C interfaces to GALAHAD TRB

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory

Sun Apr 2 2023

Chapter 1

GALAHAD C package trb

1.1 Introduction

1.1.1 Purpose

The trb package uses a **trust-region method to find a (local) minimizer of a differentiable objective function $\mathbf{f(x)}$ of many variables $\mathbf{x}$, where the variables satisfy the simple bounds $\mathbf{x^l \leq x \leq x^u}$.** The method offers the choice of direct and iterative solution of the key 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.

1.1.2 Authors

N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England.

C interface, additionally J. Fowkes, STFC-Rutherford Appleton Laboratory.

Julia interface, additionally A. Montoison and D. Orban, Polytechnique Montréal.

1.1.3 Originally released

July 2021, C interface August 2021.

1.1.4 Terminology

The *gradient* $\nabla_x f(x)$ of $f(x)$ is the vector whose i-th component is $\partial f(x)/\partial x_i$. The *Hessian* $\nabla_{xx} f(x)$ of $f(x)$ is the symmetric matrix whose i,j-th entry is $\partial^2 f(x)/\partial x_i \partial x_j$. The Hessian is *sparse* if a significant and useful proportion of the entries are universally zero.

1.1.5 Method

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

Either linear or quadratic models $m_k(s)$ may be used. The former will be taken as the first two terms $f(x_k) + s^T \nabla_x f(x_k)$ of a Taylor series about x_k, while the latter uses an approximation to the first three terms $f(x_k) + s^T \nabla_x f(x_k) + \frac{1}{2} s^T B_k s$, for which B_k is a symmetric approximation to the Hessian $\nabla_{xx} f(x_k)$; 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 Krlov subspace. The direct approach is based on the knowledge that the required solution satisfies the linear system of equations $(B_k + \lambda_k I) s_k = -\nabla_x f(x_k)$, 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 the GALAHAD package TRS. The iterative approach uses GALAHAD package GLTR, and is best accelerated by preconditioning with good approximations to B_k using GALAHAD's PSLS. The iterative approach has the advantage that only matrix-vector products $B_k v$ are required, and thus B_k is not required explicitly. However when factorizations of 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,

$$\| \min(\max(x_k - \nabla_x f(x_k), x^l), x^u) - x_k \|_2,$$

is sufficiently small. At such a point, $\nabla_x f(x_k) = z_k$, 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.

1.1.6 References

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

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

1.2 Call order

To solve a given problem, functions from the trb package must be called in the following order:

- trb_initialize - provide default control parameters and set up initial data structures
- trb_read_specfile (optional) - override control values by reading replacement values from a file
- trb_import - set up problem data structures and fixed values
- trb_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved

- solve the problem by calling one of
  - trb_solve_with_mat - solve using function calls to evaluate function, gradient and Hessian values
  - trb_solve_without_mat - solve using function calls to evaluate function and gradient values and Hessian-vector products
  - trb_solve_reverse_with_mat - solve returning to the calling program to obtain function, gradient and Hessian values, or
  - trb_solve_reverse_without_mat - solve returning to the calling prorgram to obtain function and gradient values and Hessian-vector products
- trb_information (optional) - recover information about the solution and solution process
- trb_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.3 Symmetric matrix storage formats

The symmetric n by n matrix $H = \nabla_{xx} f$ may be presented and stored in a variety of formats. But crucially symmetry is exploited by only storing values from the lower triangular part (i.e, those entries that lie on or below the leading diagonal).

Both C-style (0 based) and fortran-style (1-based) indexing is allowed. Choose `control.f_indexing` as `false` for C style and `true` for fortran style; the discussion below presumes C style, but add 1 to indices for the corresponding fortran version.

Wrappers will automatically convert between 0-based (C) and 1-based (fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing.

1.3.1 Dense storage format

The matrix 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 H is symmetric, only the lower triangular part (that is the part H_{ij} for $0 \leq j \leq i \leq n-1$) need be held. In this case the lower triangle should be stored by rows, that is component $i * i/2 + j$ of the storage array H_val will hold the value H_{ij} (and, by symmetry, H_{ji}) for $0 \leq j \leq i \leq n-1$.

1.3.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne-1$, of H, its row index i, column index j and value H_{ij}, $0 \leq j \leq i \leq n-1$, are stored as the l-th components of the integer arrays H_row and H_col and real array H_val, respectively, while the number of nonzeros is recorded as H_ne = ne. Note that only the entries in the lower triangle should be stored.

1.3.3 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 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) holds the total number of entries. The column indices j, $0 \leq j \leq i$, and values H_{ij} of the entries in the i-th row are stored in components l = H_ptr(i), . . ., 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.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 galahad_trb.h File Reference

```
#include <stdbool.h>
#include <stdint.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_trs.h"
#include "galahad_gltr.h"
#include "galahad_psls.h"
#include "galahad_lms.h"
#include "galahad_sha.h"
```

Data Structures

- struct trb_control_type
- struct trb_time_type
- struct trb_inform_type

Functions

- void trb_initialize (void ∗∗data, struct trb_control_type ∗control, int ∗status)
- void trb_read_specfile (struct trb_control_type ∗control, const char specfile[ ])
- void trb_import (struct trb_control_type ∗control, void ∗∗data, int ∗status, int n, const real_wp_ x_l[ ], const real_wp_ x_u[ ], const char H_type[ ], int ne, const int H_row[ ], const int H_col[ ], const int H_ptr[ ])
- void trb_reset_control (struct trb_control_type ∗control, void ∗∗data, int ∗status)
- void trb_solve_with_mat (void ∗∗data, void ∗userdata, int ∗status, int n, real_wp_ x[ ], real_wp_ g[ ], int ne, int(∗eval_f)(int, const real_wp_[ ], real_wp_ ∗, const void ∗), int(∗eval_g)(int, const real_wp_[ ], real_wp_↩ [ ], const void ∗), int(∗eval_h)(int, int, const real_wp_[ ], real_wp_[ ], const void ∗), int(∗eval_prec)(int, const real_wp_[ ], real_wp_[ ], const real_wp_[ ], const void ∗))
- void trb_solve_without_mat (void ∗∗data, void ∗userdata, int ∗status, int n, real_wp_ x[ ], real_wp_ g[ ], int(∗eval_f)(int, const real_wp_[ ], real_wp_ ∗, const void ∗), int(∗eval_g)(int, const real_wp_[ ], real_wp_↩ [ ], const void ∗), int(∗eval_hprod)(int, const real_wp_[ ], real_wp_[ ], const real_wp_[ ], bool, const void ∗), int(∗eval_shprod)(int, const real_wp_[ ], int, const int[ ], const real_wp_[ ], int ∗, int[ ], real_wp_[ ], bool, const void ∗), int(∗eval_prec)(int, const real_wp_[ ], real_wp_[ ], const real_wp_[ ], const void ∗))
- void trb_solve_reverse_with_mat (void ∗∗data, int ∗status, int ∗eval_status, int n, real_wp_ x[ ], real_wp_ f, real_wp_ g[ ], int ne, real_wp_ H_val[ ], const real_wp_ u[ ], real_wp_ v[ ])
- void trb_solve_reverse_without_mat (void ∗∗data, int ∗status, int ∗eval_status, int n, real_wp_ x[ ], real_wp_ f, real_wp_ g[ ], real_wp_ u[ ], real_wp_ v[ ], int index_nz_v[ ], int ∗nnz_v, const int index_nz_u[ ], int nnz_u)
- void trb_information (void ∗∗data, struct trb_inform_type ∗inform, int ∗status)
- void trb_terminate (void ∗∗data, struct trb_control_type ∗control, struct trb_inform_type ∗inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct trb_control_type

