C interfaces to GALAHAD BLLS

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Mon May 1 2023

1 GALAHAD C package arc 1

1.1 Introduction 1

1.1.1 Purpose 1

1.1.2 Authors 1

1.1.3 Originally released 1

1.1.4 Terminology 1

1.1.5 Method 2

1.1.6 References 2

1.2 Call order 2

1.3 Symmetric matrix storage formats 3

1.3.1 Dense storage format 3

1.3.2 Sparse co-ordinate storage format 3

1.3.3 Sparse row-wise storage format 3

2 File Index 5

2.1 File List 5

3 File Documentation 7

3.1 galahad_arc.h File Reference 7

3.1.1 Data Structure Documentation 8

3.1.1.1 struct arc_control_type 8

3.1.1.2 struct arc_time_type 11

3.1.1.3 struct arc_inform_type 12

3.1.2 Function Documentation 12

3.1.2.1 arc_initialize() 12

3.1.2.2 arc_read_specfile() 13

3.1.2.3 arc_import() 13

3.1.2.4 arc_reset_control() 14

3.1.2.5 arc_solve_with_mat() 15

3.1.2.6 arc_solve_without_mat() 17

3.1.2.7 arc_solve_reverse_with_mat() 19

3.1.2.8 arc_solve_reverse_without_mat() 22

3.1.2.9 arc_information() 25

3.1.2.10 arc_terminate() 25

4 Example Documentation 27

4.1 arct.c 27

4.2 arctf.c 31

Chapter 1

GALAHAD C package blls

1.1 Introduction

1.1.1 Purpose

This package uses a preconditioned, projected-gradient method to solve the **bound-constrained regularized linear least-squares problem**

$$\mbox{minimize}\;\; r(x) = q(x) + \frac{1}{2}\sigma \|x\|^2$$

, \;\;\mbox{where}\;\; q(x) = \frac{1}{2} | A x - b|_2^2 subject to the simple bound constraints

$$x_j^l \leq x_j \leq x_j^u, \quad j = 1, \ldots, n,$$

where the m by n real matrix A, the vectors b, x^l, x^u and the non-negative weight σ are given. Any of the constraint bounds x_j^l and x_j^u may be infinite. Full advantage is taken of any zero coefficients of the Jacobian matrix A of the **residuals** $c(x) = Ax - b$; the matrix need not be provided as there are options to obtain matrix-vector products involving A and its transpose either by reverse communication or from a user-provided subroutine.

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

October 2019, C interface March 2022.

1.1.4 Terminology

The required solution x necessarily satisfies the primal optimality conditions

$$x^l \leq x \leq x^u,$$

the dual optimality conditions

$$(A^T A + \sigma I)x = A^T b + z$$

where

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

and the complementary slackness conditions

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

where the vector z is known as the dual variables for the bounds, respectively, and where the vector inequalities hold component-wise.

1.1.5 Method

The method is iterative. Each iteration proceeds in two stages. Firstly, a search direction s from the current estimate of the solution x is computed. This may be in a scaled steepest-descent direction, or, if the working set of variables on bounds has not changed dramatically, in a direction that provides an approximate minimizer of the objective over a subspace comprising the currently free-variables. The latter is computed either using an appropriate sparse factorization by the GALAHAD package SBLS, or by the conjugate-gradient least-squares (CGLS) method; tt may be necessary to regularize the subproblem very slightly to avoid a ill-posedness. Thereafter, a piecewise linesearch (arc search) is carried out along the arc $x(\alpha) = P(x + \alpha s)$ for $\alpha > 0$, where the projection operator is defined component-wise at any feasible point v to be

$$P_j(v) = \min(\max(x_j, x_j^l), x_j^u);$$

thus this arc bends the search direction into the feasible region. The arc search is performed either exactly, by passing through a set of increasing breakpoints at which it changes direction, or inexactly, by evaluating a sequence of different α on the arc. All computation is designed to exploit sparsity in A.

1.1.6 Reference

Full details are provided in

N. I. M. Gould (2022). Numerical methods for solving bound-constrained linear least squares problems. In preparation.

1.1.7 Call order

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

- blls_initialize - provide default control parameters and set up initial data structures
- blls_read_specfile (optional) - override control values by reading replacement values from a file
- set up problem data structures and fixed values by caling one of
  - blls_import - in the case that A is explicitly available

    - blls_import_without_a - in the case that only the effect of applying A and its transpose to a vector is possible

- blls_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
    - blls_solve_given_a - solve the problem using values of A
    - blls_solve_reverse_a_prod - solve the problem by returning to the caller for products of A and its transpose with specified vectors
- blls_information (optional) - recover information about the solution and solution process
- blls_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.8 Unsymmetric matrix storage formats

The unsymmetric m by n matrix A may be presented and stored in a variety of convenient input formats.

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.

1.1.8.1 Dense row storage format

The matrix A 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. In this case, component $n * i + j$ of the storage array A_val will hold the value A_{ij} for $0 \leq i \leq m-1, 0 \leq j \leq n-1$.

1.1.8.2 Dense column storage format

The matrix A is stored as a compact dense matrix by columns, that is, the values of the entries of each column in turn are stored in order within an appropriate real one-dimensional array. In this case, component $m * j + i$ of the storage array A_val will hold the value A_{ij} for $0 \leq i \leq m-1, 0 \leq j \leq n-1$.

