective Value**), $objerr$ sets your desired relative error (from above if **Minimize** is set, from below if **Maximize** is set) between **obj** and $objval$, as described in Section 7. See also the description of the optional argument **Target Objective Safeguard**.

Constraint: $objerr \geq 2\epsilon$.

Target Objective Safeguard r Default $= \epsilon^{\frac{1}{2}}$

If you have given a target objective value to reach in $objval$ (the value of the optional argument **Target Objective Value**), $objsfg$ sets your desired safeguarded termination tolerance, for when $objval$ is close to zero.

Constraint: $objsfg \geq 2\epsilon$.

Target Objective Value r

This argument may be set if you wish nag_glopt_bnd_mcs_solve (e05jbc) to use a specific value as the target function value to reach during the optimization. Setting $objval$ overrides the default termination criterion determined by the optional argument **Static Limit**.