



Science and  
Technology  
Facilities Council



# GALAHAD

# DGO

USER DOCUMENTATION

GALAHAD Optimization Library version 5.5

## 1 SUMMARY

This package uses a **deterministic partition-and-bound trust-region method to find an approximation to the global minimizer of a differentiable objective function  $f(\mathbf{x})$  of  $n$  variables  $\mathbf{x}$ , subject to a finite set of simple bounds  $\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u$  on the variables.** The method offers the choice of direct and iterative solution of the key trust-region subproblems, and is suitable for large problems. First derivatives are required, and if second derivatives can be calculated, they will be exploited—if the product of second derivatives with a vector may be found but not the derivatives themselves, that may also be exploited.

Although there are theoretical guarantees, these may require a large number of evaluations as the dimension and nonconvexity increase. The alternative GALAHAD package BGO may sometimes be preferred.

**ATTRIBUTES — Versions:** GALAHAD\_DGO\_single, GALAHAD\_DGO\_double. **Uses:** GALAHAD\_CLOCK, GALAHAD\_SYMBOLS, GALAHAD\_HASH, GALAHAD\_NLPT, GALAHAD\_USERDATA, GALAHAD\_SPECFILE, GALAHAD\_SPACE, GALAHAD\_NORMS, GALAHAD\_UGO and GALAHAD\_TRB. **Date:** July 2021. **Origin:** J. Fowkes and N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

## 2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the USE statement

```
USE GALAHAD_DGO_single
```

with the obvious substitution GALAHAD\_DGO\_double, GALAHAD\_DGO\_quadruple, GALAHAD\_DGO\_single\_64, GALAHAD\_DGO\_double\_64 and GALAHAD\_DGO\_quadruple\_64 for the other variants.

If it is required to use more than one of the modules at the same time, the derived types SMT\_type, USERDATA\_type, DGO\_time\_type, DGO\_control\_type, DGO\_inform\_type, DGO\_data\_type and NLPT\_problem\_type, (Section 2.3) and the subroutines DGO\_initialize, DGO\_solve, DGO\_terminate, (Section 2.4) and DGO\_read\_specfile (Section 2.8) must be renamed on one of the USE statements.

### 2.1 Matrix storage formats

If available, the Hessian matrix  $\mathbf{H} = \nabla_{\mathbf{x}\mathbf{x}} f(\mathbf{x})$  may be stored in a variety of input formats.

#### 2.1.1 Dense storage format

The matrix  $\mathbf{H}$  is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Since  $\mathbf{H}$  is symmetric, only the lower triangular part (that is the part  $h_{ij}$  for  $1 \leq j \leq i \leq n$ ) need be held. In this case the lower triangle should be stored by rows, that is component  $i * (i - 1) / 2 + j$  of the storage array H%val will hold the value  $h_{ij}$  (and, by symmetry,  $h_{ji}$ ) for  $1 \leq j \leq i \leq n$ .

#### 2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the  $l$ -th entry,  $1 \leq l \leq \text{H\%ne}$ , of  $\mathbf{H}$ , its row index  $i$ , column index  $j$  and value  $h_{ij}$ ,  $1 \leq j \leq i \leq n$ , are stored in the  $l$ -th components of the integer arrays H%row, H%col and real array H%val, respectively. Note that only the entries in the lower triangle should be stored.

**All use is subject to the conditions of a BSD-3-Clause License.**

**See <https://www.galahad.rl.ac.uk/download/> for full details.**

### 2.1.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row  $i$  appear directly before those in row  $i + 1$ . For the  $i$ -th row of  $\mathbf{H}$ , the  $i$ -th component of the integer array `H%ptr` holds the position of the first entry in this row, while `H%ptr (n + 1)` holds the total number of entries plus one. The column indices  $j$ ,  $1 \leq j \leq i$ , and values  $h_{ij}$  of the entries in the  $i$ -th row are stored in components  $l = \text{H\%ptr}(i), \dots, \text{H\%ptr}(i + 1) - 1$  of the integer array `H%col`, and real array `H%val`, respectively. Note that as before only the entries in the lower triangle should be stored. For sparse matrices, this scheme almost always requires less storage than its predecessor.

### 2.1.4 Diagonal storage format

If  $\mathbf{H}$  is diagonal (i.e.,  $h_{ij} = 0$  for all  $1 \leq i \neq j \leq n$ ) only the diagonal entries  $h_{ii}$ ,  $1 \leq i \leq n$ , need be stored, and the first  $n$  components of the array `H%val` may be used for the purpose.

## 2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

## 2.3 The derived data types

Seven derived data types are accessible from the package.

### 2.3.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the Hessian matrix  $\mathbf{H}$  if this is available. The components of `SMT_TYPE` used here are:

- `n` is a scalar component of type `INTEGER(ip_)`, that holds the dimension of the matrix.
- `ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of matrix entries.
- `type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.3.2).
- `val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries. Each pair of off-diagonal entries  $h_{ij} = h_{ji}$  of the *symmetric* matrix  $\mathbf{H}$  is represented as a single entry (see §2.1.1–2.1.3). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.
- `row` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the row indices of the entries. (see §2.1.2).
- `col` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the column indices of the entries (see §2.1.2–2.1.3).
- `ptr` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `n + 1`, that may hold the pointers to the first entry in each row (see §2.1.3).

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

### 2.3.2 The derived data type for holding the problem

The derived data type `NLPT_problem_type` is used to hold the problem. The relevant components of `NLPT_problem_type` are:

`n` is a scalar variable of type `INTEGER(ip_)`, that holds the number of optimization variables,  $n$ .

`H` is scalar variable of type `SMT_TYPE` that holds the Hessian matrix  $\mathbf{H}$ . The following components are used here:

`H%type` is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of `H%type` must contain the string `DENSE`. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of `H%type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of `H%type` must contain the string `SPARSE_BY_ROWS`, and for the diagonal storage scheme (see Section 2.1.4), the first eight components of `H%type` must contain the string `DIAGONAL`.

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `H%type`. For example, if `nlp` is of derived type `DGO_problem_type` and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( nlp%H%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.

`H%ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of entries in the **lower triangular** part of  $\mathbf{H}$  in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for any of the other three schemes.

`H%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the **lower triangular** part of the Hessian matrix  $\mathbf{H}$  in any of the storage schemes discussed in Section 2.1.

`H%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of the **lower triangular** part of  $\mathbf{H}$  in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other three schemes.

`H%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of the **lower triangular** part of  $\mathbf{H}$  in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.

`H%ptr` is a rank-one allocatable array of dimension  $n+1$  and type `INTEGER(ip_)`, that holds the starting position of each row of the **lower triangular** part of  $\mathbf{H}$ , as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.

`G` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that holds the gradient  $\mathbf{g}$  of the objective function. The  $j$ -th component of `G`,  $j = 1, \dots, n$ , contains  $\mathbf{g}_j$ . These are equivalently the values  $\mathbf{z}$  of estimates of the dual variables corresponding to the simple bound constraints (see Section 4).

`f` is a scalar variable of type `REAL(rp_)`, that holds the value of the objective function.

`X_l` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that holds the vector of lower bounds  $\mathbf{x}^l$  on the variables. The  $j$ -th component of `X_l`,  $j = 1, \dots, n$ , contains  $\mathbf{x}_j^l$ . Infinite bounds are allowed by setting the corresponding components of `X_l` to any value smaller than `-infinity`, where `infinity` is a component of the control array `control` (see Section 2.3.3).

`X_u` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that holds the vector of upper bounds  $\mathbf{x}^u$  on the variables. The  $j$ -th component of `X_u`,  $j = 1, \dots, n$ , contains  $\mathbf{x}_j^u$ . Infinite bounds are allowed by setting the corresponding components of `X_u` to any value larger than that `infinity`, where `infinity` is a component of the control array `control` (see Section 2.3.3).