1.1.8.3 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 A, its row index i, column index j and value A_{ij}, $0 \leq i \leq m-1$, $0 \leq j \leq n-1$, are stored as the l-th components of the integer arrays A_row and A_col and real array A_val, respectively, while the number of nonzeros is recorded as A_ne = ne.

1.1.8.4 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 A the i-th component of the integer array A_ptr holds the position of the first entry in this row, while A_ptr(m) holds the total number of entries. The column indices j, $0 \leq j \leq n-1$, and values A_{ij} of the nonzero entries in the i-th row are stored in components l = A_ptr(i), . . ., A_ptr(i+1)-1, $0 \leq i \leq m-1$, of the integer array A_col, and real array A_val, respectively. For sparse matrices, this scheme almost always requires less storage than its predecessors.

1.1.8.5 Sparse column-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in column j appear directly before those in column j+1. For the j-th column of A the j-th component of the integer array A_ptr holds the position of the first entry in this column, while A_ptr(n) holds the total number of entries. The row indices i, $0 \leq i \leq m-1$, and values A_{ij} of the nonzero entries in the j-th column are stored in components l = A_ptr(j), . . ., A_ptr(j+1)-1, $0 \leq j \leq n-1$, of the integer array A_row, and real array A_val, respectively. Once again, for sparse matrices, this scheme almost always requires less storage than the dense of coordinate formats.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

galahad_blls.h . . . . **??**

Chapter 3

File Documentation

3.1 galahad_blls.h File Reference

```
#include <stdbool.h>
#include <stdint.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_sbls.h"
#include "galahad_convert.h"
```

Data Structures

- struct blls_control_type
- struct blls_time_type
- struct blls_inform_type

Functions

- void blls_initialize (void **data, struct blls_control_type *control, int *status)
- void blls_read_specfile (struct blls_control_type *control, const char specfile[ ])
- void blls_import (struct blls_control_type *control, void **data, int *status, int n, int m, const char A_type[ ], int A_ne, const int A_row[ ], const int A_col[ ], const int A_ptr[ ])
- void blls_import_without_a (struct blls_control_type *control, void **data, int *status, int n, int m)
- void blls_reset_control (struct blls_control_type *control, void **data, int *status)
- void blls_solve_given_a (void **data, void *userdata, int *status, int n, int m, int A_ne, const real_wp_ A_val[ ], const real_wp_ b[ ], const real_wp_ x_l[ ], const real_wp_ x_u[ ], real_wp_ x[ ], real_wp_ z[ ], real_wp_ c[ ], real_wp_ g[ ], int x_stat[ ], const real_wp_ w[ ], int(*eval_prec)(int, const real_wp_[ ], real_wp_[ ], const void *))
- void blls_solve_reverse_a_prod (void **data, int *status, int *eval_status, int n, int m, const real_wp_ b[ ], const real_wp_ x_l[ ], const real_wp_ x_u[ ], real_wp_ x[ ], real_wp_ z[ ], real_wp_ c[ ], real_wp_ g[ ], int x_stat[ ], real_wp_ v[ ], const real_wp_ p[ ], int nz_v[ ], int *nz_v_start, int *nz_v_end, const int nz_p[ ], int nz_p_end, const real_wp_ w[ ])
- void blls_information (void **data, struct blls_inform_type *inform, int *status)
- void blls_terminate (void **data, struct blls_control_type *control, struct blls_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct blls_control_type

control derived type as a C struct

Examples

bllst.c, and bllstf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	error	unit number for error and warning diagnostics
int	out	general output unit number
int	print_level	the level of output required
int	start_print	on which iteration to start printing
int	stop_print	on which iteration to stop printing
int	print_gap	how many iterations between printing
int	maxit	how many iterations to perform (-ve reverts to HUGE(1)-1)
int	cold_start	cold_start should be set to 0 if a warm start is required (with variable assigned according to X_stat, see below), and to any other value if the values given in prob.X suffice
int	preconditioner	the preconditioner (scaling) used. Possible values are: /li 0. no preconditioner. /li 1. a diagonal preconditioner that normalizes the rows of A. /li anything else. a preconditioner supplied by the user either via a subroutine call of eval_prec} or via reverse communication.
int	ratio_cg_vs_sd	the ratio of how many iterations use CGLS rather than steepest descent
int	change_max	the maximum number of per-iteration changes in the working set permitted when allowing CGLS rather than steepest descent
int	cg_maxit	how many CG iterations to perform per BLLS iteration (-ve reverts to n+1)
int	arcsearch_max_steps	the maximum number of steps allowed in a piecewise arcsearch (-ve=infini
int	sif_file_device	the unit number to write generated SIF file describing the current probl
real_wp_	weight	the value of the non-negative regularization weight sigma, i.e., the quadratic objective function q(x) will be regularized by adding 1/2 weight $\|\|x\|\|$^2; any value smaller than zero will be regarded as zero.
real_wp_	infinity	any bound larger than infinity in modulus will be regarded as infinite
real_wp_	stop_d	the required accuracy for the dual infeasibility
real_wp_	identical_bounds_tol	any pair of constraint bounds (x_l,x_u) that are closer than identical_bounds_tol will be reset to the average of their values

