C interfaces to GALAHAD TRU

Chapter 1

GALAHAD C package tru

1.1 Introduction

1.1.1 Purpose

The tru package uses a **trust-region method to find a (local) unconstrained minimizer of a differentiable objective function $\mathbf{f(x)}$ of many variables $\mathbf{x}$.** 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.

1.1.2 Authors

N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England, and Ph. L. Toint, The University of Namur, Belgium.

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

1.1.3 Originally released

July 2008, 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 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.

An approximate minimizer of the model within the trust region is found 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. This multiplier is found by uni-variate root finding, using a safeguarded Newton-like process, by the GALAHAD packages TRS or DPS (depending on the norm chosen). The iterative approach uses the 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.

1.1.6 Reference

The generic 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 tru package must be called in the following order:

- tru_initialize - provide default control parameters and set up initial data structures

- tru_read_specfile (optional) - override control values by reading replacement values from a file

- tru_import - set up problem data structures and fixed values

- solve the problem by calling one of

  - tru_solve_with_mat - solve using function calls to evaluate function, gradient and Hessian values

  - tru_solve_without_mat - solve using function calls to evaluate function and gradient values and Hessian-vector products

  - tru_solve_reverse_with_mat - solve returning to the calling program to obtain function, gradient and Hessian values, or

  - tru_solve_reverse_without_mat - solve returning to the calling prorgram to obtain function and gradient values and Hessian-vector products

- tru_information (optional) - recover information about the solution and solution process

- tru_terminate - deallocate data structures

See Section 6.1 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 plus one. 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

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 tru_control_type Struct Reference

```
#include <tru.h>
```

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.
- 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 allowed
- int alive_unit

  removal of the file alive_file from unit alive_unit terminates execution
- char alive_file [31]

  see alive_unit
- int non_monotone

  the descent strategy used.
- int model

  the model used.
- int norm

  the trust-region norm used.
- 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

- 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*

- real_wp_ stop_g_absolute

  *overall convergence tolerances. The iteration will terminate when the norm of the gradient of the objective function is smaller than MAX(.stop_g_absolute, .stop_g_relative * norm of the initial gradient or if the step is less than .stop_s*

- real_wp_ stop_g_relative

  see stop_g_absolute

- real_wp_ stop_s

  see stop_g_absolute

- int advanced_start

  try to pick a good initial trust-region radius using .advanced_start iterates of a variant on the strategy of Sartenaer SISC 18(6) 1990:1788-1803

- real_wp_ initial_radius

  initial value for the trust-region radius

- real_wp_ maximum_radius

  maximum permitted trust-region radius

- 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 by 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?
- bool renormalize_radius

    should the radius be renormalized to account for a change in preconditioner?
- 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'

4.1.1 Detailed Description

control derived type as a C struct

Examples

> trut.c, and trutf.c.

4.1.2 Field Documentation

4.1.2.1 model

```
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)

- 6 secant second-order (limited-memory SR1, with .lbfgs_vectors history)

4.1.2.2 non_monotone

```
int non_monotone
```

the descent strategy used.

Possible values are

- $<= 0$ a monotone strategy is used.

- anything else, a non-monotone strategy with history length .non_monotine is used.

4.1.2.3 norm

```
int norm
```

the trust-region norm used.

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*)

- 10 diagonalizing norm from GALAHAD_DPS (*subproblem_direct only*)

4.1.2.4 print_level

```
int print_level
```

the level of output required.

- $\leq$ 0 gives no output,

- = 1 gives a one-line summary for every iteration,

- = 2 gives a summary of the inner iteration for each iteration,

- $\geq$ 3 gives increasingly verbose (debugging) output

The documentation for this struct was generated from the following file:

- tru/tru.h

4.2 tru_inform_type Struct Reference

```
#include <tru.h>
```

Data Fields

- int status

  return status. See TRU_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 f_eval

  the total number of evaluations of the objection function
- int g_eval

  the total number of evaluations of the gradient of the objection function
- int h_eval

  the total number of evaluations of the Hessian of the objection function
- int factorization_max

  the maximum number of factorizations in a sub-problem solve
- int factorization_status

  the return status from the factorization
- long int max_entries_factors

  the maximum number of entries in the factors
- int factorization_integer

  the total integer workspace required for the factorization
- int factorization_real

> *the total real workspace required for the factorization*

- real_wp_ factorization_average

  the average number of factorizations per sub-problem solve

- real_wp_ obj

  the value of the objective function at the best estimate of the solution determined by the package.

- real_wp_ norm_g

  the norm of the gradient of the objective function at the best estimate of the solution determined by the package.

- real_wp_ radius

  the current value of the trust-region radius

- struct tru_time_type time

  timings (see above)

4.2.1 Detailed Description

inform derived type as a C struct

Examples

> trut.c, and trutf.c.

The documentation for this struct was generated from the following file:

- tru/tru.h

4.3 tru_time_type Struct Reference