control derived type as a C struct

Examples

trbt.c, and trbtf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	error	error and warning diagnostics occur on stream error
int	out	general output occurs on stream out
int	print_level	the level of output required. • ≤ 0 gives no output, • = 1 gives a one-line summary for every iteration, • = 2 gives a summary of the inner iteration for each iteration, • ≥ 3 gives increasingly verbose (debugging) output
int	start_print	any printing will start on this iteration
int	stop_print	any printing will stop on this iteration
int	print_gap	the number of iterations between printing
int	maxit	the maximum number of iterations performed
int	alive_unit	removal of the file alive_file from unit alive_unit terminates execution
char	alive_file[31]	see alive_unit
int	more_toraldo	more_toraldo >= 1 gives the number of More'-Toraldo projected searches to be used to improve upon the Cauchy point, anything else is for the standard add-one-at-a-time CG search
int	non_monotone	non-monotone <= 0 monotone strategy used, anything else non-monotone strategy with this history length used
int	model	the model used. Possible values are • 0 dynamic (*not yet implemented*) • 1 first-order (no Hessian) • 2 second-order (exact Hessian) • 3 barely second-order (identity Hessian) • 4 secant second-order (sparsity-based) • 5 secant second-order (limited-memory BFGS, with .lbfgs_vectors history) (*not yet implemented*) • 6 secant second-order (limited-memory SR1, with .lbfgs_vectors history) (*not yet implemented*)

Data Fields

int	norm	The norm is defined via $\|v\|^2 = v^T P v$, and will define the preconditioner used for iterative methods. Possible values for P are. • -3 users own preconditioner • -2 $P =$ limited-memory BFGS matrix (with .lbfgs_vectors history) • -1 identity (= Euclidan two-norm) • 0 automatic (*not yet implemented*) • 1 diagonal, $P =$ diag(max(Hessian, .min_diagonal)) • 2 banded, $P =$ band(Hessian) with semi-bandwidth .semi_bandwidth • 3 re-ordered band, P=band(order(A)) with semi-bandwidth .semi_bandwidth • 4 full factorization, $P =$ Hessian, Schnabel-Eskow modification • 5 full factorization, $P =$ Hessian, GMPS modification (*not yet implemented*) • 6 incomplete factorization of Hessian, Lin-More' • 7 incomplete factorization of Hessian, HSL_MI28 • 8 incomplete factorization of Hessian, Munskgaard (*not yet implemented*) • 9 expanding band of Hessian (*not yet implemented*)
int	semi_bandwidth	specify the semi-bandwidth of the band matrix P if required
int	lbfgs_vectors	number of vectors used by the L-BFGS matrix P if required
int	max_dxg	number of vectors used by the sparsity-based secant Hessian if required
int	icfs_vectors	number of vectors used by the Lin-More' incomplete factorization matrix P if required
int	mi28_lsize	the maximum number of fill entries within each column of the incomplete factor L computed by HSL_MI28. In general, increasing .mi28_lsize improve the quality of the preconditioner but increases the time to compute and then apply the preconditioner. Values less than 0 are treated as 0

Data Fields

int	mi28_rsize	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
int	advanced_start	iterates of a variant on the strategy of Sartenaer SISC 18(6)1990:1788-1803
real_wp_	infinity	any bound larger than infinity in modulus will be regarded as infinite
real_wp_	stop_pg_absolute	overall convergence tolerances. The iteration will terminate when the norm of the gradient of the objective function is smaller than MAX(.stop_pg_absolute, .stop_pg_relative $*$ norm of the initial gradient) or if the step is less than .stop_s
real_wp_	stop_pg_relative	see stop_pg_absolute
real_wp_	stop_s	see stop_pg_absolute
real_wp_	initial_radius	initial value for the trust-region radius
real_wp_	maximum_radius	maximum permitted trust-region radius
real_wp_	stop_rel_cg	required relative reduction in the resuiduals from CG
real_wp_	eta_successful	a potential iterate will only be accepted if the actual decrease f - f(x_new) is larger than .eta_successful times that predicted by a quadratic model of the decrease. The trust-region radius will be increased if this relative decrease is greater than .eta_very_successful but smaller than .eta_too_successful
real_wp_	eta_very_successful	see eta_successful
real_wp_	eta_too_successful	see eta_successful
real_wp_	radius_increase	on very successful iterations, the trust-region radius will be increased the factor .radius_increase, while if the iteration is unsucceful, the radius will be decreased by a factor .radius_reduce but no more than .radius_reduce_max
real_wp_	radius_reduce	see radius_increase
real_wp_	radius_reduce_max	see radius_increase
real_wp_	obj_unbounded	the smallest value the objective function may take before the problem is marked as unbounded
real_wp_	cpu_time_limit	the maximum CPU time allowed (-ve means infinite)
real_wp_	clock_time_limit	the maximum elapsed clock time allowed (-ve means infinite)
bool	hessian_available	is the Hessian matrix of second derivatives available or is access only via matrix-vector products?
bool	subproblem_direct	use a direct (factorization) or (preconditioned) iterative method to find the search direction
bool	retrospective_trust_region	is a retrospective strategy to be used to update the trust-region radius

Data Fields

bool	renormalize_radius	should the radius be renormalized to account for a change in preconditioner?
bool	two_norm_tr	should an ellipsoidal trust-region be used rather than an infinity norm one?
bool	exact_gcp	is the exact Cauchy point required rather than an approximation?
bool	accurate_bqp	should the minimizer of the quadratic model within the intersection of the trust-region and feasible box be found (to a prescribed accuracy) rather than a (much) cheaper approximation?
bool	space_critical	if .space_critical true, every effort will be made to use as little space as possible. This may result in longer computation time
bool	deallocate_error_fatal	if .deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue
char	prefix[31]	all output lines will be prefixed by .prefix(2:LEN(TRIM(.prefix))-1) where .prefix contains the required string enclosed in quotes, e.g. "string" or 'string'
struct trs_control_type	trs_control	control parameters for TRS
struct gltr_control_type	gltr_control	control parameters for GLTR
struct psls_control_type	psls_control	control parameters for PSLS
struct lms_control_type	lms_control	control parameters for LMS
struct lms_control_type	lms_control_prec	control parameters for LMS used for preconditioning
struct sha_control_type	sha_control	control parameters for SHA

3.1.1.2 struct trb_time_type

time derived type as a C struct

Data Fields

real_sp_	total	the total CPU time spent in the package
real_sp_	preprocess	the CPU time spent preprocessing the problem
real_sp_	analyse	the CPU time spent analysing the required matrices prior to factorization
real_sp_	factorize	the CPU time spent factorizing the required matrices
real_sp_	solve	the CPU time spent computing the search direction
real_wp_	clock_total	the total clock time spent in the package
real_wp_	clock_preprocess	the clock time spent preprocessing the problem
real_wp_	clock_analyse	the clock time spent analysing the required matrices prior to factorization
real_wp_	clock_factorize	the clock time spent factorizing the required matrices
real_wp_	clock_solve	the clock time spent computing the search direction

3.1.1.3 struct trb_inform_type

inform derived type as a C struct

Examples

trbt.c, and trbtf.c.