Data Fields

real_wp_	stop_cg_relative	the CG iteration will be stopped as soon as the current norm of the preconditioned gradient is smaller than max(stop_cg_relative ∗ initial preconditioned gradient, stop_cg_absolute)
real_wp_	stop_cg_absolute	
real_wp_	alpha_max	the largest permitted arc length during the piecewise line search
real_wp_	alpha_initial	the initial arc length during the inexact piecewise line search
real_wp_	alpha_reduction	the arc length reduction factor for the inexact piecewise line search
real_wp_	arcsearch_acceptance_tol	the required relative reduction during the inexact piecewise line search
real_wp_	stabilisation_weight	the stabilisation weight added to the search-direction subproblem
real_wp_	cpu_time_limit	the maximum CPU time allowed (-ve = no limit)
bool	direct_subproblem_solve	direct_subproblem_solve is true if the least-squares subproblem is to be solved using a matrix factorization, and false if conjugate gradients are to be preferred
bool	exact_arc_search	exact_arc_search is true if an exact arc_search is required, and false if an approximation suffices
bool	advance	advance is true if an inexact exact arc_search can increase steps as well as decrease them
bool	space_critical	if space_critical is true, every effort will be made to use as little space as possible. This may result in longer computation times
bool	deallocate_error_fatal	if deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue
bool	generate_sif_file	if generate_sif_file is true, a SIF file describing the current problem will be generated
char	sif_file_name[31]	name (max 30 characters) of generated SIF file containing input problem
char	prefix[31]	all output lines will be prefixed by a string (max 30 characters) prefix(2:LEN(TRIM(.prefix))-1) where prefix contains the required string enclosed in quotes, e.g. "string" or 'string'
struct sbls_control_type	sbls_control	control parameters for SBLS
struct convert_control_type	convert_control	control parameters for CONVERT

3.1.1.2 struct blls_time_type

time derived type as a C struct

Data Fields

real_wp_	total	the total CPU time spent in the package
real_wp_	analyse	the CPU time spent analysing the required matrices prior to factorization
real_wp_	factorize	the CPU time spent factorizing the required matrices

Data Fields

real_wp_	solve	the CPU time spent in the linear solution phase
real_wp_	clock_total	the total clock time spent in the package
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 in the linear solution phase

3.1.1.3 struct blls_inform_type

inform derived type as a C struct

Examples

bllst.c, and bllstf.c.

Data Fields

int	status	reported return status.
int	alloc_status	Fortran STAT value after allocate failure.
int	factorization_status	status return from factorization
int	iter	number of iterations required
int	cg_iter	number of CG iterations required
real_wp_	obj	current value of the objective function, r(x).
real_wp_	norm_pg	current value of the Euclidean norm of projected gradient of r(x).
char	bad_alloc[81]	name of array which provoked an allocate failure
struct blls_time_type	time	times for various stages
struct sbls_inform_type	sbls_inform	inform values from SBLS
struct convert_inform_type	convert_inform	inform values for CONVERT

3.1.2 Function Documentation

3.1.2.1 blls_initialize()

```
void blls_initialize (
            void ** data,
            struct blls_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 blls_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

bllst.c, and bllstf.c.

3.1.2.2 blls_read_specfile()

```
void blls_read_specfile (
            struct blls_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 RUNBLLS.SPC and lie in the current directory. Refer to Table 2.1 in the fortran documentation provided in $GALAHAD/doc/blls.pdf for a list of keywords that may be set.

Parameters

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

3.1.2.3 blls_import()

```
void blls_import (
            struct blls_control_type * control,
            void ** data,
            int * status,
            int n,
            int m,
            const char A_type[],
            int A_ne,
            const int A_row[],
            const int A_col[],
            const int A_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 bls_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 restrictions n $>$ 0, m $>$ 0 or requirement that type contains its relevant string 'coordinate', 'sparse_by_rows', 'sparse_by_columns', 'dense_by_rows', or 'dense_by_columns'; has been violated.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of residuals.
in	*A_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Jacobian A. It should be one of 'coordinate', 'sparse_by_rows', 'sparse_by_columns', 'dense_by_rows', or 'dense_by_columns'; lower or upper case variants are allowed.
in	*A_ne*	is a scalar variable of type int, that holds the number of entries in A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*A_row*	is a one-dimensional array of size A_ne and type int, that holds the row indices of A in the sparse co-ordinate or sparse column-wise storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
in	*A_col*	is a one-dimensional array of size A_ne and type int, that holds the column indices of A in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
in	*A_ptr*	is a one-dimensional array of size n+1 or m+1 and type int, that holds the starting position of each row of A, as well as the total number of entries, in the sparse row-wise storage scheme, or the starting position of each column of A, as well as the total number of entries, in the sparse column-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.

Examples

bllst.c, and bllstf.c.

3.1.2.4 blls_import_without_a()

```
void blls_import_without_a (
            struct blls_control_type * control,
```

```
void ** data,
int * status,
int n,
int m )
```

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 blls_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 m $>$ 0 has been violated.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of residuals.

Examples

bllst.c, and bllstf.c.

3.1.2.5 blls_reset_control()

```
void blls_reset_control (
            struct blls_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 blls_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

3.1.2.6 blls_solve_given_a()

```
void blls_solve_given_a (
            void ** data,
            void * userdata,
            int * status,
            int n,
            int m,
            int A_ne,
            const real_wp_ A_val[ ],
            const real_wp_ b[ ],
            const real_wp_ x_l[ ],
            const real_wp_ x_u[ ],
            real_wp_ x[ ],
            real_wp_ z[ ],
            real_wp_ c[ ],
            real_wp_ g[ ],
            int x_stat[ ],
            const real_wp_ w[ ],
            int(*)(int, const real_wp_[], real_wp_[], const void *) eval_prec )
```

Solve the bound-constrained linear least-squares problem when the Jacobian A is available.