---

**All use is subject to the conditions of a BSD-3-Clause License.**

See <https://www.galahad.rl.ac.uk/download/> for full details.

`X` is a rank-one allocatable array of dimension `n` and type `REAL(rp_)`, that holds the values  $\mathbf{x}$  of the optimization variables. The  $j$ -th component of `X`,  $j = 1, \dots, n$ , contains  $x_j$ .

`pname` is a scalar variable of type default `CHARACTER` and length 10, which contains the “name” of the problem for printing. The default “empty” string is provided.

`VNAMES` is a rank-one allocatable array of dimension `n` and type default `CHARACTER` and length 10, whose  $j$ -th entry contains the “name” of the  $j$ -th variable for printing. This is only used if “debug” printing control%print\_level > 4) is requested, and will be ignored if the array is not allocated.

### 2.3.3 The derived data type for holding control parameters

The derived data type `DGO_control_type` is used to hold controlling data. Default values may be obtained by calling `DGO_initialize` (see Section 2.4.1), while components may also be changed by calling `GALAHAD_DGO_read_spec` (see Section 2.8.1). The components of `DGO_control_type` are:

`error` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for error messages. Printing of error messages in `DGO_solve` and `DGO_terminate` is suppressed if `error`  $\leq 0$ . The default is `error` = 6.

`out` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for informational messages. Printing of informational messages in `DGO_solve` is suppressed if `out` < 0. The default is `out` = 6.

`print_level` is a scalar variable of type `INTEGER(ip_)`, that is used to control the amount of informational output which is required. No informational output will occur if `print_level`  $\leq 0$ . If `print_level` = 1, a single line of output will be produced for each iteration of the process. If `print_level`  $\geq 2$ , this output will be increased to provide significant detail of each iteration. The default is `print_level` = 0.

`start_print` is a scalar variable of type `INTEGER(ip_)`, that specifies the first iteration for which printing will occur in `DGO_solve`. If `start_print` is negative, printing will occur from the outset. The default is `start_print` = -1.

`stop_print` is a scalar variable of type `INTEGER(ip_)`, that specifies the last iteration for which printing will occur in `DGO_solve`. If `stop_print` is negative, printing will occur once it has been started by `start_print`. The default is `stop_print` = -1.

`print_gap` is a scalar variable of type `INTEGER(ip_)`. Once printing has been started, output will occur once every `print_gap` iterations. If `print_gap` is no larger than 1, printing will be permitted on every iteration. The default is `print_gap` = 1.

`maxit` is a scalar variable of type `INTEGER(ip_)`, that holds the maximum number of iterations which will be allowed in `DGO_solve`. The default is `maxit` = 1000.

`max_evals` is a scalar variable of type `INTEGER(ip_)`, that gives the maximum number of function evaluations that are allowed. The default is `max_evals` = 10000.

`dictionary_size` is a scalar variable of type `INTEGER(ip_)`, that gives the size of the initial hash dictionary. The default is `dictionary_size` = 100000.

`alive_unit` is a scalar variable of type `INTEGER(ip_)`. If `alive_unit` > 0, a temporary file named `alive_file` (see below) will be created on stream number `alive_unit` on initial entry to `GALAHAD_DGO_solve`, and execution of `GALAHAD_DGO_solve` will continue so long as this file continues to exist. Thus, a user may terminate execution simply by removing the temporary file from this unit. If `alive_unit`  $\leq 0$ , no temporary file will be created, and execution cannot be terminated in this way. The default is `alive_unit` = 60.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
 See <https://www.galahad.rl.ac.uk/download/> for full details.

`infinity` is a scalar variable of type `REAL(rp_)`, that is used to specify which constraint bounds are infinite. Any bound larger than `infinity` in modulus will be regarded as infinite. The default is `infinity = 1019`.

`lipschitz_lower_bound` is a scalar variable of type `REAL(rp_)`, that provides a lower bound on the Lipschitz constant for the gradient. This must be non-negative (and not zero unless the function is constant). The default is `lipschitz_lower_bound = 10-6`.

`lipschitz_reliability` and `lipschitz_control` are scalar variables of type `REAL(rp_)`, that are used to provide a reliable estimate of the Lipschitz constant in the current sub-box. Specifically, the Lipschitz constant used will be `lipschitz_reliability + max(1, n - 1) * lipschitz_control / iteration counter times the largest value observed`. The defaults are `lipschitz_reliability = 2.0` and `lipschitz_control = 50`.

`stop_length` is a scalar variable of type `REAL(rp_)`, that is used to stop the iteration. This will happen if the length of the “diagonal” in the sub-box with the smallest-found objective function is smaller than `stop_length` times that of the original bound box. The default is `stop_length = 10-4`.

`stop_f` is a scalar variable of type `REAL(rp_)`, that is used to stop the iteration. This will happen if the gap between the best objective value found and the smallest lower bound is smaller than `stop_f`. The default is `stop_f = 10-4`.

`obj_unbounded` is a scalar variable of type default `REAL(rp_)`, that specifies smallest value of the objective function that will be tolerated before the problem is declared to be unbounded from below. The default is `potential_unbounded = -u-2`, where `u` is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_DGO_double`).

`cpu_time_limit` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted CPU time. Any negative value indicates no limit will be imposed. The default is `cpu_time_limit = -1.0`.

`clock_time_limit` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted elapsed system clock time. Any negative value indicates no limit will be imposed. The default is `clock_time_limit = -1.0`.

`hessian_available` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if the user will provide second derivatives (either by providing an appropriate evaluation routine to the solver or by reverse communication, see Section 2.6), and `.FALSE.` if the second derivatives are not explicitly available. The default is `hessian_available = .TRUE..`

`prune` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if sub-boxes that cannot contain the global minimizer be pruned (i.e., removed from further consideration), and `.FALSE.` if a no local pruning is required. The default is `prune = .TRUE..`

`perform_local_optimization` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if approximate minimizers are to be improved by judicious local minimization, and `.FALSE.` if a no local improvement is required. The default is `perform_local_optimization = .TRUE..`

`space_critical` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if space is critical when allocating arrays and `.FALSE.` otherwise. The package may run faster if `space_critical` is `.FALSE.` but at the possible expense of a larger storage requirement. The default is `space_critical = .FALSE..`

`deallocate_error_fatal` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the user wishes to terminate execution if a deallocation fails, and `.FALSE.` if an attempt to continue will be made. The default is `deallocate_error_fatal = .FALSE..`

`alive_file` is a scalar variable of type default `CHARACTER` and length 30, that gives the name of the temporary file whose removal from stream number `alive_unit` terminates execution of `GALAHAD_DGO_solve`. The default is `alive_unit = ALIVE.d`.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

`prefix` is a scalar variable of type default `CHARACTER` and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM( prefix ))-1)`, thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default `prefix = ""`.

`HASH_control` is a scalar variable of type `HASH_control_type` whose components are used to control the the hash table used to store the dictionary of vertices of the sub-boxes maintained by the package `GALAHAD_HASH`. See the specification sheet for the package `GALAHAD_HASH` for details, and appropriate default values.

`TRB_control` is a scalar variable of type `TRB_control_type` whose components are used to control the local multivariate optimization aspects of the calculation, as performed by the package `GALAHAD_TRB`. See the specification sheet for the package `GALAHAD_TRB` for details, and appropriate default values.

`UGO_control` is a scalar variable of type `UGO_control_type` whose components are used to control the univariate global optimization calculation (if any), performed by the package `GALAHAD_UGO`. See the specification sheet for the package `GALAHAD_UGO` for details, and appropriate default values.