Data Fields

int	status	return status. See TRB_solve for details
int	alloc_status	the status of the last attempted allocation/deallocation
char	bad_alloc[81]	the name of the array for which an allocation/deallocation error ocurred
int	iter	the total number of iterations performed
int	cg_iter	the total number of CG iterations performed
int	cg_maxit	the maximum number of CG iterations allowed per iteration
int	f_eval	the total number of evaluations of the objective function
int	g_eval	the total number of evaluations of the gradient of the objective function
int	h_eval	the total number of evaluations of the Hessian of the objective function
int	n_free	the number of variables that are free from their bounds
int	factorization_max	the maximum number of factorizations in a sub-problem solve
int	factorization_status	the return status from the factorization
int64_t	max_entries_factors	the maximum number of entries in the factors
int64_t	factorization_integer	the total integer workspace required for the factorization
int64_t	factorization_real	the total real workspace required for the factorization
real_wp_	obj	the value of the objective function at the best estimate of the solution determined by TRB_solve
real_wp_	norm_pg	the norm of the projected gradient of the objective function at the best estimate of the solution determined by TRB_solve
real_wp_	radius	the current value of the trust-region radius
struct trb_time_type	time	timings (see above)
struct trs_inform_type	trs_inform	inform parameters for TRS
struct gltr_inform_type	gltr_inform	inform parameters for GLTR
struct psls_inform_type	psls_inform	inform parameters for PSLS
struct lms_inform_type	lms_inform	inform parameters for LMS
struct lms_inform_type	lms_inform_prec	inform parameters for LMS used for preconditioning
struct sha_inform_type	sha_inform	inform parameters for SHA

3.1.2 Function Documentation

3.1.2.1 trb_initialize()

```
void trb_initialize (
            void ** data,
            struct trb_control_type * control,
            int * status )
```

Set default control values and initialize private data

Parameters

in,out	*data*	holds private internal data
out	*control*	is a struct containing control information (see trb_control_type)
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): • 0. The import was succesful.

Examples

trbt.c, and trbtf.c.

3.1.2.2 trb_read_specfile()

```
void trb_read_specfile (
            struct trb_control_type * control,
            const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters. By default, the spcification file will be named RUNTRB.SPC and lie in the current directory. Refer to Table 2.1 in the fortran documentation provided in $GALAHAD/doc/trb.pdf for a list of keywords that may be set.

Parameters

in,out	*control*	is a struct containing control information (see trb_control_type)
in	*specfile*	is a character string containing the name of the specification file

3.1.2.3 trb_import()

```
void trb_import (
            struct trb_control_type * control,
            void ** data,
            int * status,
            int n,
            const real_wp_ x_l[],
            const real_wp_ x_u[],
            const char H_type[],
            int ne,
            const int H_row[],
            const int H_col[],
            const int H_ptr[] )
```

Import problem data into internal storage prior to solution.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see trb_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 1. The import was succesful, and the package is ready for the solve phase • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the values x^l of the lower bounds on the optimization variables x. The j-th component of x_l, $j = 0, \ldots, n-1$, contains x^l_j.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the values x^u of the upper bounds on the optimization variables x. The j-th component of x_u, $j = 0, \ldots, n-1$, contains x^u_j.
in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal' or 'absent', the latter if access to the Hessian is via matrix-vector products; lower or upper case variants are allowed.
in	*ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes.
in	*H_row*	is a one-dimensional array of size ne and type int, that holds the row indices of the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes, and in this case can be NULL
in	*H_col*	is a one-dimensional array of size ne and type int, that holds the column indices of the lower triangular part of H in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL
in	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of H, as well as the total number of entries, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL

Examples

trbt.c, and trbtf.c.

3.1.2.4 trb_reset_control()

```
void trb_reset_control (
            struct trb_control_type * control,
            void ** data,
            int * status )
```

Reset control parameters after import if required.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see trb_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 1. The import was succesful, and the package is ready for the solve phase

3.1.2.5 trb_solve_with_mat()

```
void trb_solve_with_mat (
            void ** data,
            void * userdata,
            int * status,
            int n,
            real_wp_ x[],
            real_wp_ g[],
            int ne,
            int(*)(int, const real_wp_[], real_wp_ *, const void *) eval_f,
            int(*)(int, const real_wp_[], real_wp_[], const void *) eval_g,
            int(*)(int, int, const real_wp_[], real_wp_[], const void *) eval_h,
            int(*)(int, const real_wp_[], real_wp_[], const real_wp_[], const void *) eval_↩
prec )
```

Find a local minimizer of a given function subject to simple bounds on the variables using a trust-region method.

This call is for the case where $H = \nabla_{xx} f(x)$ is provided specifically, and all function/derivative information is available by function calls.

Parameters

in,out	*data*	holds private internal data
in	*userdata*	is a structure that allows data to be passed into the function and derivative evaluation programs.

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.
in	*n*	is a scalar variable of type int, that holds the number of variables
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in,out	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
in	*ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix H.
	eval_f	is a user-supplied function that must have the following signature: `int eval_f( int n, const double x[], double *f, const void *userdata )` The value of the objective function $f(x)$ evaluated at x= x must be assigned to f, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_f` via the structure `userdata`.

Parameters

	eval_g	is a user-supplied function that must have the following signature: `int eval_g( int n, const double x[], double g[], const void *userdata )` The components of the gradient $g = \nabla_x f(x)$ of the objective function evaluated at x= x must be assigned to g, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_g` via the structure `userdata`.
	eval_h	is a user-supplied function that must have the following signature: `int eval_h( int n, int ne, const double x[], double h[], const void *userdata )` The nonzeros of the Hessian $H = \nabla_{xx} f(x)$ of the objective function evaluated at x= x must be assigned to h in the same order as presented to trb_import, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_h` via the structure `userdata`.
	eval_prec	is an optional user-supplied function that may be NULL. If non-NULL, it must have the following signature: `int eval_prec( int n, const double x[], double u[], const double v[], const void *userdata )` The product $u = P(x)v$ of the user's preconditioner $P(x)$ evaluated at x with the vector v = v, the result u must be retured in u, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_prec` via the structure `userdata`.

Examples

trbt.c, and trbtf.c.

3.1.2.6 trb_solve_without_mat()

```
void trb_solve_without_mat (
            void ** data,
            void * userdata,
            int * status,
            int n,
            real_wp_ x[],
            real_wp_ g[],
            int(*)(int, const real_wp_[], real_wp_ *, const void *) eval_f,
            int(*)(int, const real_wp_[], real_wp_[], const void *) eval_g,
            int(*)(int, const real_wp_[], real_wp_[], const real_wp_[], bool, const void *)
eval_hprod,
            int(*)(int, const real_wp_[], int, const int[], const real_wp_[], int *, int[],
real_wp_[], bool, const void *) eval_shprod,
            int(*)(int, const real_wp_[], real_wp_[], const real_wp_[], const void *) eval_↩
prec )
```

Find a local minimizer of a given function subject to simple bounds on the variables using a trust-region method.

This call is for the case where access to $H = \nabla_{xx} f(x)$ is provided by Hessian-vector products, and all function/derivative information is available by function calls.

Parameters

`in,out`	*data*	holds private internal data

Parameters

in	*userdata*	is a structure that allows data to be passed into the function and derivative evaluation programs.
in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.
in	*n*	is a scalar variable of type int, that holds the number of variables
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in,out	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
	eval_f	is a user-supplied function that must have the following signature: `int eval_f( int n, const double x[], double *f, const void *userdata )` The value of the objective function $f(x)$ evaluated at x= x must be assigned to f, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_f` via the structure `userdata`.
	eval_g	is a user-supplied function that must have the following signature:

```
int eval_g( int n, const double x[], double g[], const void *userdata )
```

The components of the gradient $g = \nabla_x f(x)$ of the objective function evaluated at x= x must be assigned to g, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_g` via the structure `userdata`.

Parameters

eval_hprod	is a user-supplied function that must have the following signature: `int eval_hprod( int n, const double x[], double u[], const double v[], bool got_h, const void *userdata )` The sum $u + \nabla_{xx} f(x) v$ of the product of the Hessian $\nabla_{xx} f(x)$ of the objective function evaluated at x= x with the vector v= v and the vector $ u must be returned in u, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. The Hessian has already been evaluated or used at x if got_h is true. Data may be passed into `eval_hprod` via the structure `userdata`.

eval_shprod	is a user-supplied function that must have the following signature:
	`int eval_shprod( int n, const double x[], int nnz_v, const int index_nz_v[], const double v[],`
	`int *nnz_u, int index_nz_u[], double u[],`
	`bool got_h, const void *userdata )`
	The product $u = \nabla_{xx} f(x) v$ of the Hessian $\nabla_{xx} f(x)$ of the objective function evaluated at x with the sparse vector v= v must be returned in u, and the function return value set to 0. Only the components index_nz_v[0:nnz_v-1] of v are nonzero, and the remaining components may not have been be set. On exit, the user must indicate the nnz_u indices of u that are nonzero in index_nz_u[0:nnz_u-1], and only these components of u need be set. If the evaluation is impossible at x, return should be set to a nonzero value. The Hessian has already been evaluated or used at x if got_h is true. Data may be passed into `eval_prec` via the structure `userdata`.

eval_prec	is an optional user-supplied function that may be NULL. If non-NULL, it must have the following signature: `int eval_prec( int n, const double x[], double u[], const double v[], const void *userdata )` The product $u = P(x) v$ of the user's preconditioner $P(x)$ evaluated at x with the vector v = v, the result u must be retured in u, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_prec` via the structure `userdata`.

Examples

trbt.c, and trbtf.c.

3.1.2.7 trb_solve_reverse_with_mat()

```
void trb_solve_reverse_with_mat (
            void ** data,
            int * status,
            int * eval_status,
            int n,
            real_wp_ x[],
            real_wp_ f,
            real_wp_ g[],
            int ne,
            real_wp_ H_val[],
```

```
const real_wp_ u[],
real_wp_ v[] )
```

Find a local minimizer of a given function subject to simple bounds on the variables using a trust-region method.

This call is for the case where $H = \nabla_{xx} f(x)$ is provided specifically, but function/derivative information is only available by returning to the calling procedure

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.

Parameters

	status	(continued) • 2. The user should compute the objective function value $f(x)$ at the point x indicated in x and then re-enter the function. The required value should be set in f, and eval_status should be set to 0. If the user is unable to evaluate $f(x)$— for instance, if the function is undefined at x— the user need not set f, but should then set eval_status to a non-zero value. • 3. The user should compute the gradient of the objective function $\nabla_x f(x)$ at the point x indicated in x and then re-enter the function. The value of the i-th component of the g radient should be set in g[i], for i = 0, ..., n-1 and eval_status should be set to 0. If the user is unable to evaluate a component of $\nabla_x f(x)$ — for instance if a component of the gradient is undefined at x -the user need not set g, but should then set eval_status to a non-zero value. • 4. The user should compute the Hessian of the objective function $\nabla_{xx} f(x)$ at the point x indicated in x and then re-enter the function. The value l-th component of the Hessian stored according to the scheme input in the remainder of H should be set in H_val[l], for l = 0, ..., ne-1 and eval_status should be set to 0. If the user is unable to evaluate a component of $\nabla_{xx} f(x)$ — for instance, if a component of the Hessian is undefined at x — the user need not set H_val, but should then set eval_status to a non-zero value. • 6. The user should compute the product $u = P(x)v$ of their preconditioner $P(x)$ at the point x indicated in x with the vector v and then re-enter the function. The vector v is given in v, the resulting vector $u = P(x)v$ should be set in u and 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 x — the user need not set u, but should then set eval_status to a non-zero value.
`in,out`	*eval_status*	is a scalar variable of type int, that is used to indicate if objective function/gradient/Hessian values can be provided (see above)
`in`	*n*	is a scalar variable of type int, that holds the number of variables
`in,out`	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
`in`	*f*	is a scalar variable pointer of type double, that holds the value of the objective function.
`in,out`	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
`in`	*ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix H.
`in`	*H_val*	is a one-dimensional array of size ne and type double, that holds the values of the entries of the lower triangular part of the Hessian matrix H in any of the available storage schemes.
`in`	*u*	is a one-dimensional array of size n and type double, that is used for reverse communication (see above for details)
`in,out`	*v*	is a one-dimensional array of size n and type double, that is used for reverse communication (see above for details)

Examples

trbt.c, and trbtf.c.

3.1.2.8 trb_solve_reverse_without_mat()

```
void trb_solve_reverse_without_mat (
            void ** data,
            int * status,
            int * eval_status,
            int n,
            real_wp_ x[],
            real_wp_ f,
            real_wp_ g[],
            real_wp_ u[],
            real_wp_ v[],
            int index_nz_v[],
            int * nnz_v,
            const int index_nz_u[],
            int nnz_u )
```

Find a local minimizer of a given function subject to simple bounds on the variables using a trust-region method.

This call is for the case where access to $H = \nabla_{xx} f(x)$ is provided by Hessian-vector products, but function/derivative information is only available by returning to the calling procedure.

Parameters

in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.

Parameters

	status	(continued) • 2. The user should compute the objective function value $f(x)$ at the point x indicated in x and then re-enter the function. The required value should be set in f, and eval_status should be set to 0. If the user is unable to evaluate $f(x)$ — for instance, if the function is undefined at x — the user need not set f, but should then set eval_status to a non-zero value. • 3. The user should compute the gradient of the objective function $\nabla_x f(x)$ at the point x indicated in x and then re-enter the function. The value of the i-th component of the g radient should be set in g[i], for i = 0, ..., n-1 and eval_status should be set to 0. If the user is unable to evaluate a component of $\nabla_x f(x)$ — for instance if a component of the gradient is undefined at x -the user need not set g, but should then set eval_status to a non-zero value. • 5. The user should compute the product $\nabla_{xx} f(x)v$ of the Hessian of the objective function $\nabla_{xx} f(x)$ at the point x indicated in x with the vector v, add the result to the vector u and then re-enter the function. The vectors u and v are given in u and v respectively, the resulting vector $u + \nabla_{xx} f(x)v$ should be set in u and 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 x — the user need not alter u, but should then set eval_status to a non-zero value. • 6. The user should compute the product $u = P(x)v$ of their preconditioner $P(x)$ at the point x indicated in x with the vector v and then re-enter the function. The vector v is given in v, the resulting vector $u = P(x)v$ should be set in u and 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 x — the user need not set u, but should then set eval_status to a non-zero value. • 7. The user should compute the product $u = \nabla_{xx} f(x)v$ of the Hessian of the objective function $\nabla_{xx} f(x)$ at the point x indicated in x with the **sparse** vector v= v and then re-enter the function. The nonzeros of v are stored in v[index_nz_v[0:nnz_v-1]] while the nonzeros of u should be returned in u[index_nz_u[0:nnz_u-1]]; the user must set nnz_u and index_nz_u accordingly, and set eval_status to 0. If the user is unable to evaluate the product— for instance, if a component of the Hessian is undefined at x— the user need not alter u, but should then set eval_status to a non-zero value.
`in,out`	*eval_status*	is a scalar variable of type int, that is used to indicate if objective function/gradient/Hessian values can be provided (see above)
`in`	*n*	is a scalar variable of type int, that holds the number of variables
`in,out`	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
`in`	*f*	is a scalar variable pointer of type double, that holds the value of the objective function.
`in,out`	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
`in,out`	*u*	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=5,6,7 above for details)
`in,out`	*v*	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=5,6,7 above for details)
`in,out`	*index_nz↩_v*	is a one-dimensional array of size n and type int, that is used for reverse communication (see status=7 above for details)

Parameters

in,out	*nnz_v*	is a scalar variable of type int, that is used for reverse communication (see status=7 above for details)
in	*index_nz_u*	s a one-dimensional array of size n and type int, that is used for reverse communication (see status=7 above for details)
in	*nnz_u*	is a scalar variable of type int, that is used for reverse communication (see status=7 above for details). On initial (status=1) entry, nnz_u should be set to an (arbitrary) nonzero value, and nnz_u=0 is recommended

Examples

trbt.c, and trbtf.c.

3.1.2.9 trb_information()

