



Science and
Technology
Facilities Council



GALAHAD

TRB

USER DOCUMENTATION

GALAHAD Optimization Library version 5.5

1 SUMMARY

This package uses a **trust-region method to find a (local) minimizer of a differentiable objective function $f(\mathbf{x})$ of many variables $\mathbf{x}$, subject to 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 $\mathbf{x}^l$ and $\mathbf{x}^u$ may be infinite. The method offers the choice of direct and iterative solution of the key trust-region 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_TRB_single, GALAHAD_TRB_double. **Uses:** GALAHAD_CLOCK, GALAHAD_NLPT, GALAHAD_SYMBOLS, GALAHAD_SPECFILE, GALAHAD_SLS, GALAHAD_PSLs, GALAHAD_GLTR, GALAHAD_TRS, LANCELOT_CAUCHY, LANCELOT_CG, GALAHAD_LMS, GALAHAD_SHA, GALAHAD_MOP, GALAHAD_STRINGS, GALAHAD_SPACE, GALAHAD_NORMS, GALAHAD_BLAS_interface, and GALAHAD_LAPACK_interface. **Date:** July 2021. **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_TRB_single
```

with the obvious substitution GALAHAD_TRB_double, GALAHAD_TRB_quadruple, GALAHAD_TRB_single_64, GALAHAD_TRB_double_64 and GALAHAD_TRB_quadruple_64 for the other variants.

If it is required to use more than one of the modules at the same time, the derived types SMT_type, USERDATA_type, TRB_time_type, TRB_control_type, TRB_inform_type, TRB_data_type and NLPT_problem_type, (Section 2.3) and the subroutines TRB_initialize, TRB_solve, TRB_terminate, (Section 2.4) and TRB_read_specfile (Section 2.8) must be renamed on one of the USE statements.

2.1 Matrix storage formats

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

2.1.1 Dense storage format

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

2.1.2 Sparse co-ordinate storage format

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

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

2.1.3 Sparse row-wise storage format

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

2.1.4 Diagonal storage format

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

2.2 Real and integer kinds

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

2.3 The derived data types

Seven derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

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

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

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

2.3.2 The derived data type for holding the problem

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

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

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

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

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `H%type`. For example, if `nlp` is of derived type `TRB_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 2.1.2). It need not be set for any of the other three schemes.

`H%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the **lower triangular** part of the Hessian matrix 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 H in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other three schemes.

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

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

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

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

`X_l` is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that holds the vector of lower bounds 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`, where `infinity` is a component of the control array `control` (see Section 2.3.3).

`X_u` is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that holds the vector of upper bounds 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 that `infinity`, where `infinity` is a component of the control array `control` (see Section 2.3.3).

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

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

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

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

2.3.3 The derived data type for holding control parameters

The derived data type `TRB_control_type` is used to hold controlling data. Default values may be obtained by calling `TRB_initialize` (see Section 2.4.1), while components may also be changed by calling `GALAHAD_TRB_read_spec` (see Section 2.8.1). The components of `TRB_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 `TRB_solve` and `TRB_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 `TRB_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 `TRB_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 `TRB_solve`. If `stop_print` is negative, printing will occur once it has been started by `start_print`. The default is `stop_print` = -1.

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

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

`more_toraldo` is a scalar variable of type `INTEGER(ip_)`, that specifies the number of Moré-Toraldo projected searches that are to be used to improve upon the Cauchy point when finding the step. Any non-positive value results in a standard add-one-at-a-time conjugate-gradient search. The default is `more_toraldo` = 0.

`non_monotone` is a scalar variable of type `INTEGER(ip_)`, that specifies the history-length for non-monotone descent strategy. Any non-positive value results in standard monotone descent, for which merit function improvement occurs at each iteration. There are often definite advantages in using a non-monotone strategy with a modest history, since the occasional local increase in the merit function may enable the algorithm to move across (gentle) “ripples” in the merit function surface. However, we do not usually recommend large values of `non_monotone`. The default is `non_monotone` = 1.

`model` is a scalar variable of type `INTEGER(ip_)`, that specifies which model to be used to approximate $f(\mathbf{x})$ when computing the step. Possible values are:

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

- 0 the model is chosen automatically on the basis of which option looks likely to be the most efficient at any given stage of the solution process. Different models may be used at different stages. **Not yet implemented.**
- 1 a first-order model, not involving the Hessian, will be used.
- 2 a second-order model, using the Hessian, will be used.
- 3 a barely-second-order model, in which the Hessian is approximated by the identity matrix, will be used.
- 4 a secant-based sparse second-order model, in which the Hessian is approximated within its sparsity pattern using secant formulae will be used.
- 5 a secant-based second-order model, in which the Hessian is approximated by a limited-memory BFGS formula, will be used.
- 6 a secant-based second-order model, in which the Hessian is approximated by a limited-memory symmetric rank-one (SR1) formula, will be used.

The default is `model = 2`.

`norm` is a scalar variable of type `INTEGER(ip_)`, that specifies which norm is to be used to define the trust region. In particular the norm $\|\cdot\|$ will be defined by a symmetric, positive-definite matrix $\mathbf{P}$ so that for every vector $\mathbf{v}$, $\|\mathbf{v}\|^2 = \mathbf{v}^T \mathbf{P} \mathbf{v}$. If `%subproblem_direct = .FALSE.`, the same $\mathbf{P}$ also defines the preconditioner to be used to accelerate the generalized-Lanczos inner model minimization. Possible values are:

- 3 the user's own norm will be used.
- 2 a norm based on a limited-memory BFGS formula will be used.
- 1 the Euclidean (ℓ_2 -) norm is used.
- 0 the type is chosen automatically on the basis of which option looks likely to be the most efficient at any given stage of the solution process. Different norms may be used at different stages. **Not yet implemented.**