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 restrictions n $>$ 0, m $>$ 0 or requirement that a type contains its relevant string 'coordinate', 'sparse_by_rows', 'sparse_by_columns', 'dense_by_rows' or 'dense_by_columns' has been violated. • -4. The simple-bound constraints are inconsistent. • -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. • -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.
in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of residuals.
in	*A_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	*A_val*	is a one-dimensional array of size A_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	*b*	is a one-dimensional array of size m and type double, that holds the constant term b in the residuals. The i-th component of b, i = 0, ... , m-1, contains b_i.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x^l_j.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x^l_j.
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	*z*	is a one-dimensional array of size n and type double, that holds the values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the values of the residuals $c = Ax - b$. The i-th component of c, i = 0, ... , m-1, contains c_i.

Parameters

out	*g*	is a one-dimensional array of size n and type double, that holds the values of the gradient $g = A^T c$. The j-th component of g, j = 0, ... , n-1, contains g_j.
in,out	*x_stat*	is a one-dimensional array of size n and type int, that gives the optimal status of the problem variables. If x_stat(j) is negative, the variable x_j most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.
in	*w*	is an optional one-dimensional array of size m and type double, that holds the values w of the weights on the residuals in the least-squares objective function. It need not be set if the weights are all ones, and in this case can be NULL.
	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 v[], double p[], const void *userdata )` The product $p = P^{-1} v$ involving the user's preconditioner P with the vector v = v, the result p must be retured in p, and the function return value set to 0. If the evaluation is impossible, return should be set to a nonzero value. Data may be passed into `eval_prec` via the structure `userdata`.

Examples

bllst.c, and bllstf.c.

3.1.2.7 blls_solve_reverse_a_prod()

```
void blls_solve_reverse_a_prod (
            void ** data,
            int * status,
            int * eval_status,
            int n,
            int m,
            const real_wp_ b[ ],
            const real_wp_ x_l[ ],
            const real_wp_ x_u[ ],
            real_wp_ x[ ],
            real_wp_ z[ ],
            real_wp_ c[ ],
            real_wp_ g[ ],
            int x_stat[ ],
            real_wp_ v[ ],
            const real_wp_ p[ ],
            int nz_v[ ],
            int * nz_v_start,
            int * nz_v_end,
            const int nz_p[ ],
            int nz_p_end,
            const real_wp_ w[ ] )
```

Solve the bound-constrained linear least-squares problem when the products of the Jacobian A and its transpose with specified vectors may be computed by the calling program.

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. 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 a type contains its relevant string 'coordinate', 'sparse_by_rows', 'sparse_by_columns', 'dense_by_rows' or 'dense_by_columns' has been violated. • -4. The simple-bound constraints are inconsistent. • -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. • -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.

Parameters

	status	(continued) • 2. The product Av of the residual Jacobian A with a given output vector v is required from the user. The vector v will be stored in v and the product Av must be returned in p, status_eval should be set to 0, and blls_solve_reverse_a_prod re-entered with all other arguments unchanged. If the product cannot be formed, v need not be set, but blls_solve_reverse_a_prod should be re-entered with eval_status set to a nonzero value. • 3. The product $A^T v$ of the transpose of the residual Jacobian A with a given output vector v is required from the user. The vector v will be stored in v and the product $A^T v$ must be returned in p, status_eval should be set to 0, and blls_solve_reverse_a_prod re-entered with all other arguments unchanged. If the product cannot be formed, v need not be set, but blls_solve_reverse_a_prod should be re-entered with eval_status set to a nonzero value. • 4. The product Av of the residual Jacobian A with a given sparse output vector v is required from the user. The nonzero components of the vector v will be stored as entries nz_in[nz_in_start-1:nz_in_end-1] of v and the product Av must be returned in p, status_eval should be set to 0, and blls_solve_reverse_a_prod re-entered with all other arguments unchanged; The remaining components of v should be ignored. If the product cannot be formed, v need not be set, but blls_solve_reverse_a_prod should be re-entered with eval_status set to a nonzero value. • 5. The nonzero components of the product Av of the residual Jacobian A with a given sparse output vector v is required from the user. The nonzero components of the vector v will be stored as entries nz_in[nz_in_start-1:nz_in_end-1] of v; the remaining components of v should be ignored. The resulting **nonzeros** in the product Av must be placed in their appropriate comnponents of p, while a list of indices of the nonzeros placed in nz_out[0 : nz_out_end-1] and the number of nonzeros recorded in nz_out_end. Additionally, status_eval should be set to 0, and blls_solve_reverse_a_prod re-entered with all other arguments unchanged. If the product cannot be formed, v, nz_out_end and nz_out need not be set, but blls_solve_reverse_a_prod should be re-entered with eval_status set to a nonzero value. • 6. A subset of the product $A^T v$ of the transpose of the residual Jacobian A with a given output vector v is required from the user. The vector v will be stored in v and components nz_in[nz_in_start-1:nz_in_end-1] of the product $A^T v$ must be returned in the relevant components of p (the remaining components should not be set), status_eval should be set to 0, and blls_solve_reverse_a_prod re-entered with all other arguments unchanged. If the product cannot be formed, v need not be set, but blls_solve_reverse_a_prod should be re-entered with eval_status set to a nonzero value. • 7. The product $P^{-1} v$ of the inverse of the preconditioner P with a given output vector v is required from the user. The vector v will be stored in v and the product $P^{-1} v$ must be returned in p, status_eval should be set to 0, and blls_solve_reverse_a_prod re-entered with all other arguments unchanged. If the product cannot be formed, v need not be set, but blls_solve_reverse_a_prod should be re-entered with eval_status set to a nonzero value. This value of status can only occur if the user has set control.preconditioner = 2.

