GALAHAD EXPO

USER DOCUMENTATION GALAHAD Optimization Library version 5.3

1 SUMMARY

This package uses an **exponential-penalty function to find a (local) minimizer of a differentiable objective function $f(\mathbf{x})$ of n variables $\mathbf{x}$, subject to m general constraints $\mathbf{c}^L \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}^U$ and simple bounds $\mathbf{x}^L \leq \mathbf{x} \leq \mathbf{x}^U$ on the variables.** Here, any of the components of the vectors of bounds $\mathbf{c}^L$, $\mathbf{c}^U$, $\mathbf{x}^L$ and $\mathbf{x}^U$ may be infinite. The method offers the choice of direct and iterative solution of the key unconstrained-optimization subproblems, and is most 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.

ATTRIBUTES — Versions: `GALAHAD_EXPO_single`, `GALAHAD_EXPO_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_NLPT`, `GALAHAD_SYMBOLS`, `GALAHAD_USERDATA`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_BSC`, `GALAHAD_MOP`, `GALAHAD_SSLS`, `GALAHAD_TRU`, `GALAHAD_GLTR`, `GALAHAD_STRINGS`, `GALAHAD_SPACE`, `GALAHAD_NORMS`, `GALAHAD_BLAS_interface`, and `GALAHAD_LAPACK_interface`. **Date:** may 2025. **Origin:** 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_EXPO_single
```

with the obvious substitution `GALAHAD_EXPO_double`, `GALAHAD_EXPO_quadruple`, `GALAHAD_EXPO_single_64`, `GALAHAD_EXPO_double_64` and `GALAHAD_EXPO_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`, `GALAHAD_userdata_type`, `EXPO_time_type`, `EXPO_control_type`, `EXPO_inform_type`, `EXPO_data_type` and `NLPT_problem_type`, (Section 2.4) and the subroutines `EXPO_initialize`, `EXPO_solve`, `EXPO_terminate`, (Section 2.5) and `EXPO_read_specfile` (Section 2.9) must be renamed on one of the `USE` statements.

2.1 Basic terminology

The exponential penalty function is defined to be

$$\begin{aligned}\phi(\mathbf{x},\mathbf{w},\boldsymbol{\mu},\mathbf{v},\boldsymbol{\nu}) &= f(x) + \sum_i \mu_i^L w_i^L \exp[(c_i^L - c_i(\mathbf{x}))/\mu_i^L] + \sum_i \mu_i^U w_i^U \exp[(c_i(\mathbf{x}) - c_i^U)/\mu_i^U] \\ &\quad + \sum_j \nu_j^L v_j^L \exp[(x_j^L - x_j)/\nu_j^L] + \sum_j \nu_j^U v_j^U \exp[(x_j - x_j^U)/\nu_j^U],\end{aligned} \tag{2.1}$$

where c_i^L, c_i^U and $c_i(\mathbf{x})$ are the i-th components of $\mathbf{c}^L$, $\mathbf{c}^U$ and $\mathbf{c}(\mathbf{x})$, and c_j^L, c_j^U and x_j are the j-th components of $\mathbf{x}^L$, $\mathbf{x}^U$ and $\mathbf{x}$, respectively. Here the components of $\boldsymbol{\mu}^L$, $\boldsymbol{\mu}^U$, $\boldsymbol{\nu}^L$ and $\boldsymbol{\nu}^U$ are separate **penalty parameters** for each lower and upper, general and simple-bound constraint, respectively, while those of $\mathbf{w}^L$, $\mathbf{w}^U$, $\mathbf{v}^L$, $\mathbf{v}^U$ are likewise separate **weights** for the same. The algorithm iterates by approximately minimizing $\phi(\mathbf{x},\mathbf{w},\boldsymbol{\mu},\mathbf{v},\boldsymbol{\nu})$ for a fixed set of penalty parameters and weights, and then adjusting these parameters and weights. The adjustments are designed so the sequence of approximate minimizers of ϕ converge to that of the specified constrained optimization problem.

Key constructs are the **gradient** of the objective function

$$\mathbf{g}(\mathbf{x}) \stackrel{\text{def}}{=} \nabla_x f(\mathbf{x}), \tag{2.2}$$

the **Jacobian** of the vector of constraints,

$$\mathbf{J}(\mathbf{x}) \stackrel{\text{def}}{=} \nabla_x c(\mathbf{x}), \tag{2.3}$$

and the **gradient** and **Hessian** of the Lagrangian function

$$\mathbf{g}_{\text{L}}(\mathbf{x},\mathbf{y}) \stackrel{\text{def}}{=} \mathbf{g}(\mathbf{x}) - \mathbf{J}^T(\mathbf{x})\mathbf{y} - \mathbf{z} \;\text{ and }\; \mathbf{H}_{\text{L}}(\mathbf{x},\mathbf{y}) \stackrel{\text{def}}{=} \nabla_{xx}\left[f(\mathbf{x}) - \sum_i y_i \nabla^2 c_i(\mathbf{x})\right] \tag{2.4}$$

for given vectors **y** and **z**.

The required solution **x** necessarily satisfies the primal optimality conditions

$$\mathbf{c}^{\text{L}} \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}^{\text{U}} \;\text{ and }\; \mathbf{x}^{\text{L}} \leq \mathbf{x} \leq \mathbf{x}^{\text{U}}, \tag{2.5}$$

the dual optimality conditions

$$\mathbf{g}(\mathbf{x}) = \mathbf{J}^T(\mathbf{x})\mathbf{y} + \mathbf{z} \tag{2.6}$$

where

$$\mathbf{y} = \mathbf{y}^{\text{L}} - \mathbf{y}^{\text{U}}, \;\; \mathbf{z} = \mathbf{z}^{\text{L}} - \mathbf{z}^{\text{U}}, \text{ and } (\mathbf{y}^{\text{L}},\mathbf{y}^{\text{U}},\mathbf{z}^{\text{L}},\mathbf{z}^{\text{U}}) \geq 0, \tag{2.7}$$

and the complementary slackness conditions

$$(\mathbf{c}(\mathbf{x}) - \mathbf{c}^{\text{L}})^T\mathbf{y}^{\text{L}} = 0, \;\; (\mathbf{c}(\mathbf{x}) - \mathbf{c}^{\text{U}})^T\mathbf{y}^{\text{U}} = 0, \;\; (\mathbf{x} - \mathbf{x}^{\text{L}})^T\mathbf{z}^{\text{L}} = 0 \;\text{ and }\; (\mathbf{x} - \mathbf{x}^{\text{U}})^T\mathbf{z}^{\text{U}} = 0, \tag{2.8}$$

where the vectors **y** and **z** are known as the Lagrange multipliers for the general constraints, and the dual variables for the simple bounds, respectively, and where the vector inequalities hold component-wise.

2.2 Matrix storage formats

The matrices $\mathbf{J}(\mathbf{x})$ and $\mathbf{H}_{\text{L}}(\mathbf{x},\mathbf{y})$ (as required and when available) may be stored in a variety of input formats.

2.2.1 Dense storage format

The matrix **J** 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 $\mathbf{J}_{i,j}$ for $i = 1,\ldots,m$, $j = 1,\ldots,n$. Since $\mathbf{H}_{\text{L}}$ is symmetric, only the lower triangular part (that is the part $\mathbf{H}_{\text{L}ij}$ for $1 \leq j \leq i \leq n$) should be stored. 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 $\mathbf{H}_{\text{L}ij}$ (and, by symmetry, $\mathbf{H}_{\text{L}ji}$) for $1 \leq j \leq i \leq n$.

2.2.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of **J**, its row index i, column index j and value $\mathbf{J}_{ij}$ 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 required. The same scheme is applicable to $\mathbf{H}_{\text{L}}$ (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 should be stored.

2.2.3 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 the 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 $\mathbf{J}_{ij}$ of the entries in the i-th row are stored in components $l =$ `J%ptr`(i), $\ldots$,`J%ptr` $(i+1)-1$ of the integer array `J%col`, and real array `J%val`, respectively. The same scheme is applicable to $\mathbf{H}_{\text{L}}$ (thus requiring integer arrays `H%ptr`, `H%col`, and a real array `H%val`), except that only the entries in the lower triangle should be stored.

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

2.2.4 Diagonal storage format

If $\mathbf{H}_{\text{L}}$ is diagonal (i.e., $\mathbf{H}_{\text{L}ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonal entries $\mathbf{H}_{\text{L}ii}$ for $1 \leq i \leq n$ need be stored, and the first n components of the array `H%val` may be used for the purpose. There is no sensible equivalent for the non-square matrix $\mathbf{J}$.

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 types

Seven derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the Jacobian matrix $\mathbf{J}$ and Hessian matrix $\mathbf{H}_{\text{L}}$ if these are available. The components of `SMT_TYPE` used here are:

`m` is a scalar component of type `INTEGER(ip_)`, that holds the row dimension of the matrix.

`n` is a scalar component of type `INTEGER(ip_)`, that holds the column 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 §2.4.2).

`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 $\mathbf{H}_{\text{L}}$ is represented as a single entry (see §2.2.1–2.2.3). 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 §2.2.2).

`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 §2.2.2–2.2.3).

`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 §2.2.3).

2.4.2 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.

`m` is a scalar variable of type `INTEGER(ip_)`, that holds the number of general constraints, m.

`H` is scalar variable of type `SMT_TYPE` that holds the Hessian matrix of the Lagrangian, $\mathbf{H}_L$. 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 2.2.1) is used, the first five components of `H%type` must contain the string `DENSE`. For the sparse co-ordinate scheme (see Section 2.2.2), the first ten components of `H%type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see Section 2.2.3), the first fourteen components of `H%type` must contain the string `SPARSE_BY_ROWS`, and for the diagonal storage scheme (see Section 2.2.4), 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 `EXPO_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 **H** in the sparse co-ordinate storage scheme (see Section 2.2.2). 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 **H** in any of the storage schemes discussed in Section 2.2.