- 1 $\mathbf{P}$ is the diagonal of the Hessian matrix, suitably modified to ensure that it is significantly positive definite, is used.
- 2 $\mathbf{P}$ is the Hessian matrix whose entries outside a band of given semi-bandwidth are replaced by zeros (see `nsemib` below).
- 3 $\mathbf{P}$ is the Hessian matrix whose entries outside a bandwidth-reduced reordered band of given semi-bandwidth are replaced by zeros (see `nsemib` below).
- 4 $\mathbf{P}$ is the (possibly perturbed) Hessian, using the Schnabel-Eskow modification method to ensure that the resultant matrix is positive definite.
- 5 $\mathbf{P}$ is the (possibly perturbed) Hessian, using the Gill-Murray-Poncéleon-Saunders modification method to ensure that the resultant matrix is positive definite. **Not yet implemented.**
- 6 $\mathbf{P}$ will be that from the incomplete factorization of the Hessian using the Lin-Moré method.
- 7 $\mathbf{P}$ will be that from the incomplete factorization of the Hessian using the method implemented by HSL_MI28.
- 8 $\mathbf{P}$ will be that from the incomplete factorization of the Hessian using Munksgaars's method. **Not yet implemented.**
- 9 $\mathbf{P}$ will be that from an expanding band of the Hessian. **Not yet implemented.**
- 10 $\mathbf{P}$ will be that which gives a diagonalising norm as implemented in TRB_DPS. Note that this is currently **only** available when `subproblem_direct = .TRUE.` (see below).

The default is `norm = 1`.

`semi_bandwidth` is a scalar variable of type `INTEGER(ip_)`, that specifies the semi-bandwidth of $\mathbf{P}$ when `norm = 2`, if appropriate. The default is `semi_bandwidth = 5`.

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

`lbfgs_vectors` is a scalar variable of type `INTEGER(ip_)`, that specifies the number of limited-memory vectors used in the model when `model = 5` or `6`, and/or by the norm when `norm = -2`, if appropriate. The default is `lbfgs_vectors = 10`.

`max_dxcg` is a scalar variable of type `INTEGER(ip_)`, that specifies the maximum number of sparse difference vectors used by the model when `model = 4`. The default is `max_dxcg = 100`.

`icfs_vectors` is a scalar variable of type `INTEGER(ip_)`, that specifies the number of multiples of the problem dimension n that is available to hold fill-in when computing the Lin-Moré factorization. The default is `icfs_vectors = 10`.

`mi28_lsize` is a scalar variable of type `INTEGER(ip_)`, that specifies the maximum number of fill entries within each column of the incomplete factor L computed by HSL_MI28. In general, increasing `mi28_lsize` improves the quality of the preconditioner but increases the time to compute and then it. Values less than 0 are treated as 0. The default is `mi28_lsize = 10`.

`mi28_rsize` is a scalar variable of type `INTEGER(ip_)`, that specifies the the maximum number of entries within each column of the strictly lower triangular matrix R used in the computation of the preconditioner by HSL_MI28. Rank-1 arrays of size `mi28_rsize * n` are allocated internally to hold R . Thus the amount of memory used, as well as the amount of work involved in computing the preconditioner, depends on `mi28_rsize`. Setting `mi28_rsize > 0` generally leads to a higher quality preconditioner than using `mi28_rsize = 0`, and choosing `mi28_rsize ≥ mi28_lsize` is generally recommended. The default is `mi28_rsize = 10`.

`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_TRB_solve, and execution of GALAHAD_TRB_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`.

`advanced_start` is a scalar variable of type `INTEGER(ip_)` that specifies the number of evaluations of the objective function that may be performed If the user wishes to try to select a good initial value of the trust-region radius. If the user is content with the initial value provided, `advanced_start` should be set to 0, and this is the default.

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

`stop_pg_absolute` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted (infinity) norm of the projected gradient of the objective function (see Section 4) at the estimate of the solution sought. The default is `stop_pg_absolute = 10-5`.

`stop_pg_relative` is a scalar variable of type `REAL(rp_)`, that is used to specify the largest relative reduction in the norm of the projected gradient of the objective function that will be permitted (see Section 4) at the estimate of the solution sought compared to that at the initial point. The default is `stop_pg_relative = 1`.

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

`initial_radius` is a scalar variable of type `REAL(rp_)`, that holds the required initial value of the trust-region radius. If `initial_radius ≤ 0`, the radius will be chosen automatically by GALAHAD_TRB_solve. The default is `initial_radius = 100.0`.

`maximum_radius` is a scalar variable of type `REAL(rp_)`, that holds the largest permitted value of the trust-region radius as the algorithm proceeds. The default is `maximum_radius = 108`.

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

`stop_rel_cg` is a scalar variable of type `REAL(rp_)`, that is used to specify the largest relative residual that is tolerated before any conjugate-gradient inner iteration terminates. The default is `stop_rel_cg = 0.01`.

`radius_increase`, `radius_reduce` and `radius_reduce_max` are scalar variables of type `REAL(rp_)`, that control the maximum amounts by which the trust-region radius can contract or expand during an iteration. The radius will be decreased by powers of `radius_reduce`, but not in total more than `radius_reduce_max`, until it is smaller than the norm of the current step. It can be increased by at most a factor `radius_increase`. The defaults are `radius_increase = 2.0`, `radius_reduce = 0.5` and `radius_reduce_max = 0.0625`.

`eta_successful`, `eta_very_successful` and `eta_too_successful` are scalar variables of type default `REAL(rp_)`, that control the acceptance and rejection of the trial step and the updates to the trust-region radius. At every iteration, the ratio of the actual reduction in the merit function following the trial step to that predicted by the model is computed. The step is accepted whenever this ratio exceeds `eta_successful`; otherwise the trust-region radius will be reduced. If, in addition, the ratio exceeds `eta_very_successful` but not `eta_too_successful`, the trust-region radius may be increased. The defaults are `eta_successful = 10-8`, `eta_very_successful = 0.9` and `eta_too_successful = 2.0`.

`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_TRB_double`).

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

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

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

`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 = .FALSE..`

`retrospective_trust_region` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if a retrospective trust-region strategy, based on the model at the next iterate, is to be used, and `.FALSE.` if the traditional strategy suffices. The default is `retrospective_trust_region = .FALSE..`

`renormalize_radius` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if the trust-region radius is to be re-normalized to account for the shape of the trust-region norm every iteration, and `.FALSE.` if no re-normalization is required. The default is `renormalize_radius = .FALSE..`

`exact_gcp` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if the exact generalized Cauchy point, the first estimate of the minimizer of the quadratic model within the feasible box, is required, and `.FALSE.` if an approximation suffices. The default is `exact_gcp = .TRUE..`