Parameters

in,out	*eval_status*	is a scalar variable of type int, that is used to indicate if the matrix products can be provided (see `status` above)
in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of residuals.
in	*b*	is a one-dimensional array of size m and type double, that holds the constant term b in the residuals. The i-th component of b, i = 0, ... , m-1, contains b_i.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x^l_j.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x^l_j.
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.
out	*c*	is a one-dimensional array of size m and type double, that holds the values of the residuals $c = Ax - b$. The i-th component of c, i = 0, ... , m-1, contains c_i.
out	*g*	is a one-dimensional array of size n and type double, that holds the values of the gradient $g = A^T c$. The j-th component of g, j = 0, ... , n-1, contains g_j.
in,out	*z*	is a one-dimensional array of size n and type double, that holds the values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
in,out	*x_stat*	is a one-dimensional array of size n and type int, that gives the optimal status of the problem variables. If x_stat(j) is negative, the variable x_j most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.
out	*v*	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=2-4 above for details).
in	*p*	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=2-4 above for details).
out	*nz_v*	is a one-dimensional array of size n and type int, that is used for reverse communication (see status=3-4 above for details).
out	*nz_v_start*	is a scalar of type int, that is used for reverse communication (see status=3-4 above for details).
out	*nz_v_end*	is a scalar of type int, that is used for reverse communication (see status=3-4 above for details).
in	*nz_p*	is a one-dimensional array of size n and type int, that is used for reverse communication (see status=4 above for details).
in	*nz_p_end*	is a scalar of type int, that is used for reverse communication (see status=4 above for details).
in	*w*	is an optional one-dimensional array of size m and type double, that holds the values w of the weights on the residuals in the least-squares objective function. It need not be set if the weights are all ones, and in this case can be NULL.

Examples

bllst.c, and bllstf.c.

3.1.2.8 blls_information()

```
void blls_information (
            void ** data,
```

```
            struct blls_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 blls_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

bllst.c, and bllstf.c.

3.1.2.9 blls_terminate()

```
void blls_terminate (
            void ** data,
            struct blls_control_type * control,
            struct blls_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 blls_control_type)
out	*inform*	is a struct containing output information (see blls_inform_type)

Examples

bllst.c, and bllstf.c.

Chapter 4

Example Documentation

4.1 bllst.c

This is an example of how to use the package to solve a bound-constrained linear least-squares problem. A variety of supported Jacobian storage formats are shown. An example of preconditioning, in this case with the identity matrix which actually achieves nothing, is also illustrated.