`H%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.2.2). 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 **H** in either the sparse co-ordinate (see Section 2.2.2), or the sparse row-wise (see Section 2.2.3) 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 **H**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.2.3). 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`, $j = 1, \ldots, n$, contains $\mathbf{g}_j$.

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

`J` is scalar variable of type `SMT_TYPE` that holds the Jacobian matrix $\mathbf{J}(\mathbf{x})$ (if it is available). The following components are used here:

J%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 2.2.1) is used, the first five components of J%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.2.2), the first ten components of J%type must contain the string COORDINATE, and for the sparse row-wise storage scheme (see Section 2.2.3), the first fourteen components of J%type must contain the string SPARSE_BY_ROWS.

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

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

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

J%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in **J** in the sparse co-ordinate storage scheme (see Section 2.2.2). It need not be set for any of the other two schemes.

J%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the Jacobian matrix **J** in any of the storage schemes discussed in Section 2.2.

J%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of **J** in the sparse co-ordinate storage scheme (see Section 2.2.2). It need not be allocated for any of the other two schemes.

J%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of **J** in either the sparse co-ordinate (see Section 2.2.2), or the sparse row-wise (see Section 2.2.3) storage scheme. It need not be allocated when the dense 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 **J**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.2.3). It need not be allocated when the other schemes are used.

C is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the constraint function $\mathbf{c}(\mathbf{x})$. The i-th component of C, $j = 1,\ldots,m$, contains $\mathbf{c}_j(\mathbf{x})$.

C_l is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of lower bounds $\mathbf{c}^{\mathrm{L}}$ on the constraints. The i-th component of C_l, $i = 1,\ldots,m$, contains c_i^{L}. Infinite bounds are allowed by setting the corresponding components of C_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.4.3).

C_u is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of upper bounds $\mathbf{c}^{\mathrm{U}}$ on the constraints. The i-th component of C_u, $i = 1,\ldots,m$, contains c_i^{U}. Infinite bounds are allowed by setting the corresponding components of C_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.4.3).

X_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{x}^{\mathrm{L}}$ on the the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains 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 2.4.3).

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{x}^{\mathrm{U}}$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains 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 2.4.3).

GL is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the gradient $\mathbf{g}_{\mathrm{L}}$ of the Lagrangian function. The j-th component of GL, $j = 1,\ldots,n$, contains $\mathbf{g}_{\mathrm{L}\,j}$.

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, $j = 1,\ldots,n$, contains 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`, $j = 1, \ldots, n$, contains x_j.

`Y` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the values $\mathbf{y}$ of the Lagrange multipliers. The i-th component of `Y`, $i = 1, \ldots, m$, contains y_i.

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

`CNAMES` is a rank-one allocatable array of dimension `m` and type default `CHARACTER` and length 10, whose i-th entry contains the "name" of the i-th constraint 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.

2.4.3 The derived data type for holding control parameters

The derived data type `EXPO_control_type` is used to hold controlling data. Default values may be obtained by calling `EXPO_initialize` (see Section 2.5.1), while components may also be changed by calling `GALAHAD_EXPO_read_spec` (see Section 2.9.1). The components of `EXPO_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 `EXPO_solve` and `EXPO_terminate` is suppressed if `error` ≤ 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 `EXPO_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` ≤ 0. If `print_level` $= 1$, a single line of output will be produced for each iteration of the process. If `print_level` ≥ 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 `EXPO_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 `EXPO_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`.

`max_it` is a scalar variable of type `INTEGER(ip_)`, that holds the maximum number of iterations which that will be allowed in `EXPO_solve`. The default is `max_it = 100`.

`max_eval` is a scalar variable of type `INTEGER(ip_)`, that holds the maximum number of function evaluations that will be allowed in `EXPO_solve`. The default is `max_eval = 10000`.

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_EXPO_solve, and execution of GALAHAD_EXPO_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 ≤ 0, no temporary file will be created, and execution cannot be terminated in this way. The default is alive_unit $= 60$.

update_multipliers_itmin is a scalar variable of type INTEGER(ip_), that holds the smallest iteration number for which a Lagrange multipliers/dual variables update will be attempted. Up until this value, only penalty parameter reductions will be allowed. The default is update_multipliers_itmin = 0.

update_multipliers_tol is a scalar variable of type REAL(rp_), that is used to specify the minimum value the dual infeasibility is allowed to be before Lagrange multipliers/dual variables updates will be attempted. The default is update_multipliers_tol = 10^{19}.

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 = 10^{19}.

stop_abs_p and stop_rel_p are scalar variables of type REAL(rp_), that hold the required absolute and relative accuracy for the primal infeasibility (see Section 4). The absolute value of each component of the primal infeasibility on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_p = stop_rel_p = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_EXPO_double).

stop_abs_d and stop_rel_d are scalar variables of type REAL(rp_), that hold the required absolute and relative accuracy for the dual infeasibility (see Section 4). The absolute value of each component of the dual infeasibility on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_d = stop_rel_d = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_EXPO_double).

stop_abs_c and stop_rel_c are scalar variables of type REAL(rp_), that hold the required absolute and relative accuracy for the violation of complementary slackness (see Section 4). The absolute value of each component of the complementary slackness on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_c = stop_rel_c = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_EXPO_double).

stop_s is a scalar variable of type REAL(rp_), that is used to specify the minimum acceptable correction step $\mathbf{s}$ relative to the current estimate of the solution $\mathbf{x}$ The algorithm will be deemed to have converged if $|s_i| \leq$ stop_s $*\max(1,|x_i|)$ for all $i = 1,\ldots,n$. The default is stop_s = u, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_EXPO_double).

stop_subproblem_rel is a scalar variable of type REAL(rp_), that determines the required accuracy of the subproblem solver GALAHAD_TRU. The subproblem minimization will be stopped as soon as the relative decrease in the subproblem gradient falls below stop_subproblem_rel. If stop_subproblem_rel is 1.0 or bigger or 0.0 or smaller, this value will be ignored, and the choice of stopping rule delegated to control_tru%stop_g_relative (see below). The default is stop_subproblem_rel = -1.0.