`accurate_bqp` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if an accurate minimizer of the quadratic model within the feasible box is required, and `.FALSE.` if an approximation suffices. The accurate minimizer often requires considerably more work, but occasionally this reduces the overall number of iterations. The default is `accurate_bqp = .FALSE..`

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

`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_TRB_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 ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default `prefix = ""`.

`PSLS_control` is a scalar variable of type `PSLS_control_type` whose components are used to control the preconditioning aspects of the calculation, as performed by the package `GALAHAD_PSLs`. See the specification sheet for the package `GALAHAD_PSLs` for details, and appropriate default values (but note that values for `PSLS_control%preconditioner`, `PSLS_control%semi_bandwidth` and `PSLS_control%icfs_vectors` may be overridden by `GALAHAD_TRB_solve`).

`GLTR_control` is a scalar variable of type `GLTR_control_type` whose components are used to control the iterative trust-region step calculation (if any), performed by the package `GALAHAD_GLTR`. See the specification sheet for the package `GALAHAD_GLTR` for details, and appropriate default values (but note that value of `GLTR_control%unitm` may be changed by `GALAHAD_TRB_solve`).

`TRS_control` is a scalar variable of type `TRS_control_type` whose components are used to control the direct trust-region step calculation (if any), performed by the package `GALAHAD_TRS`. See the specification sheet for the package `GALAHAD_TRS` for details, and appropriate default values (but note that values of `TRS_control%initial_multiplier` and `TRS_control%new_h` may be changed by `GALAHAD_TRB_solve`).

`LMS_control` and `LMS_control_prec` are scalar variables of type `LMS_control_type` whose components are used to control the limited memory secant approximations for the model Hessian and trust region norm as performed by the package `GALAHAD_LMS`. See the specification sheet for the package `GALAHAD_LMS` for details, and appropriate default values.

`SHA_control` is a scalar variable of type `SHA_control_type` whose components are used to control the calculation of the sparse model Hessian (if required), performed by the package `GALAHAD_SHA`. See the specification sheet for the package `GALAHAD_SHA` for details, and appropriate default values.

2.3.4 The derived data type for holding timing information

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

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

`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.

2.3.5 The derived data type for holding informational parameters

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

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

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

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

`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.

`cg_iter` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of conjugate-gradient iterations required.

`cg_maxit` is a scalar variable of type `INTEGER(ip_)`, that gives the number of conjugate-gradient iterations permitted during each major iteration.

`factorization_status` is a scalar variable of type `INTEGER(ip_)`, that gives the return status from the matrix factorization.

`max_entries_factors` is a scalar variable of type `INTEGER(int64)`, that gives the maximum number of entries in any of the matrix factorizations performed during the calculation.

`factorization_max` is a scalar variable of type `INTEGER(ip_)`, that gives the largest number of factorizations required during a subproblem solution.

`factorization_integer` is a scalar variable of type default `INTEGER(ip_)`, that gives the amount of integer storage used for the matrix factorization.

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

`factorization_real` is a scalar variable of type `INTEGER(ip_)`, that gives the amount of real storage used for the matrix factorization.

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

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

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

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

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

`radius` is a scalar variable of type `REAL(rp_)`, that holds the current value of the trust-region radius

`factorization_average` is a scalar variable of type `REAL(rp_)`, that gives the average number of factorizations per subproblem solved.

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

`PSLS_inform` is a scalar variable of type `PSLS_inform_type` whose components give information about the progress and needs of the preconditioning stages of the algorithm performed by the package `GALAHAD_PSLs`. See the specification sheet for the package `GALAHAD_PSLs` for details.

`GLTR_inform` is a scalar variable of type `GLTR_inform_type` whose components give information about the progress and needs of the iterative solution stages of the algorithm performed by the package `GALAHAD_GLTR`. See the specification sheet for the package `GALAHAD_GLTR` for details.

`TRS_inform` is a scalar variable of type `TRS_inform_type` whose components give information about the progress and needs of the direct solution stages of the algorithm performed by the package `GALAHAD_TRS`. See the specification sheet for the package `GALAHAD_TRS` for details.

`LMS_inform` and `LMS_inform_prec` are scalar variables of type `LMS_inform_type` whose components give information about the progress and needs of the limited memory secant approximations for the model Hessian and trust region norm as performed by the package `GALAHAD_LMS`. See the specification sheet for the package `GALAHAD_LMS` for details.

`SHA_inform` is a scalar variable of type `SHA_inform_type` whose components give information about the progress and needs of the sparse model Hessian calculation performed by the package `GALAHAD_SHA`. See the specification sheet for the package `GALAHAD_SHA` for details.

2.3.6 The derived data type for holding problem data

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

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

2.3.7 The derived data type for holding user data

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

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

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

`complex` is a rank-one allocatable array of type default `COMPLEX` (double precision complex in `GALAHAD_TRB_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_TRB_double`).

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

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

2.4 Argument lists and calling sequences

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

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

We use square brackets [] to indicate OPTIONAL arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

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

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

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

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

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

2.4.2 The minimization subroutine

The minimization algorithm is called as follows:

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

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