### 2.3.4 The derived data type for holding timing information

The derived data type `DGO_time_type` is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of `DGO_time_type` are:

`total` is a scalar variable of type default `REAL`, that gives the CPU total time spent in the package.

`univariate_global` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent performing univariate global optimization.

`multivariate_local` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent performing multivariate local optimization.

`clock_total` is a scalar variable of type default `REAL`, that gives the total elapsed system clock time spent in the package.

`clock_univariate_global` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent performing univariate global optimization.

`clock_multivariate_local` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent performing multivariate local optimization.

### 2.3.5 The derived data type for holding informational parameters

The derived data type `DGO_inform_type` is used to hold parameters that give information about the progress and needs of the algorithm. The components of `DGO_inform_type` are:

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the exit status of the algorithm. See Sections 2.6 and 2.7 for details.

`alloc_status` is a scalar variable of type `INTEGER(ip_)`, that gives the status of the last attempted array allocation or deallocation. This will be 0 if `status = 0`.

`bad_alloc` is a scalar variable of type default `CHARACTER` and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if `status = 0`.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

`f_eval` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of objective function evaluations performed.

`g_eval` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of objective function gradient evaluations performed.

`h_eval` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of objective function Hessian evaluations performed.

`obj` is a scalar variable of type `REAL(rp_)`, that holds the value of the objective function at the best estimate of the solution found.

`norm_pg` is a scalar variable of type `REAL(rp_)`, that holds the value of the norm of the projected gradient of the objective function at the best estimate of the solution found.

`length_ratio` is a scalar variable of type `REAL(rp_)`, that holds the ratio of the final to the initial box lengths.

`f_gap` is a scalar variable of type `REAL(rp_)`, that holds the gap between the best objective value found and the lowest bound.

`why_stop` is a scalar variable of type default `CHARACTER` and length 1 that summarises why the iteration stopped. It will be 'D' if the box length is small enough, 'F' if the objective gap is small enough, and ' ' otherwise.

`time` is a scalar variable of type `DGO_time_type` whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.3.4).

`HASH_inform` is a scalar variable of type `HASH_inform_type` whose components give information about the hash table used to store the dictionary of vertices of the sub-boxes maintained by the package `GALAHAD_HASH`. See the specification sheet for the package `GALAHAD_HASH` for details.

`TRB_inform` is a scalar variable of type `TRB_inform_type` whose components give information about the progress and needs of the local multivariate optimization stages of the algorithm performed by the package `GALAHAD_TRB`. See the specification sheet for the package `GALAHAD_TRB` for details.

`UGO_inform` is a scalar variable of type `UGO_inform_type` whose components give information about the progress and needs of the univariate global optimization stages of the algorithm performed by the package `GALAHAD_UGO`. See the specification sheet for the package `GALAHAD_UGO` for details.

### 2.3.6 The derived data type for holding problem data

The derived data type `DGO_data_type` is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of DGO procedures. This data should be preserved, untouched (except as directed on return from `GALAHAD_DGO_solve` with positive values of `inform%status`, see Section 2.6), from the initial call to `DGO_initialize` to the final call to `DGO_terminate`.

### 2.3.7 The derived data type for holding user data

The derived data type `USERDATA_type` is available to allow the user to pass data to and from user-supplied subroutines for function and derivative calculations (see Section 2.5). Components of variables of type `USERDATA_type` may be allocated as necessary. The following components are available:

`integer` is a rank-one allocatable array of type `INTEGER(ip_)`.

`real` is a rank-one allocatable array of type default `REAL(rp_)`

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

`complex` is a rank-one allocatable array of type default COMPLEX (double precision complex in GALAHAD\_DGO\_double).

`character` is a rank-one allocatable array of type default CHARACTER.

`logical` is a rank-one allocatable array of type default LOGICAL.

`integer_pointer` is a rank-one pointer array of type INTEGER (ip\_).

`real_pointer` is a rank-one pointer array of type default REAL (rp\_)

`complex_pointer` is a rank-one pointer array of type default COMPLEX (double precision complex in GALAHAD\_DGO\_double).

`character_pointer` is a rank-one pointer array of type default CHARACTER.

`logical_pointer` is a rank-one pointer array of type default LOGICAL.

## 2.4 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.8 for further features):

1. The subroutine `DGO_initialize` is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.
2. The subroutine `DGO_solve` is called to solve the problem.
3. The subroutine `DGO_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `DGO_solve`, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since `DGO_initialize` cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [ ] to indicate OPTIONAL arguments.

### 2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL DGO_initialize( data, control, inform )
```

`data` is a scalar INTENT(INOUT) argument of type `DGO_data_type` (see Section 2.3.6). It is used to hold data about the problem being solved.

`control` is a scalar INTENT(OUT) argument of type `DGO_control_type` (see Section 2.3.3). On exit, `control` contains default values for the components as described in Section 2.3.3. These values should only be changed after calling `DGO_initialize`.

`inform` is a scalar INTENT(OUT) argument of type `DGO_inform_type` (see Section 2.3.5). A successful call to `DGO_initialize` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.7.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

### 2.4.2 The minimization subroutine

The minimization algorithm is called as follows:

```
CALL DGO_solve( nlp, control, inform, data, userdata[, eval_F, eval_G,      &
               eval_H, eval_HPROD, eval_SHPROD, eval_PREC] )
```

`nlp` is a scalar `INTENT (INOUT)` argument of type `NLPT_problem_type` (see Section 2.3.2). It is used to hold data about the problem being solved. For a new problem, the user must allocate all the array components, and set values for `nlp%n` and the required integer components of `nlp%H` if second derivatives will be used. Users are free to choose whichever of the matrix formats described in Section 2.1 is appropriate for  $H$  for their application.

The component `nlp%X` must be set to an initial estimate,  $\mathbf{x}^0$ , of the minimization variables. A good choice will increase the speed of the package, but the underlying method is designed to converge (at least to a local solution) from an arbitrary initial guess.

On exit, the component `nlp%X` will contain the best estimates of the minimization variables  $\mathbf{x}$ , while `nlp%G` will contain the best estimates of the dual variables  $\mathbf{z}$ .

**Restrictions:** `nlp%n` > 0 and `nlp%H%type`  $\in \{ 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL' \}$ .

`control` is a scalar `INTENT (IN)` argument of type `DGO_control_type` (see Section 2.3.3). Default values may be assigned by calling `DGO_initialize` prior to the first call to `DGO_solve`.

`inform` is a scalar `INTENT (INOUT)` argument of type `DGO_inform_type` (see Section 2.3.5). On initial entry, the component `status` must be set to the value 1. Other entries need not be set. A successful call to `DGO_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see Sections 2.6 and 2.7.

`data` is a scalar `INTENT (INOUT)` argument of type `DGO_data_type` (see Section 2.3.6). It is used to hold data about the problem being solved. With the possible exceptions of the components `eval_status` and `U` (see Section 2.6), it must not have been altered **by the user** since the last call to `DGO_initialize`.

`userdata` is a scalar `INTENT (INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the OPTIONAL subroutines `eval_F`, `eval_G`, `eval_H` and `eval_HPROD` (see Section 2.3.7).

`eval_F` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the objective function  $f(\mathbf{x})$  at a given vector  $\mathbf{x}$ . See Section 2.5.1 for details. If `eval_F` is present, it must be declared `EXTERNAL` in the calling program. If `eval_F` is absent, `GALAHAD_DGO_solve` will use reverse communication to obtain objective function values (see Section 2.6).

`eval_G` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the gradient of the objective function  $\nabla_{\mathbf{x}} f(\mathbf{x})$  at a given vector  $\mathbf{x}$ . See Section 2.5.2 for details. If `eval_G` is present, it must be declared `EXTERNAL` in the calling program. If `eval_G` is absent, `GALAHAD_DGO_solve` will use reverse communication to obtain gradient values (see Section 2.6).

`eval_H` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the Hessian of the objective function  $\nabla_{\mathbf{xx}} f(\mathbf{x})$  at a given vector  $\mathbf{x}$ . See Section 2.5.3 for details. If `eval_H` is present, it must be declared `EXTERNAL` in the calling program. If `eval_H` is absent, `GALAHAD_DGO_solve` will use reverse communication to obtain Hessian function values (see Section 2.6).

`eval_HPROD` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product  $\nabla_{\mathbf{xx}} f(\mathbf{x})\mathbf{v}$  of the Hessian of the objective function  $\nabla_{\mathbf{xx}} f(\mathbf{x})$  with a given vector  $\mathbf{v}$ . See Section 2.5.4 for details. If `eval_HPROD` is present, it must be declared `EXTERNAL` in the calling program. If `eval_HPROD` is absent, `GALAHAD_DGO_solve` will use reverse communication to obtain Hessian-vector products (see Section 2.6).

---

**All use is subject to the conditions of a BSD-3-Clause License.**

See <https://www.galahad.rl.ac.uk/download/> for full details.

`eval_SHPROD` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product  $\nabla_{xx}f(\mathbf{x})\mathbf{v}$  of the Hessian of the objective function  $\mathbf{u} = \nabla_{xx}f(\mathbf{x})$  with a given *sparse* vector  $\mathbf{v}$ , and to return the nonzero components of the resulting  $\mathbf{u}$ . See Section 2.5.5 for details. If `eval_SHPROD` is present, it must be declared `EXTERNAL` in the calling program. If `eval_SHPROD` is absent, `GALAHAD_DGO_solve` will use reverse communication to obtain Hessian-sparse-vector products (see Section 2.6).

`eval_PREC` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product  $\mathbf{P}(\mathbf{x})\mathbf{v}$  of the user's preconditioner with a given vector  $\mathbf{v}$ . See Section 2.5.6 for details. If `eval_PREC` is present, it must be declared `EXTERNAL` in the calling program. If `eval_PREC` is absent, `GALAHAD_DGO_solve` will use reverse communication to obtain products with the preconditioner (see Section 2.6).

### 2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL DGO_terminate( data, control, inform )
```

`data` is a scalar `INTENT (INOUT)` argument of type `DGO_data_type` exactly as for `DGO_solve`, which must not have been altered **by the user** since the last call to `DGO_initialize`. On exit, array components will have been deallocated.

`control` is a scalar `INTENT (IN)` argument of type `DGO_control_type` exactly as for `DGO_solve`.

`inform` is a scalar `INTENT (OUT)` argument of type `DGO_inform_type` exactly as for `DGO_solve`. Only the component `status` will be set on exit, and a successful call to `DGO_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see Section 2.7.

## 2.5 Function and derivative values

### 2.5.1 The objective function value via internal evaluation

If the argument `eval_F` is present when calling `GALAHAD_DGO_solve`, the user is expected to provide a subroutine of that name to evaluate the value of the objective function  $f(\mathbf{x})$ . The routine must be specified as

```
SUBROUTINE eval_F( status, X, userdata, f )
```

whose arguments are as follows:

`status` is a scalar `INTENT (OUT)` argument of type `INTEGER(ip_)`, that should be set to 0 if the routine has been able to evaluate the objective function and to a non-zero value if the evaluation has not been possible.

`X` is a rank-one `INTENT (IN)` array argument of type `REAL(rp_)` whose components contain the vector  $\mathbf{x}$ .

`userdata` is a scalar `INTENT (INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_F`, `eval_G`, `eval_H`, `eval_HPROD` and `eval_PREC` (see Section 2.3.7).

`f` is a scalar `INTENT (OUT)` argument of type `REAL(rp_)`, that should be set to the value of the objective function  $f(\mathbf{x})$  evaluated at the vector  $\mathbf{x}$  input in `X`.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

### 2.5.2 Gradient values via internal evaluation

If the argument `eval_G` is present when calling `GALAHAD_DGO_solve`, the user is expected to provide a subroutine of that name to evaluate the value of the gradient the objective function  $\nabla_{\mathbf{x}}f(\mathbf{x})$ . The routine must be specified as

```
SUBROUTINE eval_G( status, X, userdata, G )
```

whose arguments are as follows:

`status` is a scalar `INTENT(OUT)` argument of type `INTEGER(ip_)`, that should be set to 0 if the routine has been able to evaluate the gradient of the objective function and to a non-zero value if the evaluation has not been possible.

`X` is a rank-one `INTENT(IN)` array argument of type `REAL(rp_)` whose components contain the vector  $\mathbf{x}$ .

`userdata` is a scalar `INTENT(INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_F`, `eval_G`, `eval_H`, `eval_HPROD` and `eval_PREC` (see Section 2.3.7).

`G` is a rank-one `INTENT(OUT)` argument of type `REAL(rp_)`, whose components should be set to the values of the gradient of the objective function  $\nabla_{\mathbf{x}}f(\mathbf{x})$  evaluated at the vector  $\mathbf{x}$  input in `X`.

### 2.5.3 Hessian values via internal evaluation

If the argument `eval_H` is present when calling `GALAHAD_DGO_solve`, the user is expected to provide a subroutine of that name to evaluate the values of the Hessian of the objective function  $\nabla_{\mathbf{xx}}f(\mathbf{x})$ . The routine must be specified as

```
SUBROUTINE eval_H( status, X, userdata, Hval )
```

whose arguments are as follows:

`status` is a scalar `INTENT(OUT)` argument of type `INTEGER(ip_)`, that should be set to 0 if the routine has been able to evaluate the Hessian of the objective function and to a non-zero value if the evaluation has not been possible.

`X` is a rank-one `INTENT(IN)` array argument of type `REAL(rp_)` whose components contain the vector  $\mathbf{x}$ .

`userdata` is a scalar `INTENT(INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_F`, `eval_G`, `eval_H`, `eval_HPROD` and `eval_PREC` (see Section 2.3.7).

`Hval` is a scalar `INTENT(OUT)` argument of type `REAL(rp_)`, whose components should be set to the values of the Hessian of the objective function  $\nabla_{\mathbf{xx}}f(\mathbf{x})$  evaluated at the vector  $\mathbf{x}$  input in `X`. The values should be input in the same order as that in which the array indices were given in `nlp%H`.

