



GALAHAD

NLPT

USER DOCUMENTATION

GALAHAD Optimization Library version 5.5

1 SUMMARY

This package defines a derived type capable of supporting the storage of a variety of smooth **nonlinear programming problems** of the form

$$\min \mathbf{f}(\mathbf{x})$$

subject to the general constraints

$$\mathbf{c}^l \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}^u,$$

simple bounds on the variables

$$\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u,$$

and possibly regular simplex constraints

$$\mathbf{e}_{C_i}^T \mathbf{x}_{C_i} = 1 \text{ and } \mathbf{x}_{C_i} \geq 0 \text{ for } i = 1, \dots, m_c, \quad (1.1)$$

where $\mathbf{f}$ is a smooth “objective function”, where $\mathbf{c}(\mathbf{x})$ is a smooth function from $\mathbb{R}^n$ into $\mathbb{R}^m$, the inequalities are understood componentwise, and the non-overlapping integer sets of cohorts $C_i \subseteq \{1, \dots, n\}$, for which $C_i \cap C_j = \emptyset$ for $1 \leq i < j \leq m_c$, are given. The vectors $\mathbf{c}^l \leq \mathbf{c}^u$ and $\mathbf{x}^l \leq \mathbf{x}^u$ are m - and n -dimensional, respectively, and may contain components equal to minus or plus infinity. An important function associated with the problem is its Lagrangian

$$\mathbf{L}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{f}(\mathbf{x}) - \mathbf{y}^T \mathbf{c}(\mathbf{x}) - \mathbf{z}^T \mathbf{x}$$

where $\mathbf{y}$ belongs to $\mathbb{R}^m$ and $\mathbf{z}$ belongs to $\mathbb{R}^n$. The solution of such problem may require the storage of the objective function’s gradient

$$\mathbf{g}(\mathbf{x}) = \nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x}),$$

the $n \times n$ symmetric objective function’s Hessian

$$\mathbf{H}_f(\mathbf{x}) = \nabla_{\mathbf{xx}} \mathbf{f}(\mathbf{x})$$

the $m \times n$ constraints’ Jacobian whose i -th row is the gradient of the i -th constraint:

$$\mathbf{e}_i^T \mathbf{J}(\mathbf{x}) = [\nabla_{\mathbf{x}} \mathbf{c}_i(\mathbf{x})]^T,$$

the gradient of the Lagrangian with respect to $\mathbf{x}$,

$$\mathbf{g}_L(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \nabla_{\mathbf{x}} \mathbf{L}(\mathbf{x}, \mathbf{y}, \mathbf{z})$$

and of the Lagrangian’s Hessian with respect to $\mathbf{x}$

$$\mathbf{H}_L(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \nabla_{\mathbf{xx}} \mathbf{L}(\mathbf{x}, \mathbf{y}, \mathbf{z}).$$

Note that this last matrix is equal to the Hessian of the objective function when the problem is unconstrained ($m = 0$), which allows us to use the same symbol $\mathbf{H}$ for both cases. The package also supports weighted least-squares objectives

$$\mathbf{f}(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^o w_i r_i^2(\mathbf{x}) \equiv \frac{1}{2} \|\mathbf{r}(\mathbf{x})\|_W^2,$$

and their associated m_r by n Jacobian $\mathbf{J}_r(\mathbf{x}) \equiv \nabla_{\mathbf{x}} \mathbf{r}(\mathbf{x})$, if necessary.

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

Full advantage can be taken of any zero coefficients in the matrices $\mathbf{H}$ or $\mathbf{J}$.

The module also contains subroutines that are designed for printing parts of the problem data, and for matrix storage scheme conversions.