```
#include <tru.h>
```

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

4.3.1 Detailed Description

time derived type as a C struct

The documentation for this struct was generated from the following file:

- tru/tru.h

Chapter 5

File Documentation

5.1 tru/tru.h File Reference

```
#include <stdbool.h>
#include "galahad_precision.h"
```

Data Structures

- struct tru_control_type
- struct tru_time_type
- struct tru_inform_type

Functions

- void tru_initialize (void ∗∗data, struct tru_control_type ∗control, struct tru_inform_type ∗inform)
- void tru_read_specfile (struct tru_control_type ∗control, const char specfile[ ])
- void tru_import (struct tru_control_type ∗control, void ∗∗data, int ∗status, int n, const char H_type[ ], int ne, const int H_row[ ], const int H_col[ ], const int H_ptr[ ])
- void tru_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 tru_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_prec)(int, const real_wp_[ ], real_wp_[ ], const real_wp_[ ], const void ∗))
- void tru_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 tru_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[ ])
- void tru_information (void ∗∗data, struct tru_inform_type ∗inform, int ∗status)
- void tru_terminate (void ∗∗data, struct tru_control_type ∗control, struct tru_inform_type ∗inform)

5.1.1 Function Documentation

5.1.1.1 tru_import()

```
void tru_import (
            struct tru_control_type * control,
            void ** data,
            int * status,
            int n,
            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 tru_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: • 0. The import 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.
in	n	is a scalar variable of type int, that holds the number of variables
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 plus one, 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

trut.c, and trutf.c.

5.1.1.2 tru_information()

```
void tru_information (
            void ** data,
            struct tru_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 tru_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

trut.c, and trutf.c.

5.1.1.3 tru_initialize()

```
void tru_initialize (
            void ** data,
            struct tru_control_type * control,
            struct tru_inform_type * inform )
```

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 tru_control_type)
out	*inform*	is a struct containing output information (see tru_inform_type)

Examples

trut.c, and trutf.c.

5.1.1.4 tru_read_specfile()

```
void tru_read_specfile (
            struct tru_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

Parameters

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

5.1.1.5 tru_solve_reverse_with_mat()

```
void tru_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 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

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 import 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.
		
		    • -40. 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

trut.c, and trutf.c.

5.1.1.6 tru_solve_reverse_without_mat()

```
void tru_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[ ] )
```

Find a local minimizer of a given function 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 import 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. • -40. 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.
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 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

trut.c, and trutf.c.

5.1.1.7 tru_solve_with_mat()

```
void tru_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 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 import 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.
		• -40. 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 tru_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

[trut.c](#), and [trutf.c](#).

5.1.1.8 tru_solve_without_mat()

```
void tru_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_[], real_wp_[], const real_wp_[], const void *) eval_↩
prec )
```

Find a local minimizer of a given function 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
`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 import 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. • -40. 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`.

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_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_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

[trut.c](), and [trutf.c]().

5.1.1.9 tru_terminate()

```
void tru_terminate (
            void ** data,
            struct tru_control_type * control,
            struct tru_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 tru_control_type)
`out`	*inform*	is a struct containing output information (see tru_inform_type)

Examples

[trut.c](), and [trutf.c]().

Chapter 6

Example Documentation

6.1   trut.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`.