### 2.5.4 Hessian-vector products via internal evaluation

If the argument `eval_HPROD` is present when calling `GALAHAD_DGO_solve`, the user is expected to provide a subroutine of that name to evaluate the sum  $\mathbf{u} + \nabla_{\mathbf{xx}}f(\mathbf{x})\mathbf{v}$  involving the product of the Hessian of the objective function  $\nabla_{\mathbf{xx}}f(\mathbf{x})$  with a given vector  $\mathbf{v}$ . The routine must be specified as

```
SUBROUTINE eval_HPROD( status, X, userdata, U, V, got_h )
```

whose arguments are as follows:

`status` is a scalar `INTENT(OUT)` argument of type `INTEGER(ip_)`, that should be set to 0 if the routine has been able to evaluate the sum  $\mathbf{u} + \nabla_{\mathbf{xx}}f(\mathbf{x})\mathbf{v}$  and to a non-zero value if the evaluation has not been possible.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

- X** is a rank-one INTENT (IN) array argument of type REAL (rp\_) whose components contain the vector  $\mathbf{x}$ .
- userdata** is a scalar INTENT (INOUT) argument of type USERDATA\_type whose components may be used to communicate user-supplied data to and from the subroutines eval\_F, eval\_G, eval\_H, eval\_HPROD and eval\_PREC (see Section 2.3.7).
- U** is a rank-one INTENT (INOUT) array argument of type REAL (rp\_) whose components on input contain the vector  $\mathbf{u}$  and on output the sum  $\mathbf{u} + \nabla_{xx}f(\mathbf{x})\mathbf{v}$ .
- V** is a rank-one INTENT (IN) array argument of type REAL (rp\_) whose components contain the vector  $\mathbf{v}$ .
- got\_h** is an OPTIONAL scalar INTENT (IN) argument of type default LOGICAL. If the Hessian has already been evaluated at the current  $\mathbf{x}$  got\_h will be PRESENT and set .TRUE.; if this is the first time the Hessian is to be accessed at  $\mathbf{x}$ , either got\_h will be absent or PRESENT and set .FALSE.. This gives the user the opportunity to reuse “start-up” computations required for the first instance of  $\mathbf{x}$  to speed up subsequent products.

### 2.5.5 Hessian-sparse-vector products via internal evaluation

If the argument eval\_SHPROD is present when calling GALAHAD\_DGO\_solve, the user is expected to provide a subroutine of that name to evaluate the product  $\mathbf{u} = \nabla_{xx}f(\mathbf{x})\mathbf{v}$  involving the Hessian of the objective function  $\nabla_{xx}f(\mathbf{x})$  and a given sparse vector  $\mathbf{v}$ , and to return the nonzero components of the result  $\mathbf{u}$ . This routine is **not required** if the user has set control%hessian\_available to .TRUE. and has made the values of  $\nabla_{xx}f(\mathbf{x})$  available either by calls to eval\_H (see §2.5.3) or by reverse communication (see §2.6). If needed, the routine must be specified as

```
SUBROUTINE eval_SHPROD( status, X, userdata, nnz_v, INDEX_nz_v, V,      &
                        nnz_u, INDEX_nz_u, U, got_h )
```

whose arguments are as follows:

- status** is a scalar INTENT (OUT) argument of type INTEGER(ip\_), that should be set to 0 if the routine has been able to evaluate the sum  $\mathbf{u} + \nabla_{xx}f(\mathbf{x})\mathbf{v}$  and to a non-zero value if the evaluation has not been possible.
- X** is a rank-one INTENT (IN) array argument of type REAL (rp\_) whose components contain the vector  $\mathbf{x}$ .
- userdata** is a scalar INTENT (INOUT) argument of type USERDATA\_type whose components may be used to communicate user-supplied data to and from the subroutines eval\_F, eval\_G, eval\_H, eval\_HPROD and eval\_PREC (see Section 2.3.7).
- nnz\_v** is a scalar INTENT (IN) argument of type INTEGER(ip\_), that specifies the number of nonzeros in the input sparse vector  $\mathbf{v}$ .
- INDEX\_nz\_v** is a rank-one INTENT (IN) array argument of length at least nnz\_v and type INTEGER(ip\_) whose first nnz\_v components give the indices of the nonzero components of  $\mathbf{v}$ .
- V** is a rank-one INTENT (IN) array argument of type REAL (rp\_) whose components INDEX\_nz\_v(i),  $i = 1, \dots, \text{nnz}_v$ , hold the nonzero values of  $\mathbf{v}$ . Any other components should be ignored.
- nnz\_u** is a scalar INTENT (OUT) argument of type INTEGER(ip\_), that gives the number of nonzeros in the output vector  $\mathbf{u}$ .
- INDEX\_nz\_u** is a rank-one INTENT (OUT) array argument of length at least nnz\_u and type INTEGER(ip\_) whose first nnz\_u components give the indices of the nonzero components of the computed product  $\mathbf{u}$ .
- U** is a rank-one INTENT (OUT) array argument of type REAL (rp\_) whose components INDEX\_nz\_u(i),  $i = 1, \dots, \text{nnz}_u$ , hold the nonzero values of  $\mathbf{u}$ . The remaining components should be ignored.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

`got_h` is an OPTIONAL scalar INTENT (IN) argument of type default LOGICAL. If the Hessian has already been evaluated at the current  $\mathbf{x}$  `got_h` will be PRESENT and set .TRUE.; if this is the first time the Hessian is to be accessed at  $\mathbf{x}$ , either `got_h` will be absent or PRESENT and set .FALSE.. This gives the user the opportunity to reuse “start-up” computations required for the first instance of  $\mathbf{x}$  to speed up subsequent products.

### 2.5.6 Preconditioner-vector products via internal evaluation

If the argument `eval_PREC` is present when calling `GALAHAD_DGO_solve`, the user is expected to provide a subroutine of that name to evaluate the product  $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$  involving the user’s preconditioner  $\mathbf{P}(\mathbf{x})$  with a given vector  $\mathbf{v}$ . The symmetric matrix  $\mathbf{P}(\mathbf{x})$  should ideally be chosen so that the eigenvalues of  $\mathbf{P}(\mathbf{x})(\nabla_{xx}f(\mathbf{x}))^{-1}$  are clustered. This subroutine will **only be required** if `control%norm = -3`, and the user prefers a subroutine call to that provided by reverse communication with `inform%status = 6` (see §2.6). The routine must be specified as

```
SUBROUTINE eval_PREC( status, X, userdata, U, V )
```

whose arguments are as follows:

`status` is a scalar INTENT (OUT) argument of type INTEGER (ip\_) , that should be set to 0 if the routine has been able to evaluate the product  $\mathbf{P}(\mathbf{x})\mathbf{v}$  and to a non-zero value if the evaluation has not been possible.

`X` is a rank-one INTENT (IN) array argument of type REAL (rp\_) whose components contain the vector  $\mathbf{x}$ .

`userdata` is a scalar INTENT (INOUT) argument of type USERDATA\_type whose components may be used to communicate user-supplied data to and from the subroutines `eval_F`, `eval_G`, `eval_H` and `eval_PREC` (see Section 2.3.7).

`U` is a rank-one INTENT (OUT) array argument of type REAL (rp\_) whose components on output should contain the product sum  $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$ .

`V` is a rank-one INTENT (IN) array argument of type REAL (rp\_) whose components contain the vector  $\mathbf{v}$ .

## 2.6 Reverse Communication Information

A positive value of `inform%status` on exit from `DGO_solve` indicates that `GALAHAD_DGO_solve` is seeking further information—this will happen if the user has chosen not to evaluate function or derivative values internally (see Section 2.5). The user should compute the required information and re-enter `GALAHAD_DGO_solve` with `inform%status` and all other arguments (except those specifically mentioned below) unchanged.