ATTRIBUTES — Versions: GALAHAD_NLPT_single, GALAHAD_NLPT_double, **Calls:** GALAHAD_TOOLS, GALAHAD_SMT. **Date:** May 2003. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory, and Ph. L. Toint, University of Namur, Belgium. **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_NLPT_single
```

with the obvious substitution `GALAHAD_NLPT_double`, `GALAHAD_NLPT_quadruple`, `GALAHAD_NLPT_single_64`, `GALAHAD_NLPT_double_64` and `GALAHAD_NLPT_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived type `NLPT_problem_type`, (Section 2.4) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

Both the Hessian matrix $\mathbf{H}$ and the Jacobians $\mathbf{J}$ and $\mathbf{J}_r$ may each be stored in one of three input formats (the format for the matrices being possibly different).

2.1.1 Dense storage format

The matrix $\mathbf{J}$ (or $\mathbf{J}_r$) 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. Component $n * (i - 1) + j$ of the storage array `J_val` will hold the value J_{ij} for $i = 1, \dots, m$, $j = 1, \dots, n$. 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 will 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 $1 \leq j \leq i \leq n$.

If this storage scheme is used, `J_type` and/or `H_type` must be set the value of the symbol `GALAHAD_DENSE`.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry of $\mathbf{J}$, its row index i , column index j and value J_{ij} are stored in the l -th components of the integer arrays `J_row`, `J_col` and real array `J_val`. The order is unimportant, but the total number of entries `J_ne` is also required. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H_row`, `H_col`, a real array `H_val` and an integer value `H_ne`), except that only the entries in the lower triangle need be stored.

If this storage scheme is used, `J_type` and/or `H_type` must be set the value of the symbol `GALAHAD_COORDINATE`.

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 $\mathbf{J}$, the i -th component of a integer array `J_ptr` holds the position of the first entry in this row, while `J_ptr(m + 1)` holds the total number of entries plus one. The column indices j and values J_{ij} of the entries in the i -th row are stored in components $l = J_ptr(i), \dots, J_ptr(i + 1) - 1$ of the integer array `J_col`, and real array `J_val`, respectively. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H_ptr`, `H_col`, and a real

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

array `H_val`), except that only the entries in the lower triangle need be stored. The values of `J_ne` and `H_ne` are not mandatory, since they can be recovered from

$$J_ne = J_ptr(n+1) - 1 \quad \text{and} \quad H_ne = H_ptr(n+1) - 1$$

For sparse matrices, this scheme almost always requires less storage than its predecessor.

If this storage scheme is used, `J_type` and/or `H_type` must be set the value of the symbol `GALAHAD.SPARSE_BY_ROWS`.

2.2 Optimality conditions

The solution $\mathbf{x}$ necessarily satisfies the primal first-order optimality conditions

$$\mathbf{c}^l \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}^u, \quad \text{and} \quad \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u,$$

the dual first-order optimality conditions

$$\mathbf{g}(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T \mathbf{y} + \mathbf{z}$$

where

$$\mathbf{y} = \mathbf{y}^l + \mathbf{y}^u, \quad \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u \quad \mathbf{y}^l \geq 0, \quad \mathbf{y}^u \leq 0, \quad \mathbf{z}^l \geq 0 \quad \text{and} \quad \mathbf{z}^u \leq 0,$$

and the complementary slackness conditions

$$(\mathbf{c}(\mathbf{x}) - \mathbf{c}^l)^T \mathbf{y}^l = 0, \quad (\mathbf{c}(\mathbf{x}) - \mathbf{c}^u)^T \mathbf{y}^u = 0, \quad (\mathbf{x} - \mathbf{x}^l)^T \mathbf{z}^l = 0 \quad \text{and} \quad (\mathbf{x} - \mathbf{x}^u)^T \mathbf{z}^u = 0,$$

where the vectors $\mathbf{y}$ and $\mathbf{z}$ are known as the Lagrange multipliers for the general constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold componentwise. The dual first-order optimality condition is equivalent to the condition that $\mathbf{g}_L(\mathbf{x}, \mathbf{y}, \mathbf{z}) = 0$.

2.3 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.

2.4 The derived data type

Two derived data types, `SMT_TYPE` and `NLPT_problem_type`, are accessible from the package. It is intended that, for any particular application, only those components which are needed will be set.

2.4.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the matrices $\mathbf{A}$ and $\mathbf{H}$. The components of `SMT_TYPE` used here are:

`type` is an allocatable array of rank one and type default `CHARACTER`, that holds a string which indicates the storage scheme used.

`m` is a scalar component of type `INTEGER(ip_)`, that holds the number of rows in the matrix.

`n` is a scalar component of type `INTEGER(ip_)`, that holds the number of columns in the matrix.

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

- `ne` is a scalar variable of type `INTEGER(ip_)`, that may hold the number of matrix entries (see §2.1.2).
- `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 a *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 `m + 1`, that may hold the pointers to the first entry in each row (see §2.1.3).