```c
/* trut.c */
/* Full test for the TRU C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "tru.h"
// Custom userdata struct
struct userdata_type {
   double p;
};
// Function prototypes
int fun( int n, const double x[], double *f, const void *);
int grad( int n, const double x[], double g[], const void *);
int hess( int n, int ne, const double x[], double hval[], const void *);
int hess_dense( int n, int ne, const double x[], double hval[], const void *);
int hessprod( int n, const double x[], double u[], const double v[], bool got_h,
           const void *);
int prec( int n, const double x[], double u[], const double v[], const void *);
int fun_diag( int n, const double x[], double *f, const void *);
int grad_diag( int n, const double x[], double g[], const void *);
int hess_diag( int n, int ne, const double x[], double hval[], const void *);
int hessprod_diag( int n, const double x[], double u[], const double v[],
                bool got_h, const void *);
int main(void) {
   // Derived types
   void *data;
   struct tru_control_type control;
   struct tru_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
   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
   double 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 TRU
    tru_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    //control.print_level = 1;
    // Start from 1.5
    double x[] = {1.5,1.5,1.5};
    switch(d){
        case 1: // sparse co-ordinate storage
            st = 'C';
            // control.print_level = 1;
            tru_import( &control, &data, &status, n, "coordinate",
                        ne, H_row, H_col, NULL );
            tru_solve_with_mat( &data, &userdata, &status,
                                n, x, g, ne, fun, grad, hess, prec );
            break;
        case 2: // sparse by rows
            st = 'R';
            tru_import( &control, &data, &status, n, "sparse_by_rows",
                        ne, NULL, H_col, H_ptr);
            tru_solve_with_mat( &data, &userdata, &status,
                                n, x, g, ne, fun, grad, hess, prec );
            break;
        case 3: // dense
            st = 'D';
            tru_import( &control, &data, &status, n, "dense",
                        ne, NULL, NULL, NULL );
            tru_solve_with_mat( &data, &userdata, &status,
                                n, x, g, ne, fun, grad, hess_dense, prec );
            break;
        case 4: // diagonal
            st = 'I';
            tru_import( &control, &data, &status, n, "diagonal",
                        ne, NULL, NULL, NULL );
            tru_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';
            tru_import( &control, &data, &status, n, "absent",
                        ne, NULL, NULL, NULL );
            tru_solve_without_mat( &data, &userdata, &status,
                                   n, x, g, fun, grad, hessprod, prec );
            break;
    }
    tru_information( &data, &inform, &status );
    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: TRU_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
    tru_terminate( &data, &control, &inform );
}
printf("\n tests reverse-communication options\n\n");
// reverse-communication input/output
int eval_status;
double f;
double u[n], v[n];
int index_nz_u[n], index_nz_v[n];
double H_val[ne], H_dense[n*(n+1)/2], H_diag[n];

for( int d=1; d <= 5; d++){
    // Initialize TRU
    tru_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    //control.print_level = 1;
    // Start from 1.5
    double x[] = {1.5,1.5,1.5};
    switch(d){
        case 1: // sparse co-ordinate storage
            st = 'C';
            tru_import( &control, &data, &status, n, "coordinate",
                        ne, H_row, H_col, NULL );
            while(true){ // reverse-communication loop
                tru_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';
    tru_import( &control, &data, &status, n, "sparse_by_rows", ne,
               NULL, H_col, H_ptr );
    while(true){ // reverse-communication loop
        tru_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';
    tru_import( &control, &data, &status, n, "dense",
               ne, NULL, NULL, NULL );
    while(true){ // reverse-communication loop
        tru_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';
    tru_import( &control, &data, &status, n, "diagonal",
               ne, NULL, NULL, NULL );
    while(true){ // reverse-communication loop
        tru_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';
                tru_import( &control, &data, &status, n, "absent",
                            ne, NULL, NULL, NULL );
                while(true){ // reverse-communication loop
                    tru_solve_reverse_without_mat( &data, &status, &eval_status,
                                                   n, x, f, g, 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 == 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{
                        printf(" the value %1i of status should not occur\n",
                          status);
                        break;
                    }
                }
                break;
        }
        tru_information( &data, &inform, &status );
        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: TRU_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
        tru_terminate( &data, &control, &inform );
    }
}
// Objective function
int fun( int n, const double x[], double *f, const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    *f = pow(x[0] + x[2] + p, 2) + pow(x[1] + x[2], 2) + cos(x[0]);
//    *f = pow(x[0] + x[2] + 4.0, 2) + pow(x[1] + x[2], 2) + cos(x[0]);
//        printf("p: %f \n", p);
//        printf("n:   %d \n", n);
//        printf("x: ");
//        for( int i = 0; i < n; i++) printf("%f ", x[i]);
//        printf("\n");
//        printf("objective: %f \n", *f);
    return 0;
}
// Gradient of the objective
int grad( int n, const double x[], double g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    g[0] = 2.0 * ( x[0] + x[2] + p ) - sin(x[0]);
//    g[0] = 2.0 * ( x[0] + x[2] + 4.0 ) - 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] );
//    g[2] = 2.0 * ( x[0] + x[2] + 4.0 ) + 2.0 * ( x[1] + x[2] );
    return 0;
}
// Hessian of the objective
int hess( int n, int ne, const double x[], double 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 double x[], double 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 double x[], double u[], const double 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;
}
// Apply preconditioner
int prec( int n, const double x[], double u[], const double 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 double x[], double *f, const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double 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 double x[], double g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double 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 double x[], double 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 double x[], double u[], const double 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;
}
```

6.2   trutf.c

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