Notice that C-style indexing is used, and that this is flaggeed by setting `control.f_indexing` to `false`.

```
/* bllst.c */
/* Full test for the BLLS C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_blls.h"
// define max
#define max(a,b)              \
({                            \
    __typeof__ (a) _a = (a); \
    __typeof__ (b) _b = (b); \
    _a > _b ? _a : _b;        \
})
// Custom userdata struct
struct userdata_type {
   real_wp_ scale;
};
// Function prototypes
int prec( int n, const real_wp_ v[], real_wp_ p[], const void * );
int main(void) {
    // Derived types
    void *data;
    struct blls_control_type control;
    struct blls_inform_type inform;
    // Set user data
    struct userdata_type userdata;
    userdata.scale = 1.0;
    // Set problem data
    int n = 10; // dimension
    int m = n + 1; // number of residuals
    int A_ne = 2 * n; // sparse Jacobian elements
    int A_dense_ne = m * n; // dense Jacobian elements
    // row-wise storage
    int A_row[A_ne]; // row indices,
    int A_col[A_ne]; // column indices
    int A_ptr[m+1];  // row pointers
    real_wp_ A_val[A_ne]; // values
    real_wp_ A_dense[A_dense_ne]; // dense values
    // column-wise storage
    int A_by_col_row[A_ne]; // row indices,
    int A_by_col_ptr[n+1];  // column pointers
    real_wp_ A_by_col_val[A_ne]; // values
    real_wp_ A_by_col_dense[A_dense_ne]; // dense values
    real_wp_ b[m];  // linear term in the objective
```

```
real_wp_ x_l[n]; // variable lower bound
real_wp_ x_u[n]; // variable upper bound
real_wp_ x[n]; // variables
real_wp_ z[n]; // dual variables
real_wp_ c[m]; // residual
real_wp_ g[n]; // gradient
real_wp_ w[m]; // weights
// Set output storage
int x_stat[n]; // variable status
char st[3];
int i, l, status;
x_l[0] = -1.0;
for( int i = 1; i < n; i++) x_l[i] = - INFINITY;
x_u[0] = 1.0;
x_u[1] = INFINITY;
for( int i = 2; i < n; i++) x_u[i] = 2.0;
//   A = (  I  )  and b = ( i * e )
//       ( e^T )          ( n + 1 )
for( int i = 0; i < n; i++) b[i] = i + 1;
b[n] = n+1;
w[0] = 2.0;
for( int i = 1; i < m; i++) w[i] = 1.0;
// A by rows
for( int i = 0; i < n; i++)
{
  A_ptr[i] = i;
  A_row[i] = i; A_col[i] = i; A_val[i] = 1.0;
}
A_ptr[n] = n;
for( int i = 0; i < n; i++)
{
  A_row[n+i] = n; A_col[n+i] = i; A_val[n+i] = 1.0;
}
A_ptr[m] = A_ne;
l = - 1;
for( int i = 0; i < n; i++)
{
  for( int j = 0; j < n; j++)
  {
    l = l + 1;
    if ( i == j ) {
      A_dense[l] = 1.0;
    }
    else {
      A_dense[l] = 0.0;
    }
  }
}
for( int j = 0; j < n; j++)
{
  l = l + 1;
  A_dense[l] = 1.0;
}
// A by columns
l = - 1;
for( int j = 0; j < n; j++)
{
  l = l + 1;  A_by_col_ptr[j] = l ;
  A_by_col_row[l] = j ; A_by_col_val[l] = 1.0;
  l = l + 1;
  A_by_col_row[l] = n ; A_by_col_val[l] = 1.0;
}
A_by_col_ptr[n] = A_ne;
l = - 1;
for( int j = 0; j < n; j++)
{
  for( int i = 0; i < n; i++)
  {
    l = l + 1;
    if ( i == j ) {
      A_by_col_dense[l] = 1.0;
    }
    else {
      A_by_col_dense[l] = 0.0;
    }
  }
  l = l + 1;
  A_by_col_dense[l] = 1.0;
}
printf(" C sparse matrix indexing\n\n");
printf(" basic tests of blls storage formats\n\n");
for( int d=1; d <= 5; d++){
    // Initialize BLLS
    blls_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    //control.print_level = 1;
```

```
    // Start from 0
    for( int i = 0; i < n; i++) x[i] = 0.0;
    for( int i = 0; i < n; i++) z[i] = 0.0;
    switch(d){
        case 1: // sparse co-ordinate storage
            strcpy( st, "CO" );
            blls_import( &control, &data, &status, n, m,
                         "coordinate", A_ne, A_row, A_col, NULL );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_ne, A_val, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        case 2: // sparse by rows
            strcpy( st, "SR" );
            blls_import( &control, &data, &status, n, m,
                          "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_ne, A_val, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        case 3: // dense by rows
            strcpy( st, "DR" );
            blls_import( &control, &data, &status, n, m,
                          "dense_by_rows", A_dense_ne, NULL, NULL, NULL );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_dense_ne, A_dense, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        case 4: // sparse by columns
            strcpy( st, "SC" );
            blls_import( &control, &data, &status, n, m,
                          "sparse_by_columns", A_ne, A_by_col_row,
                          NULL, A_by_col_ptr );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_ne, A_by_col_val, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        case 5: // dense by columns
            strcpy( st, "DC" );
            blls_import( &control, &data, &status, n, m,
                          "dense_by_columns", A_dense_ne, NULL, NULL, NULL );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_dense_ne, A_by_col_dense, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        }
    blls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%s:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%s: BLLS_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
    blls_terminate( &data, &control, &inform );
}
printf("\n tests reverse-communication options\n\n");
// reverse-communication input/output
int nm;
nm = max( n, m );
int eval_status, nz_v_start, nz_v_end, nz_p_end;
int nz_v[nm], nz_p[m], mask[m];
real_wp_ v[nm], p[nm];
nz_p_end = 0;
// Initialize BLLS
blls_initialize( &data, &control, &status );
// Set user-defined control options
control.f_indexing = false; // C sparse matrix indexing
// control.print_level = 1;
// Start from 0
for( int i = 0; i < n; i++) x[i] = 0.0;
for( int i = 0; i < n; i++) z[i] = 0.0;
strcpy( st, "RC" );
for( int i = 0; i < m; i++) mask[i] = 0;
blls_import_without_a( &control, &data, &status, n, m ) ;
while(true){ // reverse-communication loop
    blls_solve_reverse_a_prod( &data, &status, &eval_status, n, m, b,
                               x_l, x_u, x, z, c, g, x_stat, v, p,
                               nz_v, &nz_v_start, &nz_v_end,
                               nz_p, nz_p_end, w );
```

```
        if(status == 0){ // successful termination
            break;
        }else if(status < 0){ // error exit
            break;
        }else if(status == 2){ // evaluate p = Av
          p[n]=0.0;
          for( int i = 0; i < n; i++){
            p[i] = v[i];
            p[n] = p[n] + v[i];
          }
        }else if(status == 3){ // evaluate p = A^Tv
          for( int i = 0; i < n; i++) p[i] = v[i] + v[n];
        }else if(status == 4){ // evaluate p = Av for sparse v
          p[n]=0.0;
          for( int i = 0; i < n; i++) p[i] = 0.0;
          for( int l = nz_v_start - 1; l < nz_v_end; l++){
            i = nz_v[l];
            p[i] = v[i];
            p[n] = p[n] + v[i];
          }
        }else if(status == 5){ // evaluate p = sparse Av for sparse v
          nz_p_end = 0;
          for( int l = nz_v_start - 1; l < nz_v_end; l++){
            i = nz_v[l];
            if (mask[i] == 0){
              mask[i] = 1;
              nz_p[nz_p_end] = i;
              nz_p_end = nz_p_end + 1;
              p[i] = v[i];
            }
            if (mask[n] == 0){
              mask[n] = 1;
              nz_p[nz_p_end] = n;
              nz_p_end = nz_p_end + 1;
              p[n] = v[i];
            }else{
              p[n] = p[n] + v[i];
            }
          }
          for( int l = 0; l < nz_p_end; l++) mask[nz_p[l]] = 0;
        }else if(status == 6){ // evaluate p = sparse A^Tv
          for( int l = nz_v_start - 1; l < nz_v_end; l++){
            i = nz_v[l];
            p[i] = v[i] + v[n];
          }
        }else if(status == 7){ // evaluate p = P^{-}v
          for( int i = 0; i < n; i++) p[i] = userdata.scale * v[i];
        }else{
            printf(" the value %1i of status should not occur\n", status);
            break;
        }
        eval_status = 0;
    }
    // Record solution information
    blls_information( &data, &inform, &status );
    // Print solution details
    if(inform.status == 0){
        printf("%s:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%s: BLLS_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
    blls_terminate( &data, &control, &inform );
}
// Apply preconditioner
int prec( int n, const real_wp_ v[], real_wp_ p[], const void *userdata ){
  struct userdata_type *myuserdata = (struct userdata_type *) userdata;
  real_wp_ scale = myuserdata->scale;
  for( int i = 0; i < n; i++) p[i] = scale * v[i];
   return 0;
}
```