2.4.2 The derived data type for holding optimization problems

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

- `pname` is a scalar variable of type `default CHARACTER(LEN = 10)`, that holds the problem's name.
- `n` is a scalar variable of type `INTEGER(ip_)`, that holds the number of optimization variables, n .
- `vnames` is a rank-one allocatable array of dimension `n` and type `CHARACTER(LEN = 10)` that holds the names of the problem's variables. The j -th component of `vnames`, $j = 1, \dots, n$, contains the name of x_j .
- `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 .
- `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 x_j^l . Infinite bounds are allowed by setting the corresponding components of `x_l` to any value smaller than `-infinity`.
- `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 x_j^u . Infinite bounds are allowed by setting the corresponding components of `x_u` to any value larger than `infinity`.
- `z` is a rank-one allocatable array of dimension `n` and type `default REAL(rp_)`, that holds the values $\mathbf{z}$ of estimates of the dual variables corresponding to the simple bound constraints (see Section 2.2). The j -th component of `z`, $j = 1, \dots, n$, contains z_j .
- `x_status` is a rank-one allocatable array of dimension `n` and type `INTEGER(ip_)`, that holds the status of the problem's variables corresponding to the presence of their bounds. The j -th component of `x_status`, $j = 1, \dots, n$, contains the status of x_j . Typical values are `GALAHAD_FREE`, `GALAHAD_LOWER`, `GALAHAD_UPPER`, `GALAHAD_RANGE`, `GALAHAD_FIXED`, `GALAHAD_STRUCTURAL`, `GALAHAD_ELIMINATED`, `GALAHAD_ACTIVE`, `GALAHAD_INACTIVE` or `GALAHAD_UNDEFINED`.
- `f` is a scalar variable of type `REAL(rp_)`, that holds the current value of the objective function.
- `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 g_j .
- `H_type` is a scalar variable of type `INTEGER(ip_)`, that specifies the type of storage used for the lower triangle of the objective function's or Lagrangian's Hessian $\mathbf{H}$. Possible values are `GALAHAD_DENSE`, `GALAHAD_COORDINATE` or `GALAHAD_SPARSE_BY_ROWS`.

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

- `H_ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of non-zero entries in the lower triangle of the objective function's or Lagrangian's Hessian $\mathbf{H}$.
- `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 either of the other two 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 storage scheme is 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.
- `m` is a scalar variable of type `INTEGER(ip_)`, that holds the number of general constraints, m .
- `m_a` is a scalar variable of type `INTEGER(ip_)`, that holds the number of general linear constraints if the user wishes to distinguish these from m .
- `c` is a rank-one allocatable array of dimension `m` and type default `REAL(rp_)`, that holds the values $\mathbf{c}(\mathbf{x})$ of the constraints. The i -th component of `c`, $i = 1, \dots, m$, contains $\mathbf{c}_i(\mathbf{x})$.
- `c_l` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the vector of lower bounds $\mathbf{c}^l$ on the general constraints. The i -th component of `c_l`, $i = 1, \dots, m$, contains $\mathbf{c}_i^l$. Infinite bounds are allowed by setting the corresponding components of `c_l` to any value smaller than `-infinity`.
- `c_u` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the vector of upper bounds $\mathbf{c}^u$ on the general constraints. The i -th component of `c_u`, $i = 1, \dots, m$, contains $\mathbf{c}_i^u$. Infinite bounds are allowed by setting the corresponding components of `c_u` to any value larger than `infinity`.
- `equation` is a rank-one allocatable array of dimension `m` and type default `LOGICAL`, that specifies if each constraint is an equality or an inequality. The i -th component of `equation` is `.TRUE.` iff the i -th constraint is an equality, i.e. iff $\mathbf{c}_i^l = \mathbf{c}_i^u$.
- `linear` is a rank-one allocatable array of dimension `m` and type default `LOGICAL`, that specifies if each constraint is linear. The i -th component of `linear` is `.TRUE.` iff the i -th constraint is linear.
- `y` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the values $\mathbf{y}$ of estimates of the Lagrange multipliers corresponding to the general constraints (see Section 2.2). The i -th component of `y`, $i = 1, \dots, m$, contains $\mathbf{y}_i$.
- `c_status` is a rank-one allocatable array of dimension `m` and type `INTEGER(ip_)`, that holds the status of the problem's constraints corresponding to the presence of their bounds. The i -th component of `c_status`, $j = 1, \dots, m$, contains the status of $\mathbf{c}_i$. Typical values are `GALAHAD_FREE`, `GALAHAD_LOWER`, `GALAHAD_UPPER`, `GALAHAD_RANGE`, `GALAHAD_FIXED`, `GALAHAD_STRUCTURAL`, `GALAHAD_ELIMINATED`, `GALAHAD_ACTIVE`, `GALAHAD_INACTIVE` or `GALAHAD_UNDEFINED`.
- `J_type` is a scalar variable of type `INTEGER(ip_)`, that specifies the type of storage used for the constraints' Jacobian $\mathbf{J}$. Possible values are `GALAHAD_DENSE`, `GALAHAD_COORDINATE` or `GALAHAD_SPARSE_BY_ROWS`.
- `J_ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of non-zero entries in the constraints' Jacobian $\mathbf{J}$.

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