initial_mu is a scalar variable of type REAL(rp_), that holds the required initial value of the penalty parameter. If initial_radius ≤ 0, the initial penalty parameter will be chosen automatically by GALAHAD_EXPO_solve The default is initial_radius = 0.1.

mu_reduce is a scalar variable of type REAL(rp_), that holds the amount by which the penalty parameter is reduced at the end of an iteration. The default is mu_reduce = 0.5.

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_EXPO_double).

`try_advanced_start` is a scalar variable of type default `REAL(rp_)`, that specifies the largest value of the KKT residuals before an advanced start will be attempted. The default is `try_advanced_start` = 10^{-2}.

`try_sqp_start` is a scalar variable of type default `REAL(rp_)`, that specifies the largest value of the KKT residuals before an advanced SQP start will be attempted. The default is `try_sqp_start` = 10^{-4}.

`stop_advanced_start` is a scalar variable of type default `REAL(rp_)`, that specifies the smallest value of the KKT residuals that an advanced start will be attempted. The default is `stop_advanced_start` = 10^{-8}.

`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 2.7), and `.FALSE.` if the second derivatives are not explicitly available. The default is `hessian_available = .TRUE.`. **N.B.** `.FALSE.` **is not yet implemented.**

`subproblem_direct` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if a direct (factorization) method is desired when solving for the step, and `.FALSE.` if an iterative method suffices. The default is `subproblem_direct = .TRUE.`. **N.B.** `.FALSE.` **is not yet implemented.**

`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_EXPO_solve`. The default is `alive_unit` = `ALIVE.d`.

`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 ignoreing 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 = ""`.

`BSC_control` is a scalar variable of type `BSC_control_type` whose components are used to control the formation of the Hessian matrix of the penalty function, as performed by the package `GALAHAD_BSC`. See the specification sheet for the package `GALAHAD_BSC` for details, and appropriate default values.

`TRU_control` is a scalar variable of type `TRU_control_type` whose components are used to control the minimization of the penalty function, performed by the package `GALAHAD_TRU`. See the specification sheet for the package `GALAHAD_TRU` for details, and appropriate default values (but note that value for `TRU_control%hessian_available`, will be overridden by `GALAHAD_EXPO_solve`).

`SSLS_control` is a scalar variable of type `SSLS_control_type` whose components are used to control the linear solve aspects of the calculation, as performed by the package `GALAHAD_SSLS`. See the specification sheet for the package `GALAHAD_SSLS` for details, and appropriate default values.

2.4.4 The derived data type for holding timing information

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

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

preprocess is a scalar variable of type REAL(rp_), that gives the CPU time spent reordering the problem to standard form prior to solution.

analyse is a scalar variable of type REAL(rp_), that gives the CPU time spent analysing required matrices prior to factorization.

factorize is a scalar variable of type REAL(rp_), that gives the CPU time spent factorizing the required matrices.

solve is a scalar variable of type REAL(rp_), that gives the CPU time spent using the factors to solve relevant linear equations.

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

clock_preprocess is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent reordering the problem to standard form prior to solution.

clock_analyse is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent analysing required matrices prior to factorization.

clock_factorize is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent factorizing the required matrices.

clock_solve is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent using the factors to solve relevant linear equations.

2.4.5 The derived data type for holding informational parameters

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

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Sections 2.7 and 2.8 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.

n_free is a scalar variable of type INTEGER(ip_), that holds the number of variables that are free from their bounds.

iter is a scalar variable of type INTEGER(ip_), that holds the number of iterations performed.

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

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

hl_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.

primal_infeasibility is a scalar variable of type REAL(rp_), that holds the norm of the violation of primal optimality (see Section 2.4.4) at the best estimate of the solution found.

dual_infeasibility is a scalar variable of type REAL(rp_), that holds the norm of the violation of dual optimality (see Section 2.4.4) at the best estimate of the solution found.

complementary_slackness is a scalar variable of type REAL(rp_), that holds the norm of the violation of complementary slackness (see Section 2.4.4) at the best estimate of the solution found.

time is a scalar variable of type EXPO_time_type whose components are used to hold elapsed elapsed CPU and system clock times for the various parts of the calculation (see Section 2.4.4).

BSC_inform is a scalar variable of type BSC_inform_type whose components give information about the formation of the Hessian matrix of the penalty function, as performed by the package GALAHAD_BSC. See the specification sheet for the package GALAHAD_BSC for details.

TRU_inform is a scalar variable of type TRU_inform_type whose components give information about the progress and needs of the algorithm used to minimize the penalty function, as performed by the package GALAHAD_TRU. See the specification sheet for the package GALAHAD_TRU for details.

SSLS_inform is a scalar variable of type SSLS_inform_type whose components give information about the progress and needs of the linear-solve stages of the algorithm performed by the package GALAHAD_SSLS. See the specification sheet for the package GALAHAD_SSLS for details.

2.4.6 The derived data type for holding problem data

The derived data type EXPO_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of EXPO procedures. This data should be preserved, untouched (except as directed on return from GALAHAD_EXPO_solve with positive values of inform%status, see Section 2.7), from the initial call to EXPO_initialize to the final call to EXPO_terminate.

2.4.7 The derived data type for holding user data

The derived data type GALAHAD_userdata_type is available from the package GALAHAD_userdata to allow the user to pass data to and from user-supplied subroutines for function and derivative calculations (see Section 2.6). Components of variables of type GALAHAD_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_)

complex is a rank-one allocatable array of type default COMPLEX (double precision complex in GALAHAD_EXPO_-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_EXPO_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`.

2.5 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.9 for further features):

1. The subroutine `EXPO_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 `EXPO_solve` is called to solve the problem.
3. The subroutine `EXPO_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `EXPO_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 `EXPO_initialize` cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets `[ ]` to indicate `OPTIONAL` arguments.

2.5.1 The initialization subroutine

Default values are provided as follows:

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

`data` is a scalar `INTENT(INOUT)` argument of type `EXPO_data_type` (see Section 2.4.6). It is used to hold data about the problem being solved.

`control` is a scalar `INTENT(OUT)` argument of type `EXPO_control_type` (see Section 2.4.3). On exit, `control` contains default values for the components as described in Section 2.4.3. These values should only be changed after calling `EXPO_initialize`.

`inform` is a scalar `INTENT(OUT)` argument of type `EXPO_inform_type` (see Section 2.4.5). A successful call to `EXPO_initialize` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.8.

2.5.2 The minimization subroutine

The minimization algorithm is called as follows:

```
CALL EXPO_solve( nlp, control, inform, data, userdata[, eval_FC, eval_GJ,      &
                 eval_HL, eval_HLPROD] )
```

`nlp` is a scalar `INTENT(INOUT)` argument of type `NLPT_problem_type` (see Section 2.4.2). 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 2.2 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 `EXPO_control_type` (see Section 2.4.3). Default values may be assigned by calling `EXPO_initialize` prior to the first call to `EXPO_solve`. Note that value for `TRU_control%-hessian_available`, will be overridden by `GALAHAD_EXPO_solve`.

`inform` is a scalar `INTENT(INOUT)` argument of type `EXPO_inform_type` (see Section 2.4.5). On initial entry, the component `status` must be set to the value 1. Other entries need note be set. A successful call to `EXPO_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see Sections 2.7 and 2.8.

`data` is a scalar `INTENT(INOUT)` argument of type `EXPO_data_type` (see Section 2.4.6). It is used to hold data about the problem being solved. With the possible exceptions of the components `eval_status` and `U` (see Section 2.7), it must not have been altered **by the user** since the last call to `EXPO_initialize`.

`userdata` is a scalar `INTENT(INOUT)` argument of type `GALAHAD_userdata_type` whose components may be used to communicate user-supplied data to and from the `OPTIONAL` subroutines `eval_FC`, `eval_GJ`, `eval_HL` and `eval_HLPROD` (see Section 2.4.7).

`eval_FC` is an `OPTIONAL` user-supplied subroutine whose purpose is to evaluate the value of the objective function $f(\mathbf{x})$ and constraints $\mathbf{c}(\mathbf{x})$ at a given vector $\mathbf{x}$. See Section 2.6.1 for details. If `eval_FC` is present, it must be declared `EXTERNAL` in the calling program. If `eval_FC` is absent, `GALAHAD_EXPO_solve` will use reverse communication to obtain function values (see Section 2.7).

`eval_GJ` is an `OPTIONAL` user-supplied subroutine whose purpose is to evaluate the value of the gradient of the objective function $\mathbf{g}(\mathbf{x})$ in (2.2) and the Jacobian of the constraints $\mathbf{J}(\mathbf{x})$ in (2.3) at a given vector $\mathbf{x}$. See Section 2.6.2 for details. If `eval_GJ` is present, it must be declared `EXTERNAL` in the calling program. If `eval_GJ` is absent, `GALAHAD_EXPO_solve` will use reverse communication to obtain gradient values (see Section 2.7).

`eval_HL` is an `OPTIONAL` user-supplied subroutine whose purpose is to evaluate the value of the Hessian of the Lagrangian function $\mathbf{H}_{\text{L}}(\mathbf{x},\mathbf{y})$ in (2.4) at a given vectors $\mathbf{x}$ and $\mathbf{y}$. See Section 2.6.3 for details. If `eval_H` is present, it must be declared `EXTERNAL` in the calling program. If `eval_H` is absent, `GALAHAD_EXPO_solve` will use reverse communication to obtain Hessian values (see Section 2.7).

`eval_HLPROD` is an `OPTIONAL` user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{H}_{\text{L}}(\mathbf{x},\mathbf{y})\mathbf{v}$ of the Hessian of the Lagrangian function with a given vector $\mathbf{v}$. See Section 2.6.4 for details. If `eval_HLPROD` is present, it must be declared `EXTERNAL` in the calling program. If `eval_HLPROD` is absent, `GALAHAD_EXPO_solve` will use reverse communication to obtain Hessian-vector products (see Section 2.7).

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

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

`inform` is a scalar `INTENT(OUT)` argument of type `EXPO_inform_type` exactly as for `EXPO_solve`. Only the component `status` will be set on exit, and a successful call to `EXPO_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see Section 2.8.

2.6 Function and derivative values

2.6.1 objective function and constraint values via internal evaluation

If the argument eval_FC is present when calling GALAHAD_EXPO_solve, the user is expected to provide a subroutine of that name to evaluate the values of the objective function $f(\mathbf{x})$ and/or the constraints $\mathbf{c}(\mathbf{x})$.

The routine must be specified as