```
void trb_information (
            void ** data,
            struct trb_inform_type * inform,
            int * status )
```

Provides output information

Parameters

in,out	*data*	holds private internal data
out	*inform*	is a struct containing output information (see trb_inform_type)
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): • 0. The values were recorded succesfully

Examples

trbt.c, and trbtf.c.

3.1.2.10 trb_terminate()

```
void trb_terminate (
            void ** data,
            struct trb_control_type * control,
            struct trb_inform_type * inform )
```

Deallocate all internal private storage

Parameters

in,out	*data*	holds private internal data
out	*control*	is a struct containing control information (see trb_control_type)
out	*inform*	is a struct containing output information (see trb_inform_type)

Examples

trbt.c, and trbtf.c.

Chapter 4

Example Documentation

4.1 trbt.c

This is an example of how to use the package both when the Hessian is directly available and when its product with vectors may be found. Both function call evaluations and returns to the calling program to find the required values are illustrated. A variety of supported Hessian storage formats are shown.

Notice that C-style indexing is used, and that this is flaggeed by setting `control.f_indexing` to `false`. In addition, see how parameters may be passed into the evaluation functions via `userdata`.

```
/* trbt.c */
/* Full test for the TRB C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_trb.h"
// Custom userdata struct
struct userdata_type {
   real_wp_ p;
};
// Function prototypes
int fun( int n, const real_wp_ x[], real_wp_ *f, const void * );
int grad( int n, const real_wp_ x[], real_wp_ g[], const void * );
int hess( int n, int ne, const real_wp_ x[], real_wp_ hval[], const void * );
int hess_dense( int n, int ne, const real_wp_ x[], real_wp_ hval[], const void * );
int hessprod( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
              bool got_h, const void * );
int shessprod( int n, const real_wp_ x[], int nnz_v, const int index_nz_v[],
               const real_wp_ v[], int *nnz_u, int index_nz_u[], real_wp_ u[],
              bool got_h, const void * );
int prec( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[], const void * );
int fun_diag( int n, const real_wp_ x[], real_wp_ *f, const void * );
int grad_diag( int n, const real_wp_ x[], real_wp_ g[], const void * );
int hess_diag( int n, int ne, const real_wp_ x[], real_wp_ hval[], const void * );
int hessprod_diag( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
                   bool got_h, const void * );
int shessprod_diag( int n, const real_wp_ x[], int nnz_v, const int index_nz_v[],
                    const real_wp_ v[], int *nnz_u, int index_nz_u[], real_wp_ u[],
                    bool got_h, const void * );
int main(void) {
    // Derived types
    void *data;
    struct trb_control_type control;
    struct trb_inform_type inform;
    // Set user data
    struct userdata_type userdata;
    userdata.p = 4.0;
    // Set problem data
    int n = 3; // dimension
    int ne = 5; // Hesssian elements
    real_wp_ x_l[] = {-10,-10,-10};
    real_wp_ x_u[] = {0.5,0.5,0.5};
```

```
    int H_row[] = {0, 1, 2, 2, 2}; // Hessian H
    int H_col[] = {0, 1, 0, 1, 2}; // NB lower triangle
    int H_ptr[] = {0, 1, 2, 5};    // row pointers
    // Set storage
    real_wp_ g[n]; // gradient
    char st;
    int status;
    printf(" C sparse matrix indexing\n\n");
    printf(" tests options for all-in-one storage format\n\n");
    for( int d=1; d <= 5; d++){
        // Initialize TRB
        trb_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = false; // C sparse matrix indexing
        //control.print_level = 1;
        // Start from 1.5
        real_wp_ x[] = {1.5,1.5,1.5};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "coordinate", ne, H_row, H_col, NULL );
                trb_solve_with_mat( &data, &userdata, &status, n, x, g, ne,
                                    fun, grad, hess, prec );
                break;
            case 2: // sparse by rows
                st = 'R';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "sparse_by_rows", ne, NULL, H_col, H_ptr );
                trb_solve_with_mat( &data, &userdata, &status, n, x, g, ne,
                                    fun, grad, hess, prec );
                break;
            case 3: // dense
                st = 'D';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "dense", ne, NULL, NULL, NULL );
                trb_solve_with_mat( &data, &userdata, &status, n, x, g, ne,
                                    fun, grad, hess_dense, prec );
                break;
            case 4: // diagonal
                st = 'I';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "diagonal", ne, NULL, NULL, NULL );
                trb_solve_with_mat (&data, &userdata, &status, n, x, g, ne,
                                    fun_diag, grad_diag, hess_diag, prec );
                break;
            case 5: // access by products
                st = 'P';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "absent", ne, NULL, NULL, NULL );
                trb_solve_without_mat( &data, &userdata, &status, n, x, g,
                                       fun, grad, hessprod, shessprod, prec );
                break;
        }
        // Record solution information
        trb_information( &data, &inform, &status );
        // Print solution details
        if(inform.status == 0){
            printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
                   st, inform.iter, inform.obj, inform.status);
        }else{
            printf("%c: TRB_solve exit status = %1i\n", st, inform.status);
        }
        //printf("x: ");
        //for( int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
        //for( int i = 0; i < n; i++) printf("%f ", g[i]);
        //printf("\n");
        // Delete internal workspace
        trb_terminate( &data, &control, &inform );
    }
    printf("\n tests reverse-communication options\n\n");
    // reverse-communication input/output
    int eval_status, nnz_v;
    int nnz_u;
    real_wp_ f = 0.0;
    real_wp_ u[n], v[n];
    int index_nz_u[n], index_nz_v[n];
    real_wp_ H_val[ne], H_dense[n*(n+1)/2], H_diag[n];
    for( int d=1; d <= 5; d++){
        // Initialize TRB
        trb_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = false; // C sparse matrix indexing
        //control.print_level = 1;
        // Start from 1.5
```

```
    real_wp_ x[] = {1.5,1.5,1.5};
    switch(d){
        case 1: // sparse co-ordinate storage
            st = 'C';
            trb_import( &control, &data, &status, n, x_l, x_u,
                        "coordinate", ne, H_row, H_col, NULL );
            while(true){ // reverse-communication loop
                trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                            n, x, f, g, ne, H_val, u, v );
                if(status == 0){ // successful termination
                    break;
                }else if(status < 0){ // error exit
                    break;
                }else if(status == 2){ // evaluate f
                    eval_status = fun( n, x, &f, &userdata );
                }else if(status == 3){ // evaluate g
                    eval_status = grad( n, x, g, &userdata );
                }else if(status == 4){ // evaluate H
                    eval_status = hess( n, ne, x, H_val, &userdata );
                }else if(status == 6){ // evaluate the product with P
                    eval_status = prec( n, x, u, v, &userdata );
                }else{
                    printf(" the value %1i of status should not occur\n", status);
                    break;
                }
            }
            break;
        case 2: // sparse by rows
            st = 'R';
            trb_import( &control, &data, &status, n, x_l, x_u,
                        "sparse_by_rows", ne, NULL, H_col, H_ptr );
            while(true){ // reverse-communication loop
                trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                            n, x, f, g, ne, H_val, u, v );
                if(status == 0){ // successful termination
                    break;
                }else if(status < 0){ // error exit
                    break;
                }else if(status == 2){ // evaluate f
                    eval_status = fun( n, x, &f, &userdata );
                }else if(status == 3){ // evaluate g
                    eval_status = grad( n, x, g, &userdata );
                }else if(status == 4){ // evaluate H
                    eval_status = hess( n, ne, x, H_val, &userdata );
                }else if(status == 6){ // evaluate the product with P
                    eval_status = prec( n, x, u, v, &userdata );
                }else{
                    printf(" the value %1i of status should not occur\n", status);
                    break;
                }
            }
            break;
        case 3: // dense
            st = 'D';
            trb_import( &control, &data, &status, n, x_l, x_u,
                        "dense", ne, NULL, NULL, NULL );
            while(true){ // reverse-communication loop
                trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                            n, x, f, g, n*(n+1)/2, H_dense,
                                            u, v );
                if(status == 0){ // successful termination
                    break;
                }else if(status < 0){ // error exit
                    break;
                }else if(status == 2){ // evaluate f
                    eval_status = fun( n, x, &f, &userdata );
                }else if(status == 3){ // evaluate g
                    eval_status = grad( n, x, g, &userdata );
                }else if(status == 4){ // evaluate H
                    eval_status = hess_dense( n, n*(n+1)/2, x, H_dense,
                                              &userdata );
                }else if(status == 6){ // evaluate the product with P
                    eval_status = prec( n, x, u, v, &userdata );
                }else{
                    printf(" the value %1i of status should not occur\n", status);
                    break;
                }
            }
            break;
        case 4: // diagonal
            st = 'I';
            trb_import( &control, &data, &status, n, x_l, x_u,
                        "diagonal", ne, NULL, NULL, NULL );
            while(true){ // reverse-communication loop
                trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                            n, x, f, g, n, H_diag, u, v );
                if(status == 0){ // successful termination
```