- `J_val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the Jacobian matrix $\mathbf{J}$ in any of the storage schemes discussed in Section 2.1.
- `J_row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of $\mathbf{J}$ in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other two schemes.
- `J_col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of $\mathbf{J}$ 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 storage scheme is used.
- `J_ptr` is a rank-one allocatable array of dimension $m+1$ and type `INTEGER(ip_)`, that holds the starting position of each row of $\mathbf{J}$, 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.
- `gL` is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that holds the gradient $\mathbf{g}_L$ of the problem's Lagrangian $\mathbf{L}$ with respect to $\mathbf{x}$. The j -th component of `gL`, $j = 1, \dots, n$, contains $[\mathbf{g}_L]_j$.
- `m_r` is a scalar variable of type `INTEGER(ip_)`, that holds the number of residuals, m_r , in a least-squares objective.
- `r` is a rank-one allocatable array of dimension o and type default `REAL(rp_)`, that holds the values $\mathbf{r}(\mathbf{x})$ of the residuals in a least-squares objective. The i -th component of `r`, $i = 1, \dots, o$, contains $\mathbf{r}_i(\mathbf{x})$.
- `w` is a rank-one allocatable array of dimension o and type default `REAL(rp_)`, that holds the values $\mathbf{w}$ of the weights for a least-squares objective. The i -th component of `w`, $i = 1, \dots, o$, contains $\mathbf{w}_i$.
- `Jr_type` is a scalar variable of type `INTEGER(ip_)`, that specifies the type of storage used for the least-squares Jacobian $\mathbf{J}_r$. Possible values are `GALAHAD_DENSE`, `GALAHAD_COORDINATE` or `GALAHAD_SPARSE_BY_ROWS`.
- `Jr_ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of non-zero entries in the least-squares Jacobian $\mathbf{J}_r$.
- `Jr_val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the least-squares Jacobian matrix $\mathbf{J}_r$ in any of the storage schemes discussed in Section 2.1.
- `Jr_row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of $\mathbf{J}_r$ in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other two schemes.
- `Jr_col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of $\mathbf{J}_r$ 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 storage scheme is used.
- `Jr_ptr` is a rank-one allocatable array of dimension $o+1$ and type `INTEGER(ip_)`, that holds the starting position of each row of $\mathbf{J}_r$, 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.
- `m_c` is a scalar variable of type `INTEGER(ip_)`, that holds the number of simplex cohorts, m_c .
- `cohort` is a rank-one allocatable array of dimension n and type `INTEGER(ip_)`, that lists the simplex cohort for each variable if there are one or more non-overlapping simplex constraints of the form (1.1). In this case, if variable x_j is associated with cohort C_i , $1 \leq i \leq m$, `COHORT(j)` should be set to i , while if x_j is unconstrained `COHORT(j) = 0` should be assigned. At least one value `COHORT(j)` for $j = 1, \dots, n$ is expected to take the value i for every $1 \leq i \leq m$, that is no empty cohorts are allowed. If all the variables are contained in a single cohort, or if the constraint set is not (1.1), `COHORT` need not be allocated.
- `H`, `J`, `Jr` are scalar variable of type `SMT_TYPE` that provide compact alternatives to the arrays `H_*`, `J_*` and `Jr_*`, described above.

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 holds the value of infinity, that is the value beyond which problem bounds are judged to be infinite.

Note that not every component of this data type is used by every package.

2.5 Argument lists and calling sequences

There are seven procedures for user calls:

1. The subroutine `NLPT_write_stats` is used to write general information on the problem such as the number of variables and constraints of different types.
2. The subroutine `NLPT_write_variables` is used to write the current values of the problem's variables, bounds and of their associated duals.
3. The subroutine `NLPT_write_constraints` is used to write the current values of the problem's constraints, bounds and of their associated multipliers.
4. The subroutine `NLPT_write_problem` is used to write the problem's number of variables and constraints per type, as well as current values of the problem's variables and constraints. This broadly corresponds to successively calling the three subroutines mentioned above. The subroutine additionally (optionally) writes the values of the Lagrangian's Hessian $\mathbf{H}$ and constraints Jacobian $\mathbf{J}$.
5. The subroutine `NLPT_J_from_C_to_Srow` builds the permutation that transforms the Jacobian from coordinate storage to sparse-by-row storage, as well as the `J_ptr` and `J_col` vectors.
6. The subroutine `NLPT_J_from_C_to_Scol` builds the permutation that transforms the Jacobian from coordinate storage to sparse-by-column storage, as well as the `J_ptr` and `J_row` vectors.
7. The subroutine `NLPT_cleanup` is used to deallocate the memory space used by a problem data structure.

2.5.1 Writing the problem's statistics

The number of variables and constraints for each type of bounds (free, lower/upper bounded, range bounded, linear, equalities/fixed) is output by using the call