```
SUBROUTINE eval_FC( status, X, userdata[, f, C] )
```

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 constraints as required, 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 GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_FC, eval_GJ, eval_HL and eval_HLPROD (see Section 2.4.7).

f is an OPTIONAL 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 if f is PRESENT.

C is an OPTIONAL rank-one INTENT(OUT) argument of type REAL(rp_), whose components should be set to the values of the constraints $\mathbf{c}(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X if C is PRESENT.

2.6.2 Gradient and Jacobian values via internal evaluation

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

```
SUBROUTINE eval_GJ( status, X, userdata[, G, J_val] )
```

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 and Jacobian if required, 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 GALAHAD_userdata_type whose components may be used to communicate user-supplied data to and from the subroutines eval_FC, eval_GJ, eval_HL and eval_HLPROD (see Section 2.4.7).

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

J_val is an OPTIONAL scalar INTENT(OUT) argument of type REAL(rp_), whose components should be set to the values of the Jacobian $\mathbf{J}(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X if J_val is PRESENT. The values should be input in the same order as that in which the array indices were given in nlp%J.

2.6.3 Hessian values via internal evaluation

If the argument `eval_HL` is present when calling `GALAHAD_EXPO_solve`, the user is expected to provide a subroutine of that name to evaluate the values of the Hessian of the Lagrangian function $\mathbf{H}_L(\mathbf{x},\mathbf{y})$. The routine must be specified as

```
SUBROUTINE eval_HL( status, X, Y, userdata, H_val )
```

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 Lagrangian 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 $\mathbf{x}$.

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

`userdata` is a scalar `INTENT(INOUT)` argument of type `GALAHAD_userdata_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_FC`, `eval_GJ`, `eval_HL` and `eval_HLPROD` (see Section 2.4.7).

`H_val` is a scalar `INTENT(OUT)` argument of type `REAL(rp_)`, whose components should be set to the values of the Hessian of the Lagrangian function $\mathbf{H}_L(\mathbf{x},\mathbf{y})$ in (2.4) evaluated at the vectors $\mathbf{x}$ and $\mathbf{y}$ input in `X` and `Y`. The Hessian values should be input in the same order as that in which the array indices were given in `nlp%H`.

2.6.4 Hessian-vector products via internal evaluation

N.B. not yet implemented. If the argument `eval_HLPROD` is present when calling `GALAHAD_EXPO_solve`, the user is expected to provide a subroutine of that name to evaluate the sum $\mathbf{u}+\mathbf{H}_L(\mathbf{x},\mathbf{y})\mathbf{v}$ involving the product of the Hessian of the Lagrangian function $\mathbf{H}_L(\mathbf{x},\mathbf{y})$ with a given vector $\mathbf{v}$. The routine must be specified as

```
SUBROUTINE eval_HLPROD( status, X, Y, 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 $\mathbf{u}+\mathbf{H}_L(\mathbf{x},\mathbf{y})\mathbf{v}$, 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 $\mathbf{x}$.