```
                    break;
                }else if(status < 0){ // error exit
                    break;
                }else if(status == 2){ // evaluate f
                    eval_status = fun_diag( n, x, &f, &userdata );
                }else if(status == 3){ // evaluate g
                    eval_status = grad_diag( n, x, g, &userdata );
                }else if(status == 4){ // evaluate H
                    eval_status = hess_diag( n, n, x, H_diag, &userdata );
                }else if(status == 6){ // evaluate the product with P
                    eval_status = prec( n, x, u, v, &userdata );
                }else{
                    printf(" the value %1i of status should not occur\n", status);
                    break;
                }
            }
            break;
        case 5: // access by products
            st = 'P';
            trb_import( &control, &data, &status, n, x_l, x_u,
                        "absent", ne, NULL, NULL, NULL );
            nnz_u = 0;
            while(true){ // reverse-communication loop
                trb_solve_reverse_without_mat( &data, &status, &eval_status,
                                               n, x, f, g, u, v, index_nz_v,
                                               &nnz_v, index_nz_u, nnz_u );
                if(status == 0){ // successful termination
                    break;
                }else if(status < 0){ // error exit
                    break;
                }else if(status == 2){ // evaluate f
                    eval_status = fun( n, x, &f, &userdata );
                }else if(status == 3){ // evaluate g
                    eval_status = grad( n, x, g, &userdata );
                }else if(status == 5){ // evaluate H
                    eval_status = hessprod( n, x, u, v, false, &userdata );
                }else if(status == 6){ // evaluate the product with P
                    eval_status = prec(n, x, u, v, &userdata );
                }else if(status == 7){ // evaluate sparse Hessian-vect prod
                    eval_status = shessprod( n, x, nnz_v, index_nz_v, v,
                                             &nnz_u, index_nz_u, u,
                                             false, &userdata );
                }else{
                    printf(" the value %1i of status should not occur\n", status);
                    break;
                }
            }
            break;
        }
        // Record solution information
        trb_information( &data, &inform, &status );
        // Print solution details
        if(inform.status == 0){
            printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
                   st, inform.iter, inform.obj, inform.status);
        }else{
            printf("%c: TRB_solve exit status = %1i\n", st, inform.status);
        }
        //printf("x: ");
        //for( int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
        //for( int i = 0; i < n; i++) printf("%f ", g[i]);
        //printf("\n");
        // Delete internal workspace
        trb_terminate( &data, &control, &inform );
    }
}
// Objective function
int fun( int n, const real_wp_ x[], real_wp_ *f, const void *userdata ){
         struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    *f = pow(x[0] + x[2] + p, 2) + pow(x[1] + x[2], 2) + cos(x[0]);
    return 0;
}
// Gradient of the objective
int grad( int n, const real_wp_ x[], real_wp_ g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    g[0] = 2.0 * ( x[0] + x[2] + p ) - sin(x[0]);
    g[1] = 2.0 * ( x[1] + x[2] );
    g[2] = 2.0 * ( x[0] + x[2] + p ) + 2.0 * ( x[1] + x[2] );
    return 0;
}
// Hessian of the objective
int hess( int n, int ne, const real_wp_ x[], real_wp_ hval[],
          const void *userdata ){
```

```
    hval[0] = 2.0 - cos(x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 4.0;
    return 0;
}
// Dense Hessian
int hess_dense( int n, int ne, const real_wp_ x[], real_wp_ hval[],
                const void *userdata ){
    hval[0] = 2.0 - cos(x[0]);
    hval[1] = 0.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 2.0;
    hval[5] = 4.0;
    return 0;
}
// Hessian-vector product
int hessprod( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
              bool got_h, const void *userdata ){
    u[0] = u[0] + 2.0 * ( v[0] + v[2] ) - cos(x[0]) * v[0];
    u[1] = u[1] + 2.0 * ( v[1] + v[2] );
    u[2] = u[2] + 2.0 * ( v[0] + v[1] + 2.0 * v[2] );
    return 0;
}
// Sparse Hessian-vector product
int shessprod( int n, const real_wp_ x[], int nnz_v, const int index_nz_v[],
               const real_wp_ v[], int *nnz_u, int index_nz_u[], real_wp_ u[],
               bool got_h, const void *userdata ){
    real_wp_ p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for( int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 0:
                p[0] = p[0] + 2.0 * v[0] - cos(x[0]) * v[0];
                used[0] = true;
                p[2] = p[2] + 2.0 * v[0];
                used[2] = true;
                break;
            case 1:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;
                p[2] = p[2] + 2.0 * v[1];
                used[2] = true;
                break;
            case 2:
                p[0] = p[0] + 2.0 * v[2];
                used[0] = true;
                p[1] = p[1] + 2.0 * v[2];
                used[1] = true;
                p[2] = p[2] + 4.0 * v[2];
                used[2] = true;
                break;
        }
    }
    *nnz_u = 0;
    for( int j = 0; j < 3; j++){
        if(used[j]){
        u[j] = p[j];
        *nnz_u = *nnz_u + 1;
        index_nz_u[*nnz_u-1] = j;
        }
    }
    return 0;
}
// Apply preconditioner
int prec( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
          const void *userdata ){
   u[0] = 0.5 * v[0];
   u[1] = 0.5 * v[1];
   u[2] = 0.25 * v[2];
   return 0;
}
 // Objective function
int fun_diag( int n, const real_wp_ x[], real_wp_ *f, const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    *f = pow(x[2] + p, 2) + pow(x[1], 2) + cos(x[0]);
    return 0;
}
// Gradient of the objective
int grad_diag( int n, const real_wp_ x[], real_wp_ g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    g[0] = -sin(x[0]);
```

```
    g[1] = 2.0 * x[1];
    g[2] = 2.0 * ( x[2] + p );
    return 0;
}
// Hessian of the objective
int hess_diag( int n, int ne, const real_wp_ x[], real_wp_ hval[],
               const void *userdata ){
    hval[0] = -cos(x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    return 0;
}
// Hessian-vector product
int hessprod_diag( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
                   bool got_h, const void *userdata ){
    u[0] = u[0] + - cos(x[0]) * v[0];
    u[1] = u[1] + 2.0 * v[1];
    u[2] = u[2] + 2.0 * v[2];
    return 0;
}
// Sparse Hessian-vector product
int shessprod_diag( int n, const real_wp_ x[], int nnz_v, const int index_nz_v[],
                    const real_wp_ v[], int *nnz_u, int index_nz_u[], real_wp_ u[],
                     bool got_h, const void *userdata ){
    real_wp_ p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for( int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 0:
                p[0] = p[0] - cos(x[0]) * v[0];
                used[0] = true;
                break;
            case 1:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;
                break;
            case 2:
                p[2] = p[2] + 2.0 * v[2];
                used[2] = true;
                break;
        }
    }
    *nnz_u = 0;
    for( int j = 0; j < 3; j++){
        if(used[j]){
        u[j] = p[j];
        *nnz_u = *nnz_u + 1;
        index_nz_u[*nnz_u-1] = j;
        }
    }
    return 0;
}
```

4.2 trbtf.c

This is the same example, but now fortran-style indexing is used.