```
CALL NLPT_write_stats( problem, out )
```

where

`problem` is a scalar `INTENT(IN)` argument of type `NLPT_problem_type`, that holds the problem for which statistics must be written.

`out` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that holds the device number on which problem statistics should be written.

Note that `problem%pname` is assumed to be defined and that both `problem%c_l` and `problem%c_u` are assumed to be associated whenever `problem%m > 0`.

2.5.2 Writing the problem's variables, bounds and duals

The values of the variables and associated bounds and duals is output by using the call

```
CALL NLPT_write_variables( problem, out )
```

where

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

`problem` is a scalar `INTENT(IN)` argument of type `NLPT_problem_type`, that holds the problem for which variables values, bounds and duals must be written.

`out` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that holds the device number on which problem variables values, bounds and duals should be written.

This routine assumes that `problem%pname` and `problem%x` are associated. The bounds are printed whenever `problem%x_l` and `problem%x_u` are associated. Moreover, it is also assumed in this case that `problem%g` is associated when `problem%m = 0`, and that `problem%z` is associated when `problem%m > 0`. The variables' names are used whenever `problem%vnames` is associated, but this is not mandatory.

2.5.3 Writing the problem's constraints, bounds and multipliers

The values of the constraints and associated bounds and multipliers is output by using the call