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

`userdata` is a scalar `INTENT(INOUT)` argument of type `GALAHAD_userdata_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_FC`, `eval_GJ`, `eval_HL` and `eval_HLPROD` (see Section 2.4.7).

`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 and $\mathbf{y}$, `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.

2.7 Reverse Communication Information

N.B. not yet implemented. A positive value of `inform%status` on exit from `EXPO_solve` indicates that `GALAHAD_EXPO_solve` is seeking further information—this will happen if the user has chosen not to evaluate function or derivative values internally (see Section 2.6). The user should compute the required information and re-enter `GALAHAD_EXPO_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 $f(\mathbf{x})$ and the constraint values $\mathbf{c}(\mathbf{x})$ at the point $\mathbf{x}$ indicated in `nlp%X`. The required values should be set in `nlp%f` and `nlp%c`, and `data%eval_status` should be set to 0. If the user is unable to evaluate $f(\mathbf{x})$ or $\mathbf{c}(\mathbf{x})$—for instance, if the function is undefined at $\mathbf{x}$—the user need not set `nlp%f` or $\mathbf{c}(\mathbf{x})$, but should then set `data%eval_status` to a non-zero value.

3. The user should compute the gradient $\mathbf{g}(\mathbf{x})$ of the objective function and the Jacobian $\mathbf{J}(\mathbf{x})$ of the constraints 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 $i = 1,\ldots,n$ and those for the Jacobian should be set in `nlp%J%val` (in the same order as that in which the array indices were given in `nlp%J`, and `data%eval_status` should be set to 0. If the user is unable to evaluate a component of $\mathbf{g}(\mathbf{x})$ or $\mathbf{J}(\mathbf{x})$—for instance, if a component of the gradient is undefined at $\mathbf{x}$—the user need not set `nlp%G` and `nlp%J%val`, but should then set `data%eval_status` to a non-zero value.

4. The user should compute the Hessian $\mathbf{H}_{\mathrm{L}}(\mathbf{x},\mathbf{y})$ of the Lagrangian function at the point $\mathbf{x}$ and $\mathbf{y}$ indicated in `nlp%X` and `nlp%Y`, respectively. The value l-th component of the Hessian stored according to the scheme input in the remainder of `nlp%H` (see Section 2.4.2) should be set in `nlp%H%val(l)`, for $l = 1,\ldots,$ `nlp%H%ne`, and `data%eval_status` should be set to 0. If the user is unable to evaluate a component of $\mathbf{H}_{\mathrm{L}}(\mathbf{x},\mathbf{y})$—for instance, if a component of the Hessian is undefined at $\mathbf{x}$ and $\mathbf{y}$—the user need not set `nlp%H%val`, but should then set `data%eval_status` to a non-zero value.

5. The user should compute the product $\mathbf{H}_{\mathrm{L}}(\mathbf{x},\mathbf{y})\mathbf{v}$ of the Hessian of the Lagrangian function $\mathbf{H}_{\mathrm{L}}(\mathbf{x},\mathbf{y})$ at the point $\mathbf{x}$ and $\mathbf{y}$ indicated in `nlp%X` and `nlp%Y` 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 $\mathbf{u}+\mathbf{H}_{\mathrm{L}}(\mathbf{x},\mathbf{y})\mathbf{v}$ 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 $\mathbf{x}$ and $\mathbf{y}$—the user need not set `nlp%H%val`, but should then set `data%eval_status` to a non-zero value.

2.8 Warning and error messages

A negative value of `inform%status` on exit from `EXPO_solve` or `EXPO_terminate` indicates that an error has occurred. 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.

-7. The objective function appears to be unbounded from below on the feasible set.

-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`.

-15. The preconditioner $\mathbf{P}(\mathbf{x})$ appears not to be positive definite.

-16. The problem is so ill-conditioned that further progress is impossible.

-17. The step is too small to make further impact.

-18. Too many iterations have been performed. This may happen if `control%maxit` is too small, but may also be symptomatic of a badly scaled problem.

-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_EXPO_solve` by removing the file named `control%alive_file` from unit `control%alive_unit`.

2.9 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `EXPO_control_type` (see Section 2.4.3), by reading an appropriate data specification file using the subroutine `EXPO_read_specfile`. This facility is useful as it allows a user to change `EXPO` control parameters without editing and recompiling programs that call `EXPO`.

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 `EXPO_read_specfile` must start with a "`BEGIN EXPO`" command and end with an "`END`" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by EXPO_read_specfile .. )
  BEGIN EXPO
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by EXPO_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "`BEGIN EXPO`" 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

```
BEGIN EXPO SPECIFICATION
```