Possible values of `inform%status` and the information required are

2. The user should compute the objective function value  $f(\mathbf{x})$  at the point  $\mathbf{x}$  indicated in `nlp%X`. The required value should be set in `nlp%f`, and `data%eval_status` should be set to 0. If the user is unable to evaluate  $f(\mathbf{x})$ —for instance, if the function is undefined at  $\mathbf{x}$ —the user need not set `nlp%f`, but should then set `data%eval_status` to a non-zero value.
3. The user should compute the gradient of the objective function  $\nabla_{\mathbf{x}}f(\mathbf{x})$  at the point  $\mathbf{x}$  indicated in `nlp%X`. The value of the  $i$ -th component of the gradient should be set in `nlp%G(i)`, for  $i = 1, \dots, n$  and `data%eval_status` should be set to 0. If the user is unable to evaluate a component of  $\nabla_{\mathbf{x}}f(\mathbf{x})$ —for instance, if a component of the gradient is undefined at  $\mathbf{x}$ —the user need not set `nlp%G`, but should then set `data%eval_status` to a non-zero value.
4. The user should compute the Hessian of the objective function  $\nabla_{xx}f(\mathbf{x})$  at the point  $\mathbf{x}$  indicated in `nlp%X`. The value  $l$ -th component of the Hessian stored according to the scheme input in the remainder of `nlp%H` (see Section 2.3.2) should be set in `nlp%H%val(l)`, for  $l = 1, \dots, \text{nlp}\%H\%ne$  and `data%eval_status` should be set to 0. If the user is unable to evaluate a component of  $\nabla_{xx}f(\mathbf{x})$ —for instance, if a component of the Hessian is undefined at  $\mathbf{x}$ —the user need not set `nlp%H%val`, but should then set `data%eval_status` to a non-zero value.

---

**All use is subject to the conditions of a BSD-3-Clause License.**

See <https://www.galahad.rl.ac.uk/download/> for full details.

5. The user should compute the product  $\nabla_{xx}f(\mathbf{x})\mathbf{v}$  of the Hessian of the objective function  $\nabla_{xx}f(\mathbf{x})$  at the point  $\mathbf{x}$  indicated in `nlp%X` with the vector  $\mathbf{v}$  and add the result to the vector  $\mathbf{u}$ . The vectors  $\mathbf{u}$  and  $\mathbf{v}$  are given in `data%U` and `data%V` respectively, the resulting vector  $\mathbf{u} + \nabla_{xx}f(\mathbf{x})\mathbf{v}$  should be set in `data%U` and `data%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Hessian is undefined at  $\mathbf{x}$ —the user need not set `nlp%H%val`, but should then set `data%eval_status` to a non-zero value.
6. The user should compute the product  $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$  of their preconditioner  $\mathbf{P}(\mathbf{x})$  at the point  $\mathbf{x}$  indicated in `nlp%X` with the vector  $\mathbf{v}$ . The vectors  $\mathbf{v}$  is given in `data%V`, the resulting vector  $\mathbf{u} = \mathbf{P}(\mathbf{x})\mathbf{v}$  should be set in `data%U` and `data%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the preconditioner is undefined at  $\mathbf{x}$ —the user need not set `data%U`, but should then set `data%eval_status` to a non-zero value.  
  
This value **can only occur** if the user has set `control%norm = -3`, and has not provided an optional subroutine `eval_PREC` (see §2.5.6) to compute the required product with the preconditioner.
7. The user should compute the product  $\mathbf{h} = \nabla_{xx}f(\mathbf{x})\mathbf{p}$  of the Hessian of the objective function  $\nabla_{xx}f(\mathbf{x})$  at the point  $\mathbf{x}$  indicated in `nlp%X` with the sparse vector  $\mathbf{p}$ . The nonzeros of  $\mathbf{p}$  are stored in `data%P` (`data%INDEX_nz_p` (`data%nnz_p_l: data%nnz_p_u`)) while the nonzeros of  $\mathbf{h}$  should be returned in `data%HP` (`data%INDEX_nz_hp` (`1: data%nnz_hp`)); the user must set `data%nnz_hp` and `data%INDEX_nz_hp` accordingly, and `data%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Hessian is undefined at  $\mathbf{x}$ —the user need not set `data%HP`, `data%INDEX_nz_hp` and `data%nnz_hp` but should then set `data%eval_status` to a non-zero value.  
  
This value **will not occur** if the user has set `control%hessian_available` to `.TRUE.` and can provide values of  $\nabla_{xx}f(\mathbf{x})$  either by calls to `eval_H` (see §2.5.3) or by reverse communication (see `inform%status = 4`, above).
23. The user should follow the instructions for 2 and 3 above before returning.
25. The user should follow the instructions for 2 and 5 above before returning.
35. The user should follow the instructions for 3 and 5 above before returning.
235. The user should follow the instructions for 2, 3 and 5 above before returning.

## 2.7 Warning and error messages

A negative value of `inform%status` on exit from `DGO_solve` or `DGO_terminate` indicates that an error has occurred (but note that the value `-99` that indicates the evaluation budget has been achieved is highly likely to occur). No further calls should be made until the error has been corrected. Possible values are:

- 1. An allocation error occurred. A message indicating the offending array is written on unit `control%error`, and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc`, respectively.
- 2. A deallocation error occurred. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc`, respectively.
- 3. The restriction `nlp%n > 0` or requirement that `nlp%H_type` contains its relevant string 'DENSE', 'COORDINATE', 'SPARSE-BY-ROWS' or 'DIAGONAL' has been violated.
- 4. The bound constraints are inconsistent or infinite.
- 7. The objective function appears to be unbounded from below on the feasible set.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
See <https://www.galahad.rl.ac.uk/download/> for full details.

- 9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component `inform%factor_status`.
- 10. The factorization failed; the return status from the factorization package is given in the component `inform%factor_status`.
- 11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component `inform%factor_status`.
- 16. The problem is so ill-conditioned that further progress is impossible.
- 17. The step is too small to make further impact.
- 19. The elapsed CPU or system clock time limit has been reached. This may happen if either `control%cpu_time_limit` or `control%clock_time_limit` is too small, but may also be symptomatic of a badly scaled problem.
- 82. The user has forced termination of `GALAHAD_DGO_solve` by removing the file named `control%alive_file` from unit `control%alive_unit`.
- 90. The Hessian storage type in `nlp%H_type` is not one of 'DENSE', 'COORDINATE', 'SPARSE\_BY\_ROWS' or 'DIAGONAL'.
- 91. The hash table used to store the dictionary of vertices of the sub-boxes is full, and there is no room to increase it further.
- 99. The budget limit on function evaluations has been reached. This will happen if the limit `max_evals` is exceeded, and is quite normal for stochastic global-optimization methods. The user may explore increasing `max_evals` to see if that produces a lower value of the objective function, but there are unfortunately no guarantees.

## 2.8 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `DGO_control_type` (see Section 2.3.3), by reading an appropriate data specification file using the subroutine `DGO_read_specfile`. This facility is useful as it allows a user to change DGO control parameters without editing and recompiling programs that call DGO.

A specification file, or *specfile*, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by `DGO_read_specfile` must start with a "BEGIN DGO" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by DGO_read_specfile .. )
  BEGIN DGO
    keyword    value
    .....
    keyword    value
  END
( .. lines ignored by DGO_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN DGO" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
 See <https://www.galahad.rl.ac.uk/download/> for full details.

```
BEGIN DOGO SPECIFICATION
```

and

```
END DOGO SPECIFICATION
```

are acceptable. Furthermore, between the “BEGIN DOGO” and “END” delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or \* are ignored. The content of a line after a ! or \* character is also ignored (as is the ! or \* character itself). This provides an easy manner to “comment out” some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are “ON”, “TRUE”, “.TRUE.”, “T”, “YES”, “Y”, or “OFF”, “NO”, “N”, “FALSE”, “.FALSE.” and “F”. Empty values are also allowed for logical control parameters, and are interpreted as “TRUE”.

The specification file must be open for input when `DGO_read_specfile` is called, and the associated device number passed to the routine in `device` (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by `DGO_read_specfile`.

### 2.8.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL DGO_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `DGO_control_type` (see Section 2.3.3). Default values should have already been set, perhaps by calling `DGO_initialize`. On exit, individual components of `control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.3.3) of `control` that each affects are given in Table 2.1 on the facing page.

`device` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set to the unit number on which the specfile has been opened. If `device` is not open, `control` will not be altered and execution will continue, but an error message will be printed on unit `control%error`.