```
/* trbtf.c */
/* Full test for the TRB C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_trb.h"
// Custom userdata struct
struct userdata_type {
   real_wp_ p;
};
// Function prototypes
int fun( int n, const real_wp_ x[], real_wp_ *f, const void * );
int grad( int n, const real_wp_ x[], real_wp_ g[], const void * );
int hess( int n, int ne, const real_wp_ x[], real_wp_ hval[], const void * );
int hess_dense( int n, int ne, const real_wp_ x[], real_wp_ hval[],
                const void * );
int hessprod( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
              bool got_h, const void * );
int shessprod( int n, const real_wp_ x[], int nnz_v, const int index_nz_v[],
               const real_wp_ v[], int *nnz_u, int index_nz_u[], real_wp_ u[],
              bool got_h, const void * );
int prec( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
```

```
          const void * );
int fun_diag( int n, const real_wp_ x[], real_wp_ *f, const void * );
int grad_diag( int n, const real_wp_ x[], real_wp_ g[], const void * );
int hess_diag( int n, int ne, const real_wp_ x[], real_wp_ hval[],
               const void * );
int hessprod_diag( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
                   bool got_h, const void * );
int shessprod_diag( int n, const real_wp_ x[], int nnz_v,
                    const int index_nz_v[],
                    const real_wp_ v[], int *nnz_u, int index_nz_u[],
                    real_wp_ u[], bool got_h, const void * );
int main(void) {
    // Derived types
    void *data;
    struct trb_control_type control;
    struct trb_inform_type inform;
    // Set user data
    struct userdata_type userdata;
    userdata.p = 4.0;
    // Set problem data
    int n = 3; // dimension
    int ne = 5; // Hesssian elements
    real_wp_ x_l[] = {-10,-10,-10};
    real_wp_ x_u[] = {0.5,0.5,0.5};
    int H_row[] = {1, 2, 3, 3, 3}; // Hessian H
    int H_col[] = {1, 2, 1, 2, 3}; // NB lower triangle
    int H_ptr[] = {1, 2, 3, 6};    // row pointers
    // Set storage
    real_wp_ g[n]; // gradient
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" tests options for all-in-one storage format\n\n");
    for( int d=1; d <= 5; d++){
        // Initialize TRB
        trb_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        //control.print_level = 1;
        // Start from 1.5
        real_wp_ x[] = {1.5,1.5,1.5};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "coordinate", ne, H_row, H_col, NULL );
                trb_solve_with_mat( &data, &userdata, &status, n, x, g, ne,
                                    fun, grad, hess, prec );
                break;
            case 2: // sparse by rows
                st = 'R';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "sparse_by_rows", ne, NULL, H_col, H_ptr );
                trb_solve_with_mat( &data, &userdata, &status, n, x, g, ne,
                                    fun, grad, hess, prec );
                break;
            case 3: // dense
                st = 'D';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "dense", ne, NULL, NULL, NULL );
                trb_solve_with_mat( &data, &userdata, &status, n, x, g, ne,
                                    fun, grad, hess_dense, prec );
                break;
            case 4: // diagonal
                st = 'I';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "diagonal", ne, NULL, NULL, NULL );
                trb_solve_with_mat (&data, &userdata, &status, n, x, g, ne,
                                    fun_diag, grad_diag, hess_diag, prec );
                break;
            case 5: // access by products
                st = 'P';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "absent", ne, NULL, NULL, NULL );
                trb_solve_without_mat( &data, &userdata, &status, n, x, g,
                                       fun, grad, hessprod, shessprod, prec );
                break;
        }
        // Record solution information
        trb_information( &data, &inform, &status );
        // Print solution details
        if(inform.status == 0){
            printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
                   st, inform.iter, inform.obj, inform.status);
        }else{
            printf("%c: TRB_solve exit status = %1i\n", st, inform.status);
        }
```

```
        //printf("x: ");
        //for( int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
        //for( int i = 0; i < n; i++) printf("%f ", g[i]);
        //printf("\n");
        // Delete internal workspace
        trb_terminate( &data, &control, &inform );
    }
    printf("\n tests reverse-communication options\n\n");
    // reverse-communication input/output
    int eval_status, nnz_v;
    int nnz_u;
    real_wp_ f = 0.0;
    real_wp_ u[n], v[n];
    int index_nz_u[n], index_nz_v[n];
    real_wp_ H_val[ne], H_dense[n*(n+1)/2], H_diag[n];
    for( int d=1; d <= 5; d++){
        // Initialize TRB
        trb_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        //control.print_level = 1;
        // Start from 1.5
        real_wp_ x[] = {1.5,1.5,1.5};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "coordinate", ne, H_row, H_col, NULL );
                while(true){ // reverse-communication loop
                    trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                                n, x, f, g, ne, H_val, u, v );
                    if(status == 0){ // successful termination
                        break;
                    }else if(status < 0){ // error exit
                        break;
                    }else if(status == 2){ // evaluate f
                        eval_status = fun( n, x, &f, &userdata );
                    }else if(status == 3){ // evaluate g
                        eval_status = grad( n, x, g, &userdata );
                    }else if(status == 4){ // evaluate H
                        eval_status = hess( n, ne, x, H_val, &userdata );
                    }else if(status == 6){ // evaluate the product with P
                        eval_status = prec( n, x, u, v, &userdata );
                    }else{
                        printf(" the value %1i of status should not occur\n", status);
                        break;
                    }
                }
                break;
            case 2: // sparse by rows
                st = 'R';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "sparse_by_rows", ne, NULL, H_col, H_ptr );
                while(true){ // reverse-communication loop
                    trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                                n, x, f, g, ne, H_val, u, v );
                    if(status == 0){ // successful termination
                        break;
                    }else if(status < 0){ // error exit
                        break;
                    }else if(status == 2){ // evaluate f
                        eval_status = fun( n, x, &f, &userdata );
                    }else if(status == 3){ // evaluate g
                        eval_status = grad( n, x, g, &userdata );
                    }else if(status == 4){ // evaluate H
                        eval_status = hess( n, ne, x, H_val, &userdata );
                    }else if(status == 6){ // evaluate the product with P
                        eval_status = prec( n, x, u, v, &userdata );
                    }else{
                        printf(" the value %1i of status should not occur\n", status);
                        break;
                    }
                }
                break;
            case 3: // dense
                st = 'D';
                trb_import( &control, &data, &status, n, x_l, x_u,
                            "dense", ne, NULL, NULL, NULL );
                while(true){ // reverse-communication loop
                    trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                                n, x, f, g, n*(n+1)/2, H_dense,
                                                u, v );
                    if(status == 0){ // successful termination
                        break;
                    }else if(status < 0){ // error exit
```