4.2 bllstf.c

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

```
/* bllstf.c */
/* Full test for the BLLS C interface using fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_blls.h"
// define max
#define max(a,b)              \
({                            \
    __typeof__ (a) _a = (a); \
    __typeof__ (b) _b = (b); \
    _a > _b ? _a : _b;       \
})
// Custom userdata struct
struct userdata_type {
   real_wp_ scale;
};
// Function prototypes
int prec( int n, const real_wp_ v[], real_wp_ p[], const void * );
int main(void) {
    // Derived types
    void *data;
    struct blls_control_type control;
    struct blls_inform_type inform;
    // Set user data
    struct userdata_type userdata;
    userdata.scale = 1.0;
    // Set problem data
    int n = 10; // dimension
    int m = n + 1; // number of residuals
    int A_ne = 2 * n; // sparse Jacobian elements
    int A_dense_ne = m * n; // dense Jacobian elements
    // row-wise storage
    int A_row[A_ne]; // row indices,
    int A_col[A_ne]; // column indices
    int A_ptr[m+1];  // row pointers
    real_wp_ A_val[A_ne]; // values
    real_wp_ A_dense[A_dense_ne]; // dense values
    // column-wise storage
    int A_by_col_row[A_ne]; // row indices,
    int A_by_col_ptr[n+1];  // column pointers
    real_wp_ A_by_col_val[A_ne]; // values
    real_wp_ A_by_col_dense[A_dense_ne]; // dense values
    real_wp_ b[m];  // linear term in the objective
    real_wp_ x_l[n]; // variable lower bound
    real_wp_ x_u[n]; // variable upper bound
    real_wp_ x[n]; // variables
    real_wp_ z[n]; // dual variables
    real_wp_ c[m]; // residual
    real_wp_ g[n]; // gradient
    real_wp_ w[m]; // weights
    // Set output storage
    int x_stat[n]; // variable status
    char st[3];
    int i, l, status;
    x_l[0] = -1.0;
    for( int i = 1; i < n; i++) x_l[i] = - INFINITY;
    x_u[0] = 1.0;
    x_u[1] = INFINITY;
    for( int i = 2; i < n; i++) x_u[i] = 2.0;
    //   A = (  I  )  and b = ( i * e )
    //       ( e^T )          ( n + 1 )
    for( int i = 0; i < n; i++) b[i] = i + 1;
    b[n] = n+1;
    w[0] = 2.0;
    for( int i = 1; i < m; i++) w[i] = 1.0;
    // A by rows
    for( int i = 0; i < n; i++)
    {
      A_ptr[i] = i + 1;
      A_row[i] = i + 1; A_col[i] = i + 1; A_val[i] = 1.0;
    }
    A_ptr[n] = n + 1;
    for( int i = 0; i < n; i++)
    {
      A_row[n+i] = m; A_col[n+i] = i + 1; A_val[n+i] = 1.0;
    }
    A_ptr[m] = A_ne + 1;
    l = - 1;
```

```c
for( int i = 0; i < n; i++)
{
  for( int j = 0; j < n; j++)
  {
    l = l + 1;
    if ( i == j ) {
      A_dense[l] = 1.0;
    }
    else {
      A_dense[l] = 0.0;
    }
  }
}
for( int j = 0; j < n; j++)
{
  l = l + 1;
  A_dense[l] = 1.0;
}
// A by columns
l = - 1;
for( int j = 0; j < n; j++)
{
  l = l + 1;  A_by_col_ptr[j] = l + 1;
  A_by_col_row[l] = j + 1; A_by_col_val[l] = 1.0;
  l = l + 1;
  A_by_col_row[l] = m; A_by_col_val[l] = 1.0;
}
A_by_col_ptr[n] = A_ne + 1;
l = - 1;
for( int j = 0; j < n; j++)
{
  for( int i = 0; i < n; i++)
  {
    l = l + 1;
    if ( i == j ) {
      A_by_col_dense[l] = 1.0;
    }
    else {
      A_by_col_dense[l] = 0.0;
    }
  }
  l = l + 1;
  A_by_col_dense[l] = 1.0;
}
printf(" fortran sparse matrix indexing\n\n");
printf(" basic tests of blls storage formats\n\n");
for( int d=1; d <= 5; d++){
    // Initialize BLLS
    blls_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = true; // fortran sparse matrix indexing
    // Start from 0
    for( int i = 0; i < n; i++) x[i] = 0.0;
    for( int i = 0; i < n; i++) z[i] = 0.0;
    switch(d){
        case 1: // sparse co-ordinate storage
            strcpy( st, "CO" );
            blls_import( &control, &data, &status, n, m,
                        "coordinate", A_ne, A_row, A_col, NULL );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_ne, A_val, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        case 2: // sparse by rows
            strcpy( st, "SR" );
            blls_import( &control, &data, &status, n, m,
                         "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_ne, A_val, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        case 3: // dense by rows
            strcpy( st, "DR" );
            blls_import( &control, &data, &status, n, m,