## 2.9 Information printed

If `control%print_level` is positive, information about the progress of the algorithm will be printed on unit `control%out`. If `control%print_level = 1`, a single line of output will be produced every time the objective function improves. This will include the number of attempts to improve the objective so far, the values of the objective function and the norm of its gradient, and the number of function and gradient evaluations.

If `control%print_level ≥ 2` this output will be increased to provide significant detail of each iteration. Further details concerning the attempted solution of the models may be obtained by increasing `control%TRB_control%print_level`, and `control%UGO_control%print_level`.

## 3 GENERAL INFORMATION

**Use of common:** None.

**Workspace:** Provided automatically by the module.

**Other routines called directly:** None.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

| command                            | component of control        | value type |
|------------------------------------|-----------------------------|------------|
| error-printout-device              | %error                      | integer    |
| printout-device                    | %out                        | integer    |
| print-level                        | %print_level                | integer    |
| start-print                        | %start_print                | integer    |
| stop-print                         | %stop_print                 | integer    |
| iterations-between-printing        | %print_gap                  | integer    |
| maximum-number-of-iterations       | %maxit                      | integer    |
| maximum-number-of-evaluations      | tt %max_evals               | integer    |
| initial-dictionary-size            | tt %dictionary_size         | integer    |
| alive-device                       | %alive_unit                 | integer    |
| infinity-value                     | %infinity                   | real       |
| lipschitz-lower-bound              | %lipschitz_lower_bound      | real       |
| lipschitz-reliability-parameter    | %lipschitz_reliability      | real       |
| lipschitz-control-parameter        | %lipschitz_control          | real       |
| maximum-box-length-required        | %stop_length                | real       |
| maximum-objective-gap-required     | %stop_f                     | real       |
| minimum-objective-before-unbounded | %obj_unbounded              | real       |
| maximum-cpu-time-limit             | %cpu_time_limit             | real       |
| maximum-clock-time-limit           | %clock_time_limit           | real       |
| hessian-available                  | %hessian_available          | logical    |
| prune-boxes                        | %prune                      | logical    |
| perform-local-optimization         | %perform_local_optimization | logical    |
| space-critical                     | %space_critical             | logical    |
| deallocate-error-fatal             | %deallocate_error_fatal     | logical    |
| alive-filename                     | %alive_file                 | character  |

Table 2.1: Specfile commands and associated components of control.

**Other modules used directly:** DGO\_solve calls the GALAHAD packages GALAHAD\_CLOCK, GALAHAD\_SYMBOLS, GALAHAD\_HASH, GALAHAD\_NLPT, GALAHAD\_USERDATA, GALAHAD\_SPECFILE, GALAHAD\_SPACE, GALAHAD\_NORMS, GALAHAD\_UGO and GALAHAD\_TRB.

**Input/output:** Output is under control of the arguments control%error, control%out and control%print\_level.

**Restrictions:**  $nlp\%n > 0$  and  $nlp\%H\_type \in \{ 'DENSE', 'COORDINATE', 'SPARSE\_BY\_ROWS', 'DIAGONAL' \}$ .

**Portability:** ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

## 4 METHOD

Starting with the initial box  $\mathbf{x}' \leq \mathbf{x} \leq \mathbf{x}''$ , a sequence of boxes is generated by considering the current set, and partitioning a promising candidate into three equally-sized sub-boxes by splitting along one of the box dimensions. Each partition requires only a pair of new function and derivative evaluations, and these values, together with estimates of Lipschitz constants, makes it possible to remove other boxes from further consideration as soon as they cannot contain a global minimizer. Efficient control of the dictionary of vertices of the sub-boxes is handled using a suitable hashing procedure provided by GALAHAD\_HASH; each sub-box is indexed by the concatenated coordinates of a pair of opposite vertices. At various stages, local minimization in a promising sub-box, using GALAHAD\_TRB, may be used to improve the best-known upper bound on the global minimizer. If  $n = 1$ , the specialised univariate global minimization package GALAHAD\_UGO is called directly.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
See <https://www.galahad.rl.ac.uk/download/> for full details.

We reiterate that although there are theoretical guarantees, these may require a large number of evaluations as the dimension and nonconvexity increase. Thus the method should best be viewed as a heuristic to try to find a reasonable approximation of the global minimum.

## References:

The global minimization method employed is an extension of that due to

Ya. D. Sergeyev and D. E. Kasov (2015), “A deterministic global optimization using smooth diagonal auxiliary functions”, Communications in Nonlinear Science and Numerical Simulation, Vol 21, Nos 1-3, pp. 99-111.

but adapted to use 2nd derivatives, while in the special case when  $n = 1$ , a simplification based on the ideas in

D. Lera and Ya. D. Sergeyev (2013), “Acceleration of univariate global optimization algorithms working with Lipschitz functions and Lipschitz first derivatives” SIAM J. Optimization Vol. 23, No. 1, pp. 508–529

is used instead. The generic bound-constrained trust-region method used for local minimization is described in detail in

A. R. Conn, N. I. M. Gould and Ph. L. Toint (2000), Trust-region methods. SIAM/MPS Series on Optimization.

## 5 EXAMPLES OF USE

Suppose we wish to minimize the parametric objective function  $f(\mathbf{x}) = (4 + p * x_1^2 + \frac{1}{3}x_1^4)x_1^2 + x_1x_2 + (4x_2^2 - 4) * x_2^2$  when the parameter  $p$  takes the values -2.1, and the components of  $\mathbf{x}$  are required to satisfy the bounds  $-3 \leq x_1 \leq 3$  and  $-2 \leq x_2 \leq 2$ . We may use the following code:

```
PROGRAM GALAHAD_DGO_EXAMPLE ! GALAHAD 4.0 - 2022-03-07 AT 13:50 GMT
USE GALAHAD_DGO_double      ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( DGO_control_type ) :: control
TYPE ( DGO_inform_type ) :: inform
TYPE ( DGO_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
EXTERNAL :: FUN, GRAD, HESS, HPROD
INTEGER :: s
INTEGER, PARAMETER :: n = 2, h_ne = 3
REAL ( KIND = wp ), PARAMETER :: p = - 2.1_wp
! start problem data
nlp%pname = 'CAMEL6' ! name
nlp%n = n; nlp%H%ne = h_ne ! dimensions
ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
nlp%X_l( : n ) = (/ - 3.0_wp, - 2.0_wp /)
nlp%X_u( : n ) = (/ 3.0_wp, 2.0_wp /)
! sparse co-ordinate storage format
CALL SMT_put( nlp%H%type, 'COORDINATE', s ) ! Specify co-ordinate storage
ALLOCATE( nlp%H%val( h_ne ), nlp%H%row( h_ne ), nlp%H%col( h_ne ) )
nlp%H%row = (/ 1, 2, 2 /) ! Hessian H
nlp%H%col = (/ 1, 1, 2 /) ! NB lower triangle
! problem data complete
ALLOCATE( userdata%real( 1 ) ) ! Allocate space for parameter
userdata%real( 1 ) = p ! Record parameter, p
CALL DGO_initialize( data, control, inform ) ! Initialize control parameters
control%maxit = 2000
```