`control` is a scalar `INTENT(IN)` argument of type `TRB_control_type` (see Section 2.3.3). Default values may be assigned by calling `TRB_initialize` prior to the first call to `TRB_solve`. The arguments `control%PSLS_control%preconditioner`, `control%PSLS_control%semi_bandwidth`, `control%PSLS_control%lbfgs_vectors` and `control%PSLS_control%icfs_vectors` will be overridden by `control%norm`, `control%semi_bandwidth`, `control%lbfgs_vectors` and `control%icfs_vectors`, respectively.

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

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

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

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

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

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

`eval_HPROD` is an `OPTIONAL` user-supplied subroutine whose purpose is to evaluate the value of the product $\nabla_{\mathbf{xx}} f(\mathbf{x})\mathbf{v}$ of the Hessian of the objective function $\nabla_{\mathbf{xx}} f(\mathbf{x})$ with a given vector $\mathbf{v}$. See Section 2.5.4 for details. If

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

`eval_HPROD` is present, it must be declared `EXTERNAL` in the calling program. If `eval_HPROD` is absent, `GALAHAD_TRB_solve` will use reverse communication to obtain Hessian-vector products (see Section 2.6).

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

`eval_PREC` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{P}(\mathbf{x})^{-1}\mathbf{v}$ of the inverse of the user's preconditioner with a given vector $\mathbf{v}$. See Section 2.5.6 for details. If `eval_PREC` is present, it must be declared `EXTERNAL` in the calling program. If `eval_PREC` is absent, `GALAHAD_TRB_solve` will use reverse communication to obtain products with the preconditioner (see Section 2.6).

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

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

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

2.5 Function and derivative values

2.5.1 The objective function value via internal evaluation

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

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

whose arguments are as follows:

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

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

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

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

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

2.5.2 Gradient values via internal evaluation

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

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

whose arguments are as follows:

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

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

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

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

2.5.3 Hessian values via internal evaluation

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

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

whose arguments are as follows:

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

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

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

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

2.5.4 Hessian-vector products via internal evaluation

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

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

whose arguments are as follows:

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

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

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

2.5.5 Hessian-sparse-vector products via internal evaluation

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

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

whose arguments are as follows:

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

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

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

2.5.6 Preconditioner-vector products via internal evaluation

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

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

whose arguments are as follows:

`status` is a scalar INTENT (OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the product $\mathbf{P}(\mathbf{x})^{-1}\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

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

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

`U` is a rank-one INTENT (OUT) array argument of type REAL(rp_) whose components on output should contain the product sum $\mathbf{u} = \mathbf{P}(\mathbf{x})^{-1}\mathbf{v}$.

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

2.6 Reverse Communication Information

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

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

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

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

5. The user should compute the product $\nabla_{xx}f(\mathbf{x})\mathbf{v}$ of the Hessian of the objective function $\nabla_{xx}f(\mathbf{x})$ at the point $\mathbf{x}$ indicated in `nlp%X` with the vector $\mathbf{v}$ and add the result to the vector $\mathbf{u}$. The vectors $\mathbf{u}$ and $\mathbf{v}$ are given in `data%U` and `data%V` respectively, the resulting vector $\mathbf{u} + \nabla_{xx}f(\mathbf{x})\mathbf{v}$ should be set in `data%U` and `data%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Hessian is undefined at $\mathbf{x}$ —the user need not set `nlp%H%val`, but should then set `data%eval_status` to a non-zero value.
6. The user should compute the product $\mathbf{u} = \mathbf{P}(\mathbf{x})^{-1}\mathbf{v}$ of the inverse of their preconditioner $\mathbf{P}(\mathbf{x})$ at the point $\mathbf{x}$ of their preconditioner $\mathbf{P}(\mathbf{x})$ at the point $\mathbf{x}$ indicated in `nlp%X` with the vector $\mathbf{v}$. The vectors $\mathbf{v}$ is given in `data%V`, the resulting vector $\mathbf{u} = \mathbf{P}(\mathbf{x})^{-1}\mathbf{v}$ should be set in `data%U` and `data%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the preconditioner is undefined at $\mathbf{x}$ —the user need not set `data%U`, but should then set `data%eval_status` to a non-zero value.

This value **can only occur** if the user has set `control%norm = -3`, and has not provided an optional subroutine `eval_PREC` (see §2.5.6) to compute the required product with the preconditioner.

7. The user should compute the product $\mathbf{h} = \nabla_{xx}f(\mathbf{x})\mathbf{p}$ of the Hessian of the objective function $\nabla_{xx}f(\mathbf{x})$ at the point $\mathbf{x}$ indicated in `nlp%X` with the sparse vector $\mathbf{p}$. The nonzeros of $\mathbf{p}$ are stored in `data%P (data%INDEX_nz_p (data%nnz_p_l:data%nnz_p_u))` while the nonzeros of $\mathbf{h}$ should be returned in `data%HP (data%INDEX_nz_hp (1:data%nnz_hp))`; the user must set `data%nnz_hp` and `data%INDEX_nz_hp` accordingly, and `data%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Hessian is undefined at $\mathbf{x}$ —the user need not set `data%HP`, `data%INDEX_nz_hp` and `data%nnz_hp` but should then set `data%eval_status` to a non-zero value.

This value **will not occur** if the user has set `control%hessian_available` to `.TRUE.` and can provide values of $\nabla_{xx}f(\mathbf{x})$ either by calls to `eval_H` (see §2.5.3) or by reverse communication (see `inform%status = 4`, above).

2.7 Warning and error messages

A negative value of `inform%status` on exit from `TRB_solve` or `TRB_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.

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

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

2.8 Further features

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

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