and

```
END EXPO SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN EXPO" 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 EXPO_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 EXPO_read_specfile.

2.9.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL EXPO_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type EXPO_control_type (see Section 2.4.3). Default values should have already been set, perhaps by calling EXPO_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 2.4.3) of control that each affects are given in Table 2.1 on the following 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.

2.10 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 for each iteration of the process. This will include the values of the objective function and the norm of its gradient, the ratio of actual to predicted decrease following the step, the radius of the trust-region and the time taken so far. In addition, if a direct solution of the subproblem has been attempted, the Lagrange multiplier from the secular equation and the number of factorizations used will be recorded, while if an iterative solution has been used, the numbers of phase 1 and 2 iterations will be given.

If control%print_level $\geq$ 2 this output will be increased to provide significant detail of each iteration. This extra output includes residuals of the linear systems solved, and, for larger values of control%print_level, values of the variables and gradients. Further details concerning the attempted solution of the models may be obtained by increasing control%TRU_control%print_level, control%SSLS_control%print_level and control%GLTR_control%print_level, while details about factorizations are available by increasing control%SSLS_control%print_level. See the specification sheets for the packages GALAHAD_GLTR, GALAHAD_SSLS and GALAHAD_TRU for details.

3 GENERAL INFORMATION

Use of common: None.

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	%max_it	integer
maximum-number-of-evaluations	%max_eval	integer
alive-device	%alive_unit	integer
update-multipliers-from-iteration	%update_multipliers_itmin	real
update-multipliers-feasibility-tolerance	%update_multipliers_tol	real
infinity-value	%infinity	real
absolute-primal-accuracy	%stop_abs_p	real
relative-primal-accuracy	%stop_rel_p	real
absolute-dual-accuracy	%stop_abs_d	real
relative-dual-accuracy	%stop_rel_d	real
absolute-complementary-slackness-accuracy	%stop_abs_c	real
relative-complementary-slackness-accuracy	%stop_rel_c	real
minimum-relative-step-allowed	%stop_s	real
relative-subproblem-accuracy	%stop_subproblem_rel	real
initial-penalty-parameter	initial_mu	real
penalty-parameter-reduction-factor	mu_reduce	real
minimum-objective-before-unbounded	obj_unbounded	real
try-advanced-start-tolerance	try_advanced_start	real
try-sqp-start-tolerance	try_sqp_start	real
stop-advanced-start-tolerance	stop_advanced_start	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
hessian-available	%hessian_available	logical
sub-problem-direct	%subproblem_direct	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.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: EXPO_solve calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_NLPT, GALAHAD_SYMBOLS, GALAHAD_USERDATA, GALAHAD_SPECFILE, GALAHAD_SMT, GALAHAD_BSC, GALAHAD_MOP, GALAHAD_SSLS, GALAHAD_TRU, GALAHAD_GLTR, GALAHAD_STRINGS, GALAHAD_SPACE, GALAHAD_NORMS, GALAHAD_BLAS_interface, and GALAHAD_LAPACK_interface.

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

The method employed involves a sequential minimization of the exponential penalty function (2.1) for a sequence of positive penalty parameters $(\boldsymbol{\mu}^{\text{L}}_k, \boldsymbol{\mu}^{\text{U}}_k, \boldsymbol{\nu}^{\text{L}}_k, \boldsymbol{\nu}^{\text{U}}_k)$ and weights $(\mathbf{w}^{\text{L}}_k, \mathbf{w}^{\text{U}}_k, \mathbf{v}^{\text{L}}_k, \mathbf{v}^{\text{U}}_k)$, for increasing $k \geq 0$. Convergence is ensured if the penalty parameters are forced to zero, and may be accelerated by adjusting the weights. The minimization of (2.1) is accomplished using the trust-region unconstrained solver `GALAHAD_TRU`. Although critical points $\{x_k\}$ of $\phi(\mathbf{x}, \mathbf{w}_k, \boldsymbol{\mu}_k, \mathbf{v}_k, \boldsymbol{\nu}_k)$ converge to a local solution x_* of the underlying problem, the reduction of the penalty parameters to zero often results in x_k being a poor starting point for the minimization of $\phi(\mathbf{x}, \mathbf{w}_{k+1}, \boldsymbol{\mu}_{k+1}, \mathbf{v}_{k+1}, \boldsymbol{\nu}_{k+1})$. Consequently, a careful extrapolated starting point from x_k is used instead. Moreover, once the algorithm is confident that it is sufficiently close to x_*, it switches to Newton's method to accelerate the convergence. Both the extrapolation and the Newton iteration rely on the block-linear-system solver `GALAHAD_SSLS`.

The iteration is terminated as soon as residuals to the optimality conditions (2.5)–(2.8) are sufficiently small. For infeasible problems, this will not be possible, and instead the residuals to (2.5) will be made as small as possible.

References:

The method is described in detail in

N. Gould, S. Leyffer, A. Montoison and C. Vanaret (2025) The exponential multiplier method in the 21st century. RAL Technical Report, in preparation.

5 EXAMPLES OF USE

Suppose we wish to minimize the objective function $f(\mathbf{x}) = x_1^2 + x_2^2$ subject to the constraints

$$\begin{aligned} x_1 + x_2 &\geq 1 \\ x_1^2 + x_2^2 &\geq 1 \\ p x_1^2 + x_2^2 &\geq p \\ x_1^2 - x_2 &\geq 0 \\ x_2^2 - x_1 &\geq 0 \ \text{ and} \\ -50 \leq x_1, x_2 &\leq 50 \end{aligned}$$

when the parameter p takes the value 9. Starting from the initial guess $\mathbf{x} = (3, 1)$, we may use the following code:

```
   PROGRAM GALAHAD_EXPO_EXAMPLE  !  GALAHAD 5.3 - 2025-07-25 AT 11:15 GMT.
   USE GALAHAD_EXPO_double                      ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: rp = KIND( 1.0D+0 )    ! set precision
   TYPE ( NLPT_problem_type ):: nlp
   TYPE ( EXPO_control_type ) :: control
   TYPE ( EXPO_inform_type ) :: inform
   TYPE ( EXPO_data_type ) :: data
   TYPE ( GALAHAD_userdata_type ) :: userdata
   EXTERNAL :: FC, GJ, HL
   INTEGER :: s
   INTEGER, PARAMETER :: n = 2, m = 5, j_ne = 10, h_ne = 2
   REAL ( KIND = rp ), PARAMETER :: p = 9.0_rp
   REAL ( KIND = rp ), PARAMETER :: infinity = 10.0_rp ** 20    ! infinity
! start problem data
   nlp%pname = 'HS23'                           ! name
   nlp%n = n ; nlp%m = m ; nlp%H%ne = h_ne      ! dimensions
   ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
   ALLOCATE( nlp%C( m ), nlp%C_l( m ), nlp%C_u( m ) )
```