```
                break;
            }else if(status == 2){ // evaluate f
                eval_status = fun( n, x, &f, &userdata );
            }else if(status == 3){ // evaluate g
                eval_status = grad( n, x, g, &userdata );
            }else if(status == 4){ // evaluate H
                eval_status = hess_dense( n, n*(n+1)/2, x, H_dense,
                                          &userdata );
            }else if(status == 6){ // evaluate the product with P
                eval_status = prec( n, x, u, v, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n", status);
                break;
            }
        }
        break;
    case 4: // diagonal
        st = 'I';
        trb_import( &control, &data, &status, n, x_l, x_u,
                    "diagonal", ne, NULL, NULL, NULL );
        while(true){ // reverse-communication loop
            trb_solve_reverse_with_mat( &data, &status, &eval_status,
                                        n, x, f, g, n, H_diag, u, v );
            if(status == 0){ // successful termination
                break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate f
                eval_status = fun_diag( n, x, &f, &userdata );
            }else if(status == 3){ // evaluate g
                eval_status = grad_diag( n, x, g, &userdata );
            }else if(status == 4){ // evaluate H
                eval_status = hess_diag( n, n, x, H_diag, &userdata );
            }else if(status == 6){ // evaluate the product with P
                eval_status = prec( n, x, u, v, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n", status);
                break;
            }
        }
        break;
    case 5: // access by products
        st = 'P';
        trb_import( &control, &data, &status, n, x_l, x_u,
                    "absent", ne, NULL, NULL, NULL );
        nnz_u = 0;
        while(true){ // reverse-communication loop
            trb_solve_reverse_without_mat( &data, &status, &eval_status,
                                           n, x, f, g, u, v, index_nz_v,
                                           &nnz_v, index_nz_u, nnz_u );
            if(status == 0){ // successful termination
                break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate f
                eval_status = fun( n, x, &f, &userdata );
            }else if(status == 3){ // evaluate g
                eval_status = grad( n, x, g, &userdata );
            }else if(status == 5){ // evaluate H
                eval_status = hessprod( n, x, u, v, false, &userdata );
            }else if(status == 6){ // evaluate the product with P
                eval_status = prec(n, x, u, v, &userdata );
            }else if(status == 7){ // evaluate sparse Hessian-vect prod
                eval_status = shessprod( n, x, nnz_v, index_nz_v, v,
                                         &nnz_u, index_nz_u, u,
                                         false, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n", status);
                break;
            }
        }
        break;
}
// Record solution information
trb_information( &data, &inform, &status );
// Print solution details
if(inform.status == 0){
    printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
           st, inform.iter, inform.obj, inform.status);
}else{
    printf("%c: TRB_solve exit status = %1i\n", st, inform.status);
}
//printf("x: ");
//for( int i = 0; i < n; i++) printf("%f ", x[i]);
//printf("\n");
//printf("gradient: ");
//for( int i = 0; i < n; i++) printf("%f ", g[i]);
```

```
        //printf("\n");
        // Delete internal workspace
        trb_terminate( &data, &control, &inform );
    }
}
// Objective function
int fun( int n, const real_wp_ x[], real_wp_ *f, const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    *f = pow(x[0] + x[2] + p, 2) + pow(x[1] + x[2], 2) + cos(x[0]);
    return 0;
}
// Gradient of the objective
int grad( int n, const real_wp_ x[], real_wp_ g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    g[0] = 2.0 * ( x[0] + x[2] + p ) - sin(x[0]);
    g[1] = 2.0 * ( x[1] + x[2] );
    g[2] = 2.0 * ( x[0] + x[2] + p ) + 2.0 * ( x[1] + x[2] );
    return 0;
}
// Hessian of the objective
int hess( int n, int ne, const real_wp_ x[], real_wp_ hval[],
          const void *userdata ){
    hval[0] = 2.0 - cos(x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 4.0;
    return 0;
}
// Dense Hessian
int hess_dense( int n, int ne, const real_wp_ x[], real_wp_ hval[],
                const void *userdata ){
    hval[0] = 2.0 - cos(x[0]);
    hval[1] = 0.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 2.0;
    hval[5] = 4.0;
    return 0;
}
// Hessian-vector product
int hessprod( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
              bool got_h, const void *userdata ){
    u[0] = u[0] + 2.0 * ( v[0] + v[2] ) - cos(x[0]) * v[0];
    u[1] = u[1] + 2.0 * ( v[1] + v[2] );
    u[2] = u[2] + 2.0 * ( v[0] + v[1] + 2.0 * v[2] );
    return 0;
}
// Sparse Hessian-vector product
int shessprod( int n, const real_wp_ x[], int nnz_v, const int index_nz_v[],
               const real_wp_ v[], int *nnz_u, int index_nz_u[], real_wp_ u[],
               bool got_h, const void *userdata ){
    real_wp_ p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for( int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 1:
                p[0] = p[0] + 2.0 * v[0] - cos(x[0]) * v[0];
                used[0] = true;
                p[2] = p[2] + 2.0 * v[0];
                used[2] = true;
                break;
            case 2:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;
                p[2] = p[2] + 2.0 * v[1];
                used[2] = true;
                break;
            case 3:
                p[0] = p[0] + 2.0 * v[2];
                used[0] = true;
                p[1] = p[1] + 2.0 * v[2];
                used[1] = true;
                p[2] = p[2] + 4.0 * v[2];
                used[2] = true;
                break;
        }
    }
    *nnz_u = 0;
    for( int j = 0; j < 3; j++){
        if(used[j]){
        u[j] = p[j];
        *nnz_u = *nnz_u + 1;
        index_nz_u[*nnz_u-1] = j+1;
```

```
        }
    }
    return 0;
}
// Apply preconditioner
int prec( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
          const void *userdata ){
   u[0] = 0.5 * v[0];
   u[1] = 0.5 * v[1];
   u[2] = 0.25 * v[2];
   return 0;
}
 // Objective function
int fun_diag( int n, const real_wp_ x[], real_wp_ *f, const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    *f = pow(x[2] + p, 2) + pow(x[1], 2) + cos(x[0]);
    return 0;
}
// Gradient of the objective
int grad_diag( int n, const real_wp_ x[], real_wp_ g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    real_wp_ p = myuserdata->p;
    g[0] = -sin(x[0]);
    g[1] = 2.0 * x[1];
    g[2] = 2.0 * ( x[2] + p );
    return 0;
}
// Hessian of the objective
int hess_diag( int n, int ne, const real_wp_ x[], real_wp_ hval[],
               const void *userdata ){
    hval[0] = -cos(x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    return 0;
}
// Hessian-vector product
int hessprod_diag( int n, const real_wp_ x[], real_wp_ u[], const real_wp_ v[],
                   bool got_h, const void *userdata ){
    u[0] = u[0] + - cos(x[0]) * v[0];
    u[1] = u[1] + 2.0 * v[1];
    u[2] = u[2] + 2.0 * v[2];
    return 0;
}
// Sparse Hessian-vector product
int shessprod_diag( int n, const real_wp_ x[], int nnz_v,
                    const int index_nz_v[],
                    const real_wp_ v[], int *nnz_u, int index_nz_u[],
                    real_wp_ u[], bool got_h, const void *userdata ){
    real_wp_ p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for( int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 0:
                p[0] = p[0] - cos(x[0]) * v[0];
                used[0] = true;
                break;
            case 1:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;
                break;
            case 2:
                p[2] = p[2] + 2.0 * v[2];
                used[2] = true;
                break;
        }
    }
    *nnz_u = 0;
    for( int j = 0; j < 3; j++){
        if(used[j]){
        u[j] = p[j];
        *nnz_u = *nnz_u + 1;
        index_nz_u[*nnz_u-1] = j+1;
        }
    }
    return 0;
}
```