```
( .. lines ignored by TRB_read_specfile .. )
BEGIN TRB
    keyword    value
    .....
    keyword    value
END
( .. lines ignored by TRB_read_specfile .. )
```

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

and

```
END TRB SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN TRB" 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

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

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 `TRB_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 `TRB_read_specfile`.

2.8.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL TRB_read_specfile( control, device )
```

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

2.9 Information printed

If `control%print_level` is positive, information about the progress of the algorithm will be printed on unit `control%out`. If `control%print_level = 1`, a single line of output will be produced 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 ≥ 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%TRS_control%print_level`, `control%PSLS_control%print_level` and `control%GLTR_control%print_level`, while details about factorizations are available by increasing `control%PSLS_control%print_level`. See the specification sheets for the packages `GALAHAD_GLTR`, `GALAHAD_PSLs` and `GALAHAD_TRS` for details.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `TRB_solve` calls the `GALAHAD` packages `GALAHAD_CLOCK`, `GALAHAD_NLPT`, `GALAHAD_SYMBOLS`, `GALAHAD_SPECFILE`, `GALAHAD_SLS`, `GALAHAD_PSLs`, `GALAHAD_GLTR`, `GALAHAD_TRS`, `LANCELOT-CAUCHY`, `LANCELOT-CG`, `GALAHAD_LMS`, `GALAHAD_SHA`, `GALAHAD-MOP`, `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`.

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

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
iterations-between-printing	%print_gap	integer
maximum-number-of-iterations	%maxit	integer
alive-device	%alive_unit	integer
more-toraldo-search-length	%more_toraldo	integer
history-length-for-non-monotone-descent	%non_monotone	integer
model-used	%model	integer
norm-used	%norm	integer
semi-bandwidth-for-band-norm	%semi_bandwidth	integer
number-of-lbfgs-vectors	%lbfgs_vectors	integer
max-number-of-secant-vectors	%max_dgx	integer
number-of-lin-more-vectors	%icfs_vectors	integer
mi28-l-fill-size	%mi28_lsize	integer
mi28-r-entry-size	%mi28_rsize	integer
advanced-start	%advanced_start	integer
infinity-value	%infinity	real
absolute-gradient-accuracy-required	%stop_g_absolute	real
relative-gradient-reduction-required	%stop_g_relative	real
minimum-relative-step-allowed	%stop_s	real
initial-trust-region-radius	%initial_radius	real
maximum-trust-region-radius	%maximum_radius	real
inner-iteration-relative-accuracy-required	stop_rel_cg	real
successful-iteration-tolerance	%eta_successful	real
very-successful-iteration-tolerance	%eta_very_successful	real
too-successful-iteration-tolerance	%eta_too_successful	real
trust-region-increase-factor	%radius_increase	real
trust-region-decrease-factor	%radius_reduce	real
trust-region-maximum-decrease-factor	%radius_reduce_max	real
minimum-objective-before-unbounded	%obj_unbounded	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
hessian-available	%hessian_available	logical
sub-problem-direct	%subproblem_direct	logical
retrospective-trust-region	%retrospective_trust_region	logical
renormalize-radius	%renormalize_radius	logical
two-norm-trust-region-used	%two_norm_tr	logical
exact-GCP-used	%exact_gcp	logical
subproblem-solved-accurately	%accurate_bqp	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.

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

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

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

4 METHOD

A trust-region method is used. In this, an improvement to a current estimate of the required minimizer, $\mathbf{x}_k$ is sought by computing a step $\mathbf{s}_k$. The step is chosen to approximately minimize a model $m_k(\mathbf{s})$ of $f(\mathbf{x}_k + \mathbf{s})$ within the intersection of the bound constraints $\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u$ and a trust region $\|\mathbf{s}_k\| \leq \Delta_k$ for some specified positive "radius" Δ_k . The quality of the resulting step $\mathbf{s}_k$ is assessed by computing the "ratio" $(f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{s}_k)) / (m_k(\mathbf{0}) - m_k(\mathbf{s}_k))$. The step is deemed to have succeeded if the ratio exceeds a given $\eta_s > 0$, and in this case $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$. Otherwise $\mathbf{x}_{k+1} = \mathbf{x}_k$, and the radius is reduced by powers of a given reduction factor until it is smaller than $\|\mathbf{s}_k\|$. If the ratio is larger than $\eta_v \geq \eta_d$, the radius will be increased so that it exceeds $\|\mathbf{s}_k\|$ by a given increase factor. The method will terminate as soon as $\|\nabla_x f(\mathbf{x}_k)\|$ is smaller than a specified value.

Either linear or quadratic models $m_k(\mathbf{s})$ may be used. The former will be taken as the first two terms $f(\mathbf{x}_k) + \mathbf{s}^T \nabla_x f(\mathbf{x}_k)$ of a Taylor series about $\mathbf{x}_k$, while the latter uses an approximation to the first three terms $f(\mathbf{x}_k) + \mathbf{s}^T \nabla_x f(\mathbf{x}_k) + \frac{1}{2} \mathbf{s}^T \mathbf{B}_k \mathbf{s}$, for which $\mathbf{B}_k$ is a symmetric approximation to the Hessian $\nabla_{xx} f(\mathbf{x}_k)$; possible approximations include the true Hessian, limited-memory secant and sparsity approximations and a scaled identity matrix. Normally a two-norm trust region will be used, but this may change if preconditioning is employed.

The model minimization is carried out in two stages. Firstly, the so-called generalized Cauchy point for the quadratic subproblem is found—the purpose of this point is to ensure that the algorithm converges and that the set of bounds which are satisfied as equations at the solution is rapidly identified. Thereafter an improvement to the quadratic model on the face of variables predicted to be active by the Cauchy point is sought using either a direct approach involving factorization or an iterative (conjugate-gradient/Lanczos) approach based on approximations to the required solution from a so-called Krylov subspace. The direct approach is based on the knowledge that the required solution satisfies the linear system of equations $(\mathbf{B}_k + \lambda_k \mathbf{I}) \mathbf{s}_k = -\nabla_x f(\mathbf{x}_k)$, involving a scalar Lagrange multiplier λ_k , on the space of inactive variables. This multiplier is found by uni-variate root finding, using a safeguarded Newton-like process, by GALAHAD_TRS or GALAHAD_DPS (depending on the norm chosen). The iterative approach uses GALAHAD_GLTR, and is best accelerated by preconditioning with good approximations to $\mathbf{B}_k$ using GALAHAD_PSLs. The iterative approach has the advantage that only matrix-vector products $\mathbf{B}_k \mathbf{v}$ are required, and thus $\mathbf{B}_k$ is not required explicitly. However when factorizations of $\mathbf{B}_k$ are possible, the direct approach is often more efficient.

The iteration is terminated as soon as the Euclidean norm of the projected gradient,

$$\|\min(\max(\mathbf{x}_k - \nabla_x f(\mathbf{x}_k), \mathbf{x}^l), \mathbf{x}^u) - \mathbf{x}_k\|_2,$$

is sufficiently small. At such a point, $\nabla_x f(\mathbf{x}_k) = \mathbf{z}_k$, where the i -th dual variable z_i is non-negative if x_i is on its lower bound x_i^l , non-positive if x_i is on its upper bound x_i^u , and zero if x_i lies strictly between its bounds.

References:

The generic bound-constrained trust-region method is described in detail in

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

5 EXAMPLES OF USE

Suppose we wish to minimize the parametric objective function $f(\mathbf{x}) = (x_1 + x_3 + p)^2 + (x_2 + x_3)^2 + \cos x_1$ when the parameter p takes the value 4, and $\mathbf{x}$ is required to satisfy the bounds $x_1 \leq -1.1$, $x_2 \leq -1.1$ and $0 \leq x_3 \leq -1.1$. Starting from the initial guess $\mathbf{x} = (1, 1, 1)$, we may use the following code:

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