```
CALL NLPT_write_constraints( problem, out )
```

where

`problem` is a scalar `INTENT(IN)` argument of type `NLPT_problem_type`, that holds the problem for which constraints values, bounds and multipliers must be written.

`out` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that holds the device number on which problem constraints values, bounds and multipliers should be written.

This routine assumes that `problem%pname`, `problem%c`, `problem%c_l`, `problem%c_u` and `problem%y` are associated. The types of constraints are used whenever `problem%equation` and/or `problem%linear` are associated, but this is not mandatory. The constraints' names are used whenever `problem%cnames` is associated, but this is not mandatory.

2.5.4 Writing the entire problem

The most important data of a problem can be output by the call

```
CALL NLPT_write_problem( problem, out, print_level )
```

where

`problem` is a scalar `INTENT(IN)` argument of type `NLPT_problem_type`, that holds the problem whose data must be written.

`out` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that holds the device number on which the problem data should be written.

`print_level` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that holds the level of details required for output. Possible values are:

GALAHAD_SILENT: no output is produced;

GALAHAD_TRACE: the problem's statistics are output, plus the norms of the current vector of variables, the objective function's value and the norm of its gradient, and the maximal bound and constraint violations.

GALAHAD_ACTION: the problem's statistics are output, plus the values of the variables, bounds and associated duals, the value of the objective function, the value of the objective function's gradient, the values of the constraints and associated bounds and multipliers.

GALAHAD_DETAILS: as for GALAHAD_ACTION, plus the values of the Lagrangian's Hessian and of the constraints' Jacobian.

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

This routine assumes that `problem%pname` and `problem%x` are associated. The bounds on the variables are printed whenever `problem%x_l` and `problem%x_u` are associated. Moreover, it is also assumed in this case that `problem%g` is associated when `problem%m = 0`, and that `problem%z` is associated when `problem%m > 0`. The variables' names are used whenever `problem%vnames` is associated, but this is not mandatory. In the case where `problem%m > 0`, it is furthermore assumed that `problem%c` `problem%c_l`, `problem%c_u` and `problem%y` are associated. The types of constraints are used whenever `problem%equation` and/or `problem%linear` are associated, but this is not mandatory. The constraints' names are used whenever `problem%cnames` is associated, but this is not

2.5.5 Problem cleanup

The memory space allocated to allocatable in the problem data structure is deallocated by the call

```
CALL NLPT_cleanup( problem )
```

where

`problem` is a scalar `INTENT(IN)` argument of type `NLPT_problem_type`, that holds the problem whose memory space must be deallocated.

2.5.6 Transforming the Jacobian from co-ordinate storage to sparse-by-rows

The permutation that transforms the Jacobian from co-ordinate storage to sparse-by-rows, as well as the associated `ptr` and `col` vectors can be obtained by the call

```
CALL NLPT_J_perm_from_C_to_Srow( problem, perm, col, ptr )
```

where

`problem` is a scalar `INTENT(IN)` argument of type `NLPT_problem_type`, that holds the Jacobian matrix to transform. Note that we must have `problem%J_type = GALAHAD_COORDINATE`.

`perm` is an allocatable to a vector `INTENT(OUT)` of type `INTEGER(ip_)` and dimension equal to `problem%J_nnz`, that returns the permutation of the elements of `problem%J_val` that must be applied to transform the Jacobian from co-ordinate storage to sparse-by-rows.

`col` is an allocatable to a vector `INTENT(OUT)` of type `INTEGER(ip_)` and dimension `problem%J_ne` whose k -th component is the column index of the k -th element of `problem%J_val` after permutation by `perm`.

`ptr` is an allocatable to a vector `INTENT(OUT)` of type `INTEGER(ip_)` and dimension `problem%m + 1` whose i -th component is the index in `problem%J_val` (after permutation by `perm`) of the first entry of row i . Moreover,

$$\text{ptr}(\text{problem}\%m + 1) = \text{problem}\%J_ne + 1.$$

2.5.7 Transforming the Jacobian from co-ordinate storage to sparse-by-columns

The permutation that transforms the Jacobian from co-ordinate storage to sparse-by-columns, as well as the associated `ptr` and `row` vectors can be obtained by the call