```
   nlp%X( 1 ) = 3.0_rp ; nlp%X( 2 ) = 1.0_rp
   nlp%X_l = - 50.0_rp ; nlp%X_u = 50.0_rp      ! variable bounds
   nlp%C_l = 0.0_rp ; nlp%C_u = infinity        ! constraint bounds
!  sparse row-wise storage format for the Jacobian
   CALL SMT_put( nlp%J%type, 'SPARSE_BY_ROWS', s ) ! specify sparse row storage
   ALLOCATE( nlp%J%val( j_ne ), nlp%J%col( j_ne ), nlp%H%ptr( m + 1 ) )
   nlp%J%col = (/ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 /)  ! Jacobian J
   nlp%J%ptr = (/ 1, 3, 5, 7, 9, 11 /)
!  sparse co-ordinate storage format for the Hessian
   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 /)              ! Hessian H
   nlp%H%col = (/ 1, 2 /)              ! NB lower triangle
! problem data complete
   ALLOCATE( userdata%real( 1 ) )               ! allocate space for parameter
   userdata%real( 1 ) = p                       ! record parameter, p
   CALL EXPO_initialize( data, control, inform ) ! initialize control parameters
   control%subproblem_direct = .TRUE.
   control%max_it = 20
   control%max_eval = 100
!  control%print_level = 1
!  control%tru_control%print_level = 1
   control%stop_abs_p = 1.0D-5
   control%stop_abs_d = 1.0D-5
   control%stop_abs_c = 1.0D-5
   inform%status = 1                            ! set for initial entry
   CALL EXPO_solve( nlp, control, inform, data, userdata, eval_FC = FC,        &
                    eval_GJ = GJ, eval_HL = HL ) ! solve problem
   IF ( inform%status == 0 ) THEN               ! successful return
     WRITE( 6, "( ' EXPO: ', I0, ' major iterations -',                        &
    &     ' optimal objective value =',                                         &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                 &
     inform%iter, inform%obj, nlp%X
   ELSE                                         ! error returns
     WRITE( 6, "( ' EXPO_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL EXPO_terminate( data, control, inform )  ! delete internal workspace
   DEALLOCATE( nlp%X, nlp%GL, nlp%H%val, nlp%H%row, nlp%H%col, userdata%real )
   DEALLOCATE( nlp%J%val, nlp%J%col, nlp%J%ptr )
   DEALLOCATE( nlp%C, nlp%X_l, nlp%X_u, nlp%C_l, nlp%C_u, nlp%G )
   END PROGRAM GALAHAD_EXPO_EXAMPLE

   SUBROUTINE FC( status, X, userdata, F, C )
   USE GALAHAD_USERDATA_double
   INTEGER, PARAMETER :: rp = KIND( 1.0D+0 )
   INTEGER ( KIND = ip_ ), INTENT( OUT ) :: status
   REAL ( kind = rp ), DIMENSION( : ), INTENT( IN ) :: X
   REAL ( kind = rp ), OPTIONAL, INTENT( OUT ) :: F
   REAL ( kind = rp ), DIMENSION( : ), OPTIONAL, INTENT( OUT ) :: C
   TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
   REAL ( kind = rp ) :: r
   r = userdata%real( 1 )
   f = X( 1 ) ** 2 + X( 2 ) ** 2
   C( 1 ) = X( 1 ) + X( 2 ) - 1.0_rp
   C( 2 ) = X( 1 ) ** 2 + X( 2 ) ** 2 - 1.0_rp
```

```
C( 3 ) = r * X( 1 ) ** 2 + X( 2 ) ** 2 - r
C( 4 ) = X( 1 ) ** 2 - X( 2 )
C( 5 ) = X( 2 ) ** 2 - X( 1 )
status = 0
END SUBROUTINE FC

SUBROUTINE GJ( status, X, userdata, G, J_val )
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: rp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = rp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = rp ), DIMENSION( : ), OPTIONAL, INTENT( OUT ) :: G
REAL ( KIND = rp ), DIMENSION( : ), OPTIONAL, INTENT( OUT ) :: J_val
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( kind = rp ) :: r
r = userdata%real( 1 )
G( 1 ) = 2.0_rp * X( 1 )
G( 2 ) = 2.0_rp * X( 2 )
J_val( 1 ) = 1.0_rp
J_val( 2 ) = 1.0_rp
J_val( 3 ) = 2.0_rp * X( 1 )
J_val( 4 ) = 2.0_rp * X( 2 )
J_val( 5 ) = 2.0_rp * r * X( 1 )
J_val( 6 ) = 2.0_rp * X( 2 )
J_val( 7 ) = 2.0_rp * X( 1 )
J_val( 8 ) = - 1.0_rp
J_val( 9 ) = - 1.0_rp
J_val( 10 ) = 2.0_rp * X( 2 )
END SUBROUTINE GJ

SUBROUTINE HL( status, X, Y, userdata, H_val )
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: rp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = rp ), DIMENSION( : ), INTENT( IN ) :: X, Y
REAL ( KIND = rp ), DIMENSION( : ), INTENT( OUT ) :: H_val
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( kind = rp ) :: r
r = userdata%real( 1 )
H_val( 1 ) = 2.0_rp - 2.0_rp * ( Y( 2 ) + r * Y( 3 ) + Y( 4 ) )
H_val( 2 ) = 2.0_rp - 2.0_rp * ( Y( 2 ) + Y( 3 ) + Y( 5 ) )
END SUBROUTINE HL
```

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

```
EXPO: 11 major iterations - optimal objective value =  2.0000E+00
Optimal solution =   1.0000E+00  1.0000E+00
```