```

PROGRAM GALAHAD_TRB_EXAMPLE ! GALAHAD 4.1 - 2022-12-29 AT 11:15 GMT
USE GALAHAD_TRB_double      ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( TRB_control_type ) :: control
TYPE ( TRB_inform_type ) :: inform
TYPE ( TRB_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
EXTERNAL :: FUN, GRAD, HESS
INTEGER :: s
INTEGER, PARAMETER :: n = 3, h_ne = 5
REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp
REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20 ! infinity
! start problem data
nlp%pname = 'TRBSPEC' ! name
nlp%n = n ; nlp%H%ne = h_ne ! dimensions
ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
nlp%X = 1.0_wp ! start from one
nlp%X_l( : n ) = (/ - infinity, - infinity, 0.0_wp /) ; nlp%X_u = 1.1_wp
! sparse co-ordinate storage format
CALL SMT_put( nlp%H%type, 'COORDINATE', s ) ! Specify co-ordinate storage
ALLOCATE( nlp%H%val( h_ne ), nlp%H%row( h_ne ), nlp%H%col( h_ne ) )
nlp%H%row = (/ 1, 3, 2, 3, 3 /) ! Hessian H
nlp%H%col = (/ 1, 1, 2, 2, 3 /) ! NB lower triangle
! problem data complete
ALLOCATE( userdata%real( 1 ) ) ! Allocate space for parameter
userdata%real( 1 ) = p ! Record parameter, p
CALL TRB_initialize( data, control, inform ) ! Initialize control parameters
control%subproblem_direct = .FALSE. ! Use an iterative method
control%maxit = 10
! control%print_level = 1
inform%status = 1 ! set for initial entry
CALL TRB_solve( nlp, control, inform, data, userdata, eval_F = FUN, &
               eval_G = GRAD, eval_H = HESS ) ! Solve problem
IF ( inform%status == 0 ) THEN ! Successful return
  WRITE( 6, "( ' TRB: ', I0, ' iterations -', &
    & ' optimal objective value =', &
    & ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" ) &
  inform%iter, inform%obj, nlp%X
ELSE ! Error returns
  WRITE( 6, "( ' TRB_solve exit status = ', I6 ) " ) inform%status
END IF
CALL TRB_terminate( data, control, inform ) ! delete internal workspace
DEALLOCATE( nlp%X, nlp%G, nlp%H%val, nlp%H%row, nlp%H%col, userdata%real )
END PROGRAM GALAHAD_TRB_EXAMPLE

SUBROUTINE FUN( status, X, userdata, f ) ! Objective function
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), INTENT( OUT ) :: f
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
f = ( X( 1 ) + X( 3 ) + userdata%real( 1 ) ) ** 2 + &

```

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

```

      ( X( 2 ) + X( 3 ) ) ** 2 + COS( X( 1 ) )
status = 0
RETURN
END SUBROUTINE FUN

SUBROUTINE GRAD( status, X, userdata, G )      ! gradient of the objective
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: G
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
G( 1 ) = 2.0_wp * ( X( 1 ) + X( 3 ) + userdata%real( 1 ) ) - SIN( X( 1 ) )
G( 2 ) = 2.0_wp * ( X( 2 ) + X( 3 ) )
G( 3 ) = 2.0_wp * ( X( 1 ) + X( 3 ) + userdata%real( 1 ) ) +      &
      2.0_wp * ( X( 2 ) + X( 3 ) )
status = 0
RETURN
END SUBROUTINE GRAD

SUBROUTINE HESS( status, X, userdata, Hval ) ! Hessian of the objective
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: Hval
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
Hval( 1 ) = 2.0_wp - COS( X( 1 ) )
Hval( 2 ) = 2.0_wp
Hval( 3 ) = 2.0_wp
Hval( 4 ) = 2.0_wp
Hval( 5 ) = 4.0_wp
status = 0
RETURN
END SUBROUTINE HESS

```

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