```
CALL NLPT_J_perm_from_C_to_Scol( problem, perm, row, ptr )
```

where

`problem` is a scalar `INTENT(IN)` argument of type `NLPT_problem_type`, that holds the Jacobian matrix to transform. Note that we must have `problem%J_type = GALAHAD_COORDINATE`.

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

`perm` is an allocatable to a vector `INTENT(OUT)` of type `INTEGER(ip_)` and dimension equal to `problem%J_nnz`, that returns the permutation of the elements of `problem%J_val` that must be applied to transform the Jacobian from co-ordinate storage to sparse-by-columns.

`col` is an allocatable to a vector `INTENT(OUT)` of type `INTEGER(ip_)` and dimension `problem%J_ne` whose k -th component is the row index of the k -th element of `problem%J_val` after permutation by `perm`.

`ptr` is an allocatable to a vector `INTENT(OUT)` of type `INTEGER(ip_)` and dimension `problem%m + 1` whose i -th component is the index in `problem%J_val` (after permutation by `perm`) of the first entry of column i . Moreover,

$$\text{ptr}(\text{problem}\%m + 1) = \text{problem}\%J_ne + 1.$$

3 GENERAL INFORMATION

Other modules used directly: NLPT calls the `TOOLS` and `SMT GALAHAD` modules.

Input/output: Output is under the control of the `print_level` argument for the `NLPT_write_problem` subroutine.

Restrictions: `problem%n` > 0, `problem%m` ≥ 0. Additionally, the subroutines `NLPT_write_*` require that `problem%n` < 10¹⁴ and `problem%m` < 10¹⁴.

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

4 EXAMPLE OF USE

Suppose we wish to present the data for the problem of minimizing the objective function $(\mathbf{x}_1 - 2)\mathbf{x}_2$ subject to the constraints $\mathbf{x}_1^2 + \mathbf{x}_2^2 \leq 1$, $0 \leq -\mathbf{x}_1 + \mathbf{x}_2$, and the simple bound $0 \leq \mathbf{x}_1$, where the values are computed at the point $\mathbf{x}^T = (0, 1)$, which, together with the values $z_1 = 1$ and $\mathbf{y}^T = (-1, 0)$ defines a first-order critical point for the problem. Assume that we wish to store the Lagrangian's Hessian and the Jacobian in co-ordinate format. Assume also that we wish to write this data. We may accomplish these objectives by using the code:

```
PROGRAM GALAHAD_NLPT_EXAMPLE
  USE GALAHAD_NLPT_double      ! the problem type
  USE GALAHAD_SYMBOLS
  IMPLICIT NONE
  INTEGER, PARAMETER           :: wp = KIND( 1.0D+0 )
  INTEGER, PARAMETER           :: iout = 6      ! stdout and stderr
  REAL( KIND = wp ), PARAMETER :: INFINITY = (10.0_wp)**19
  TYPE( NLPT_problem_type )    :: problem
! Set the problem up.
  problem%pname = 'NLPT-TEST'
  problem%infinity = INFINITY
  problem%n = 2
  ALLOCATE( problem%vnames( problem%n ), problem%x( problem%n ) ,      &
            problem%x_l( problem%n ) , problem%x_u( problem%n ) ,      &
            problem%g( problem%n ) , problem%z( problem%n ) )
  problem%m = 2
  ALLOCATE( problem%equation( problem%m ), problem%linear( problem%m ), &
            problem%c( problem%m ) , problem%c_l( problem%m ) ,      &
            problem%c_u( problem%m ), problem%y( problem%m ) ,      &
            problem%cnames( problem%m ) )
  problem%J_ne = 4
  ALLOCATE( problem%J_val( problem%J_ne ), problem%J_row( problem%J_ne ), &
            problem%J_col( problem%J_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.

```

problem%H_ne      = 3
ALLOCATE( problem%H_val( problem%H_ne ), problem%H_row( problem%H_ne ),      &
          problem%H_col( problem%H_ne ) )
problem%H_type    = GALAHAD_COORDINATE
problem%J_type    = GALAHAD_COORDINATE
problem%vnames    = (/ 'X1' , 'X2'  /)
problem%x         = (/ 0.0D0 , 1.0D0 /)
problem%x_l       = (/ 0.0D0 , -INFINITY /)
problem%x_u       = (/ INFINITY, INFINITY /)
problem%cnames    = (/ 'C1' , 'C2'  /)
problem%c         = (/ 0.0D0 , 1.0D0 /)
problem%c_l       = (/ -INFINITY, 0.0D0 /)
problem%c_u       = (/ 1.0D0 , INFINITY /)
problem%y         = (/ -1.0D0 , 0.0D0 /)
problem%equation  = (/ .FALSE. , .FALSE. /)
problem%linear    = (/ .FALSE. , .TRUE. /)
problem%z         = (/ 1.0D0 , 0.0D0 /)
problem%f         = -2.0_wp
problem%g         = (/ 1.0D0 , -1.0D0 /)
problem%J_row     = (/ 1 , 1 , 2 , 2 /)
problem%J_col     = (/ 1 , 2 , 1 , 2 /)
problem%J_val     = (/ 0.0D0 , 2.0D0 , -1.0D0 , 1.0D0 /)
problem%H_row     = (/ 1 , 2 , 2 /)
problem%H_col     = (/ 1 , 1 , 2 /)
problem%H_val     = (/ 2.0D0 , 1.0D0 , 2.0D0 /)
NULLIFY( problem%x_status, problem%H_ptr, problem%J_ptr, problem%gL )
CALL NLPT_write_problem( problem, iout, GALAHAD_DETAILS )
! Cleanup the problem.
CALL NLPT_cleanup( problem )
STOP
END PROGRAM GALAHAD_NLPT_EXAMPLE