```
/* trutf.c */
/* Full test for the TRU C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "tru.h"
// Custom userdata struct
struct userdata_type {
    double p;
};
// Function prototypes
int fun( int n, const double x[], double *f, const void *);
int grad( int n, const double x[], double g[], const void *);
int hess( int n, int ne, const double x[], double hval[], const void *);
int hess_dense( int n, int ne, const double x[], double hval[], const void *);
int hessprod( int n, const double x[], double u[], const double v[], bool got_h,
              const void *);
```

```c
int prec( int n, const double x[], double u[], const double v[], const void *);
int fun_diag( int n, const double x[], double *f, const void *);
int grad_diag( int n, const double x[], double g[], const void *);
int hess_diag( int n, int ne, const double x[], double hval[], const void *);
int hessprod_diag( int n, const double x[], double u[], const double v[],
                   bool got_h, const void *);
int main(void) {
    // Derived types
    void *data;
    struct tru_control_type control;
    struct tru_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
    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
    double 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 TRU
        tru_initialize( &data, &control, &inform );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        //control.print_level = 1;
        // Start from 1.5
        double x[] = {1.5,1.5,1.5};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                tru_import( &control, &data, &status, n, "coordinate",
                            ne, H_row, H_col, NULL );
                tru_solve_with_mat( &data, &userdata, &status,
                                    n, x, g, ne, fun, grad, hess, prec );
                break;
            case 2: // sparse by rows
                st = 'R';
                tru_import( &control, &data, &status, n, "sparse_by_rows",
                            ne, NULL, H_col, H_ptr );
                tru_solve_with_mat( &data, &userdata, &status,
                                    n, x, g, ne, fun, grad, hess, prec );
                break;
            case 3: // dense
                st = 'D';
                tru_import( &control, &data, &status, n, "dense",
                            ne, NULL, NULL, NULL ) ;
                tru_solve_with_mat( &data, &userdata, &status,
                                    n, x, g, ne, fun, grad, hess_dense, prec );
                break;
            case 4: // diagonal
                st = 'I';
                tru_import( &control, &data, &status, n, "diagonal",
                            ne, NULL, NULL, NULL );
                tru_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';
                tru_import( &control, &data, &status, n, "absent",
                            ne, NULL, NULL, NULL );
                tru_solve_without_mat( &data, &userdata, &status,
                                       n, x, g, fun, grad, hessprod, prec );
                break;
        }
        tru_information( &data, &inform, &status );
        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: TRU_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
        tru_terminate( &data, &control, &inform );
    }
```

```
        printf("\n tests reverse-communication options\n\n");
        // reverse-communication input/output
        int eval_status;
        double f;
        double u[n], v[n];
        int index_nz_u[n], index_nz_v[n];
        double H_val[ne], H_dense[n*(n+1)/2], H_diag[n];

        for( int d=1; d <= 5; d++){
            // Initialize TRU
            tru_initialize( &data, &control, &inform );
            // Set user-defined control options
            control.f_indexing = true; // Fortran sparse matrix indexing
            //control.print_level = 1;
            // Start from 1.5
            double x[] = {1.5,1.5,1.5};
            switch(d){
                case 1: // sparse co-ordinate storage
                    st = 'C';
                    tru_import( &control, &data, &status, n, "coordinate",
                                ne, H_row, H_col, NULL );
                    while(true){ // reverse-communication loop
                        tru_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';
                    tru_import( &control, &data, &status, n, "sparse_by_rows", ne,
                                NULL, H_col, H_ptr );
                    while(true){ // reverse-communication loop
                        tru_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';
                    tru_import( &control, &data, &status, n, "dense",
                                ne, NULL, NULL, NULL );
                    while(true){ // reverse-communication loop
                        tru_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';
                    tru_import( &control, &data, &status, n, "diagonal",
                                ne, NULL, NULL, NULL );
                    while(true){ // reverse-communication loop
                        tru_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';
                    tru_import( &control, &data, &status, n, "absent",
                                ne, NULL, NULL, NULL );
                    while(true){ // reverse-communication loop
                        tru_solve_reverse_without_mat( &data, &status, &eval_status,
                                                       n, x, f, g, 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 == 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{
                            printf(" the value %1i of status should not occur\n",
                              status);
                            break;
                        }
                    }
                    break;
        }
        tru_information( &data, &inform, &status );
        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: TRU_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
        tru_terminate( &data, &control, &inform );
    }
}
// Objective function
int fun( int n, const double x[], double *f, const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double 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
```

```c
int grad( int n, const double x[], double g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double 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 double x[], double 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 double x[], double 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 double x[], double u[], const double 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;
}
// Apply preconditioner
int prec( int n, const double x[], double u[], const double 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 double x[], double *f, const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double 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 double x[], double g[], const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double 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 double x[], double 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 double x[], double u[], const double 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;
}
```

Index