```

forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image                PC                               Routine           Line      Source
libc.so.6             00000C0606E45330 Unknown           Unknown     Unknown
run_trb               0000000000406098 Unknown           Unknown     Unknown
libc.so.6             00000C0606E45330 Unknown           Unknown     Unknown

```

If the Hessian is unavailable, but products of the form $\mathbf{u} + \mathbf{H}\mathbf{v}$ are, the same problem may be solved as follows:

```

PROGRAM GALAHAD_TRB2_EXAMPLE ! GALAHAD 4.1 - 2022-12-29 AT 11:15 GMT
USE GALAHAD_TRB_double      ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( TRB_control_type ) :: control
TYPE ( TRB_inform_type ) :: inform
TYPE ( TRB_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
EXTERNAL :: FUN, GRAD, HESSPROD, SHESSPROD

```

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

```

INTEGER, PARAMETER :: n = 3, h_ne = 5
REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp
REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20      ! infinity
! start problem data
nlp%n = n ; nlp%H%ne = h_ne                                ! dimensions
ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
nlp%X = 1.0_wp                                              ! start from one
nlp%X_l( : n ) = (/ - infinity, - infinity, 0.0_wp /) ; nlp%X_u = 1.1_wp
! problem data complete
ALLOCATE( userdata%real( 1 ) )                            ! Allocate space for parameter
userdata%real( 1 ) = p                                     ! Record parameter, p
CALL TRB_initialize( data, control, inform ) ! Initialize control parameters
control%hessian_available = .FALSE.                      ! Hessian products will be used
! control%print_level = 1
inform%status = 1                                           ! Set for initial entry
CALL TRB_solve( nlp, control, inform, data, userdata, eval_F = FUN,      &
               eval_G = GRAD, eval_HPROD = HESSPROD,                  &
               eval_SHPROD = SHESSPROD ) ! Solve problem
IF ( inform%status == 0 ) THEN                                ! Successful return
  WRITE( 6, "( ' TRB: ', I0, ' iterations -',                      &
    &      ' optimal objective value =',                          &
    &      ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )      &
  inform%iter, inform%obj, nlp%X
ELSE
  ! Error returns
  WRITE( 6, "( ' TRB_solve exit status = ', I6 ) " ) inform%status
END IF
CALL TRB_terminate( data, control, inform ) ! delete internal workspace
DEALLOCATE( nlp%X, nlp%G, userdata%real )
END PROGRAM GALAHAD_TRB2_EXAMPLE

SUBROUTINE FUN( status, X, userdata, f ) ! Objective function
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), INTENT( OUT ) :: f
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
f = ( X( 1 ) + X( 3 ) + userdata%real( 1 ) ) ** 2 +          &
    ( X( 2 ) + X( 3 ) ) ** 2 + COS( X( 1 ) )
status = 0
RETURN
END SUBROUTINE FUN

SUBROUTINE GRAD( status, X, userdata, G ) ! gradient of the objective
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: G
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
G( 1 ) = 2.0_wp * ( X( 1 ) + X( 3 ) + userdata%real( 1 ) ) - SIN( X( 1 ) )
G( 2 ) = 2.0_wp * ( X( 2 ) + X( 3 ) )
G( 3 ) = 2.0_wp * ( X( 1 ) + X( 3 ) + userdata%real( 1 ) ) +  &
    2.0_wp * ( X( 2 ) + X( 3 ) )
status = 0

```

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

```
RETURN
END SUBROUTINE GRAD

SUBROUTINE HESSPROD( status, X, userdata, U, V, got_h ) ! Hess-vector product
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( INOUT ) :: U
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X, V
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
LOGICAL, OPTIONAL, INTENT( IN ) :: got_h
U( 1 ) = U( 1 ) + 2.0_wp * ( V( 1 ) + V( 3 ) ) - COS( X( 1 ) ) * V( 1 )
U( 2 ) = U( 2 ) + 2.0_wp * ( V( 2 ) + V( 3 ) )
U( 3 ) = U( 3 ) + 2.0_wp * ( V( 1 ) + V( 2 ) + 2.0_wp * V( 3 ) )
status = 0
RETURN
END SUBROUTINE HESSPROD

SUBROUTINE SHESSPROD( status, X, userdata, nnz_v, INDEX_nz_v, V,          &
                     nnz_u, INDEX_nz_u, U, got_h )
USE GALAHAD_USERDATA_double, ONLY: USERDATA_type
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( IN ) :: nnz_v
INTEGER, INTENT( OUT ) :: nnz_u
INTEGER, INTENT( OUT ) :: status
INTEGER, DIMENSION( : ), INTENT( IN ) :: INDEX_nz_v
INTEGER, DIMENSION( : ), INTENT( OUT ) :: INDEX_nz_u
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: U
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
LOGICAL, OPTIONAL, INTENT( IN ) :: got_h
INTEGER :: i, j
REAL ( KIND = wp ), DIMENSION( 3 ) :: P
LOGICAL, DIMENSION( 3 ) :: USED
P = 0.0_wp
USED = .FALSE.
DO i = 1, nnz_v
  j = INDEX_nz_v( i )
  SELECT CASE( j )
    CASE( 1 )
      P( 1 ) = P( 1 ) + 2.0_wp * V( 1 ) - COS( X( 1 ) ) * V( 1 )
      USED( 1 ) = .TRUE.
      P( 3 ) = P( 3 ) + 2.0_wp * V( 1 )
      USED( 3 ) = .TRUE.
    CASE( 2 )
      P( 2 ) = P( 2 ) + 2.0_wp * V( 2 )
      USED( 2 ) = .TRUE.
      P( 3 ) = P( 3 ) + 2.0_wp * V( 2 )
      USED( 3 ) = .TRUE.
    CASE( 3 )
      P( 1 ) = P( 1 ) + 2.0_wp * V( 3 )
      USED( 1 ) = .TRUE.
      P( 2 ) = P( 2 ) + 2.0_wp * V( 3 )
      USED( 2 ) = .TRUE.
```

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