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

```

! Solve the problem
inform%status = 1                                ! set for initial entry
CALL DGO_solve( nlp, control, inform, data, userdata, eval_F = FUN,      &
                eval_G = GRAD, eval_H = HESS, eval_HPROD = HPROD )
IF ( inform%status == 0 ) THEN                    ! Successful return
    WRITE( 6, "( ' DGO: ', I0, ' evaluations -', /,                      &
    &      ' Best objective value found =', ES12.4, /,                    &
    &      ' Corresponding solution = ', ( 5ES12.4 ) )" )                &
    inform%iter, inform%obj, nlp%X
ELSE                                              ! Error returns
    WRITE( 6, "( ' DGO_solve exit status = ', I6 ) " ) inform%status
END IF
CALL DGO_terminate( data, control, inform ) ! delete internal workspace
DEALLOCATE( nlp%X, nlp%G, nlp%H%val, nlp%H%row, nlp%H%col, userdata%real )
END PROGRAM GALAHAD_DGO_EXAMPLE

SUBROUTINE FUN( status, X, userdata, f )          ! Objective function
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), INTENT( OUT ) :: f
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: x1, x2, p
x1 = X( 1 ) ; x2 = X( 2 ) ; p = userdata%real( 1 )
f = ( 4.0_wp + p * x1 ** 2 + x1 ** 4 / 3.0_wp ) * x1 ** 2 + x1 * x2 +    &
    ( - 4.0_wp + 4.0_wp * x2 ** 2 ) * x2 ** 2
status = 0
RETURN
END SUBROUTINE FUN

SUBROUTINE GRAD( status, X, userdata, G )        ! gradient of the objective
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: G
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: x1, x2, p
x1 = X( 1 ) ; x2 = X( 2 ) ; p = userdata%real( 1 )
G( 1 ) = ( 8.0_wp + 4.0_wp * p * x1 ** 2 + 2.0_wp * x1 ** 4 ) * x1 + x2
G( 2 ) = x1 + ( - 8.0_wp + 16.0_wp * x2 ** 2 ) * x2
status = 0
RETURN
END SUBROUTINE GRAD

SUBROUTINE HESS( status, X, userdata, Hval ) ! Hessian of the objective
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: Hval
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: x1, x2, p
x1 = X( 1 ) ; x2 = X( 2 ) ; p = userdata%real( 1 )

```

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

```

Hval( 1 ) = 8.0_wp + 12.0_wp * p * x1 ** 2 + 10.0_wp * x1 ** 4
Hval( 2 ) = 1.0_wp
Hval( 3 ) = - 8.0_wp + 48.0_wp * x2 * x2
status = 0
RETURN
END SUBROUTINE HESS

SUBROUTINE HPROD( status, X, userdata, U, V, got_h ) ! Hessian-vector product
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( INOUT ) :: U
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
LOGICAL, OPTIONAL, INTENT( IN ) :: got_h
REAL ( KIND = wp ) :: x1, x2, p
x1 = X( 1 ) ; x2 = X( 2 ) ; p = userdata%real( 1 )
U( 1 ) = U( 1 ) + ( 8.0_wp + 12.0_wp * p * x1 ** 2 + 10.0_wp * x1 ** 4 )      &
              * V( 1 ) + V( 2 )
U( 2 ) = U( 2 ) + V( 1 ) + ( - 8.0_wp + 48.0_wp * x2 * x2 ) * V( 2 )
status = 0
RETURN
END SUBROUTINE HPROD

```

Notice how the parameter  $p$  is passed to the function evaluation routines via the real component of the derived type `USERDATA_type`. The code produces the following output:

```

DGO: 201 evaluations -
Best objective value found = -1.0316E+00
Corresponding solution =   8.9842E-02 -7.1266E-01

```

If the user prefers to provide function, gradient and Hessian information without calls to specified routines, the following code is appropriate.

```

PROGRAM GALAHAD_DGO_EXAMPLE2 ! GALAHAD 4.0 - 2022-03-12 AT 11:10 GMT
USE GALAHAD_DGO_double           ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( DGO_control_type ) :: control
TYPE ( DGO_inform_type ) :: inform
TYPE ( DGO_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
INTEGER :: s
INTEGER, PARAMETER :: n = 2, h_ne = 3
REAL ( KIND = wp ) :: x1, x2
REAL ( KIND = wp ), PARAMETER :: p = - 2.1_wp
! start problem data
nlp%pname = 'CAMEL6' ! name
nlp%n = n ; nlp%h_ne = h_ne ! dimensions
ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
nlp%X_l( : n ) = ( / - 3.0_wp, - 2.0_wp / )
nlp%X_u( : n ) = ( / 3.0_wp, 2.0_wp / )
! sparse co-ordinate storage format
CALL SMT_put( nlp%H%type, 'COORDINATE', s ) ! Specify co-ordinate storage
ALLOCATE( nlp%H%val( h_ne ), nlp%H%row( h_ne ), nlp%H%col( h_ne ) )

```

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

```

    nlp%H%row = (/ 1, 2, 2 /) ! Hessian H
    nlp%H%col = (/ 1, 1, 2 /) ! NB lower triangle
! problem data complete
    CALL DGO_initialize( data, control, inform ) ! Initialize control parameters
! Solve the problem
    inform%status = 1 ! set for initial entry
    DO ! Solve problem using reverse communication
        CALL DGO_solve( nlp, control, inform, data, userdata )
        IF ( inform%status == 0 ) THEN ! Successful return
            WRITE( 6, "( ' DGO: ', I0, ' evaluations -', /,
&          ' Best objective value found =', ES12.4, /,
&          ' Corresponding solution = ', ( 5ES12.4 ) )" )
&          inform%iter, inform%obj, nlp%X
            EXIT
        ELSE IF ( inform%status < 0 ) THEN ! Error returns
            WRITE( 6, "( ' DGO_solve exit status = ', I6 ) " ) inform%status
            EXIT
        END IF
        x1 = nlp%X( 1 ) ; x2 = nlp%X( 2 )
        IF ( inform%status == 2 .OR. inform%status == 23 .OR.
&          inform%status == 25 .OR. inform%status == 235 ) THEN ! evaluate f
&          nlp%f = ( 4.0_wp + p * x1 ** 2 + x1 ** 4 / 3.0_wp ) * x1 ** 2
&          + x1 * x2 + ( - 4.0_wp + 4.0_wp * x2 ** 2 ) * x2 ** 2
        END IF
        IF ( inform%status == 3 .OR. inform%status == 23 .OR.
&          inform%status == 35 .OR. inform%status == 235 ) THEN ! evaluate g
&          nlp%G( 1 ) = ( 8.0_wp + 4.0_wp * p * x1 ** 2 + 2.0_wp * x1 ** 4 ) * x1
&          + x2
&          nlp%G( 2 ) = x1 + ( - 8.0_wp + 16.0_wp * x2 ** 2 ) * x2
        END IF
        IF ( inform%status == 4 ) THEN ! evaluate H
&          nlp%H%val( 1 ) = 8.0_wp + 12.0_wp * p * x1 ** 2 + 10.0_wp * x1 ** 4
&          nlp%H%val( 2 ) = 1.0_wp
&          nlp%H%val( 3 ) = - 8.0_wp + 48.0_wp * x2 * x2
        END IF
        IF ( inform%status == 5 .OR. inform%status == 25 .OR.
&          inform%status == 35 .OR. inform%status == 235 ) THEN ! evaluate u = Hv
&          data%U( 1 ) = data%U( 1 ) + ( 8.0_wp + 12.0_wp * p * x1 ** 2
&          + 10.0_wp * x1 ** 4 ) * data%V( 1 ) + data%V( 2 )
&          data%U( 2 ) = data%U( 2 ) + data%V( 1 )
&          + ( - 8.0_wp + 48.0_wp * x2 * x2 ) * data%V( 2 )
        END IF
        data%eval_status = 0
    END DO
    CALL DGO_terminate( data, control, inform ) ! delete internal workspace
    DEALLOCATE( nlp%X, nlp%G, nlp%H%val, nlp%H%row, nlp%H%col )
END PROGRAM GALAHAD_DGO_EXAMPLE2

```

This produces the same output.