```

which gives the following output:

```

+-----+
|               Problem : NLPT-TEST               |
+-----+

```

	Free	Lower	Upper	Range	Fixed/	Linear	Total
		bounded	bounded	bounded	equalities		
Variables	1	1	0	0	0		2
Constraints		1	1	0	0	1	2

```

+-----+
|               Problem : NLPT-TEST               |
+-----+

```

j	Name	Lower	Value	Upper	Dual value
1	X1	0.0000E+00	0.0000E+00		1.0000E+00
2	X2		1.0000E+00		

OBJECTIVE FUNCTION value = -2.0000000E+00

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

GRADIENT of the objective function:

```
1  1.000000E+00 -1.000000E+00
```

Lower triangle of the HESSIAN of the Lagrangian:

i	j	value	i	j	value	i	j	value
1	1	2.0000E+00	2	1	1.0000E+00	2	2	2.0000E+00

```

+-----+
|               Problem : NLPT-TEST               |
+-----+
```

i	Name	Lower	Value	Upper	Dual value	
1	C1		0.0000E+00	1.0000E+00	-1.0000E+00	
2	C2	0.0000E+00	1.0000E+00		0.0000E+00	linear

JACOBIAN matrix:

i	j	value	i	j	value	i	j	value
1	1	0.0000E+00	1	2	2.0000E+00	2	1	-1.0000E+00
2	2	1.0000E+00						

```
----- END OF PROBLEM -----
```

We could choose to hold the lower triangle of H in sparse-by-rows format by replacing the lines

```

ALLOCATE( problem%H_val( problem%H_ne ), problem%H_row( problem%H_ne ),      &
          problem%H_col( problem%H_ne ) )
problem%H_type = GALAHAD_COORDINATE
```

and

```

problem%H_row = (/ 1 , 2 , 2 /)
problem%H_col = (/ 1 , 1 , 2 /)
problem%H_val = (/ 2.0D0 , 1.0D0 , 2.0D0 /)
NULLIFY( problem%x_status, problem%H_ptr, problem%J_ptr, problem%gL )
```

by

```

ALLOCATE( problem%H_val( problem%H_ne ), problem%H_col( problem%H_ne ),      &
          problem%H_ptr( problem%n + 1 ) )
problem%H_type = GALAHAD_SPARSE_BY_ROWS
```

and

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

```
problem%H_ptr    = (/      1      ,      2      ,      4      /)
problem%H_col    = (/      1      ,      1      ,      2      /)
problem%H_val    = (/  2.0D0  ,  1.0D0  ,  2.0D0  /)
NULLIFY( problem%x_status, problem%H_row, problem%J_ptr, problem%gL )
```

or using a dense storage format with the replacement lines

```
ALLOCATE( problem%H_val( ( ( problem%n + 1 ) * problem%n ) / 2 ) )
problem%H_type = GALAHAD_DENSE
```

and

```
problem%H_val    = (/  2.0D0  ,  1.0D0  ,  2.0D0  /)
NULLIFY( problem%x_status, problem%H_row, problem%H_col, problem%H_ptr,      &
        problem%J_ptr, problem%gL )
```

respectively.

For examples of how the derived data type `NLPT_problem_type` may be used in conjunction with the GALAHAD nonlinear feasibility code, see the specification sheets for the `GALAHAD_FILTRANE` package.