```

      P( 3 ) = P( 3 ) + 4.0_wp * V( 3 )
      USED( 3 ) = .TRUE.
    END SELECT
  END DO
  nnz_u = 0
  DO j = 1, 3
    IF ( USED( j ) ) THEN
      U( j ) = P( j )
      nnz_u = nnz_u + 1
      INDEX_nz_u( nnz_u ) = j
    END IF
  END DO
  status = 0
  RETURN
END SUBROUTINE SNESSPROD

```

Notice that storage for the Hessian is now not needed. This produces the same output.

If the user prefers to provide function and gradient information and Hessian-vector products without calls to specified routines, the following code is appropriate. Note the product with the user-provided preconditioner

$$\mathbf{P}(\mathbf{x}) = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{4} \end{pmatrix}$$

which is a suitable approximation to the inverse of the Hessian:

```

PROGRAM GALAHAD_TRB3_EXAMPLE ! GALAHAD 4.1 - 2022-12-29 AT 11:15 GMT
USE GALAHAD_TRB_double      ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( TRB_control_type ) :: control
TYPE ( TRB_inform_type ) :: inform
TYPE ( TRB_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
INTEGER :: s
INTEGER, PARAMETER :: n = 3, h_ne = 5
REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp
REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20 ! infinity
! start problem data
nlp%n = n ; nlp%H%ne = h_ne ! dimensions
ALLOCATE( nlp%X( n ), nlp%G( n ), nlp%X_l( n ), nlp%X_u( n ) )
nlp%X = 1.0_wp ! start from one
nlp%X_l( : n ) = (/ - infinity, - infinity, 0.0_wp /) ; nlp%X_u = 1.1_wp
! sparse co-ordinate storage format
CALL SMT_put( nlp%H%type, 'COORDINATE', s ) ! Specify co-ordinate storage
ALLOCATE( nlp%H%val( h_ne ), nlp%H%row( h_ne ), nlp%H%col( h_ne ) )
nlp%H%row = (/ 1, 3, 2, 3, 3 /) ! Hessian H
nlp%H%col = (/ 1, 1, 2, 2, 3 /) ! NB lower triangle
! problem data complete
CALL TRB_initialize( data, control, inform ) ! Initialize control parameters
! control%hessian_available = .FALSE. ! Hessian products will be used
! control%psls_control%preconditioner = - 3 ! Apply user's preconditioner
inform%status = 1 ! Set for initial entry
DO ! Loop to solve problem
  CALL TRB_solve( nlp, control, inform, data, userdata )

```

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

```

SELECT CASE ( inform%status )           ! reverse communication
CASE ( 2 )                             ! Obtain the objective function
  nlp%f = ( nlp%X( 1 ) + nlp%X( 3 ) + p ) ** 2 + &
           ( nlp%X( 2 ) + nlp%X( 3 ) ) ** 2 + COS( nlp%X( 1 ) )
  data%eval_status = 0                  ! record successful evaluation
CASE ( 3 )                             ! Obtain the gradient
  nlp%G( 1 ) = 2.0_wp * ( nlp%X( 1 ) + nlp%X( 3 ) + p ) - SIN( nlp%X( 1 ) )
  nlp%G( 2 ) = 2.0_wp * ( nlp%X( 2 ) + nlp%X( 3 ) )
  nlp%G( 3 ) = 2.0_wp * ( nlp%X( 1 ) + nlp%X( 3 ) + p ) + &
               2.0_wp * ( nlp%X( 2 ) + nlp%X( 3 ) )
  data%eval_status = 0                  ! record successful evaluation
CASE ( 4 )                             ! Obtain Hessian evaluation
  nlp%H%val( 1 ) = 2.0_wp - COS( nlp%X( 1 ) )
  nlp%H%val( 2 ) = 2.0_wp
  nlp%H%val( 3 ) = 2.0_wp
  nlp%H%val( 4 ) = 2.0_wp
  nlp%H%val( 5 ) = 4.0_wp
  data%eval_status = 0                  ! record successful evaluation
CASE ( 5 )                             ! Obtain Hessian-vector product
  data%U( 1 ) = data%U( 1 ) + 2.0_wp * ( data%V( 1 ) + data%V( 3 ) ) - &
               COS( nlp%X( 1 ) ) * data%V( 1 )
  data%U( 2 ) = data%U( 2 ) + 2.0_wp * ( data%V( 2 ) + data%V( 3 ) )
  data%U( 3 ) = data%U( 3 ) + 2.0_wp * ( data%V( 1 ) + data%V( 2 ) + &
               2.0_wp * data%V( 3 ) )
  data%eval_status = 0                  ! record successful evaluation
CASE ( 6 )                             ! Apply the preconditioner
  data%U( 1 ) = 0.5_wp * data%V( 1 )
  data%U( 2 ) = 0.5_wp * data%V( 2 )
  data%U( 3 ) = 0.25_wp * data%V( 3 )
  data%eval_status = 0                  ! record successful evaluation
CASE DEFAULT                           ! Terminal exit from loop
  EXIT
END SELECT
END DO
IF ( inform%status == 0 ) THEN          ! Successful return
  WRITE( 6, "( ' TRB: ', I0, ' iterations -', &
&      ' optimal objective value =', &
&      ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" ) &
  inform%iter, inform%obj, nlp%X
ELSE                                   ! Error returns
  WRITE( 6, "( ' TRB_solve exit status = ', I6 ) " ) inform%status
END IF
CALL TRB_terminate( data, control, inform ) ! Delete internal workspace
END PROGRAM GALAHAD_TRB3_EXAMPLE

```

This produces the following output:

```

TRB: 5 iterations - optimal objective value = -7.5897E-01
Optimal solution = -3.7247E+00  0.0000E+00  0.0000E+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.