                         "dense_by_rows", A_dense_ne, NULL, NULL, NULL );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_dense_ne, A_dense, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        case 4: // sparse by columns
            strcpy( st, "SC" );
            blls_import( &control, &data, &status, n, m,
                         "sparse_by_columns", A_ne, A_by_col_row,
                         NULL, A_by_col_ptr );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_ne, A_by_col_val, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
```

```
            break;
        case 5: // dense by columns
            strcpy( st, "DC" );
            blls_import( &control, &data, &status, n, m,
                         "dense_by_columns", A_dense_ne, NULL, NULL, NULL );
            blls_solve_given_a( &data, &userdata, &status, n, m,
                                A_dense_ne, A_by_col_dense, b, x_l, x_u,
                                x, z, c, g, x_stat, w, prec );
            break;
        }
    blls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%s:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%s: BLLS_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
    blls_terminate( &data, &control, &inform );
}
printf("\n tests reverse-communication options\n\n");
// reverse-communication input/output
int nm;
nm = max( n, m );
int eval_status, nz_v_start, nz_v_end, nz_p_end;
int nz_v[nm], nz_p[m], mask[m];
real_wp_ v[nm], p[nm];
nz_p_end = 0;
// Initialize BLLS
blls_initialize( &data, &control, &status );
// Set user-defined control options
control.f_indexing = true; // fortran sparse matrix indexing
// Start from 0
for( int i = 0; i < n; i++) x[i] = 0.0;
for( int i = 0; i < n; i++) z[i] = 0.0;
strcpy( st, "RC" );
for( int i = 0; i < m; i++) mask[i] = 0;
blls_import_without_a( &control, &data, &status, n, m ) ;
while(true){ // reverse-communication loop
    blls_solve_reverse_a_prod( &data, &status, &eval_status, n, m, b,
                               x_l, x_u, x, z, c, g, x_stat, v, p,
                               nz_v, &nz_v_start, &nz_v_end,
                               nz_p, nz_p_end, w );
    if(status == 0){ // successful termination
        break;
    }else if(status < 0){ // error exit
        break;
    }else if(status == 2){ // evaluate p = Av
      p[n]=0.0;
      for( int i = 0; i < n; i++){
        p[i] = v[i];
        p[n] = p[n] + v[i];
      }
    }else if(status == 3){ // evaluate p = A^Tv
      for( int i = 0; i < n; i++) p[i] = v[i] + v[n];
    }else if(status == 4){ // evaluate p = Av for sparse v
      p[n]=0.0;
      for( int i = 0; i < n; i++) p[i] = 0.0;
      for( int l = nz_v_start - 1; l < nz_v_end; l++){
        i = nz_v[l]-1;
        p[i] = v[i];
        p[n] = p[n] + v[i];
      }
    }else if(status == 5){ // evaluate p = sparse Av for sparse v
      nz_p_end = 0;
      for( int l = nz_v_start - 1; l < nz_v_end; l++){
        i = nz_v[l]-1;
        if (mask[i] == 0){
          mask[i] = 1;
          nz_p[nz_p_end] = i+1;
          nz_p_end = nz_p_end + 1;
          p[i] = v[i];
        }
        if (mask[n] == 0){
          mask[n] = 1;
          nz_p[nz_p_end] = m;
          nz_p_end = nz_p_end + 1;
          p[n] = v[i];
        }else{
          p[n] = p[n] + v[i];
```

```
            }
          }
          for( int l = 0; l < nz_p_end; l++) mask[nz_p[l]-1] = 0;
        }else if(status == 6){ // evaluate p = sparse A^Tv
          for( int l = nz_v_start - 1; l < nz_v_end; l++){
            i = nz_v[l]-1;
            p[i] = v[i] + v[n];
          }
        }else if(status == 7){ // evaluate p = P^{-}v
          for( int i = 0; i < n; i++) p[i] = userdata.scale * v[i];
        }else{
            printf(" the value %1i of status should not occur\n", status);
            break;
        }
        eval_status = 0;
    }
    // Record solution information
    blls_information( &data, &inform, &status );
    // Print solution details
    if(inform.status == 0){
        printf("%s:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%s: BLLS_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
    blls_terminate( &data, &control, &inform );
}
// Apply preconditioner
int prec( int n, const real_wp_ v[], real_wp_ p[], const void *userdata ){
  struct userdata_type *myuserdata = (struct userdata_type *) userdata;
  real_wp_ scale = myuserdata->scale;
  for( int i = 0; i < n; i++) p[i] = scale * v[i];
  return 0;
}
```