C interfaces to GALAHAD SBLS

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Sat Mar 26 2022

Chapter 1

GALAHAD C package sbls

1.1 Introduction

1.1.1 Purpose

Given a **block, symmetric matrix**

$$K_H = \begin{pmatrix} H & A^T \\ A & -C \end{pmatrix},$$

this package constructs a variety of **preconditioners** of the form

$$K_G = \begin{pmatrix} G & A^T \\ A & -C \end{pmatrix}.$$

Here, the leading-block matrix G is a suitably-chosen approximation to H; it may either be prescribed **explicitly**, in which case a symmetric indefinite factorization of K_G will be formed using the GALAHAD symmetric matrix factorization package SLS, or **implicitly**, by requiring certain sub-blocks of G be zero. In the latter case, a factorization of K_G will be obtained implicitly (and more efficiently) without recourse to SLS. In particular, for implicit preconditioners, a reordering

$$K_G = P \begin{pmatrix} G_{11} & G_{21}^T & A_1^T \\ G_{21} & G_{22} & A_2^T \\ A_1 & A_2 & -C \end{pmatrix} P^T$$

involving a suitable permutation P will be found, for some invertible sub-block (`'basis'') A_1 of the columns of A; the selection and factorization of A_1 uses the GALAHAD unsymmetric matrix factorization package ULS. Once the preconditioner has been constructed, solutions to the preconditioning system

$$\begin{pmatrix} G & A^T \\ A & -C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

may be obtained by the package. Full advantage is taken of any zero coefficients in the matrices H, A and C.

1.1.2 Authors

H. S. Dollar and N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England.

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

1.1.3 Originally released

April 2006, C interface November 2021.

1.1.4 Method

The method used depends on whether an explicit or implicit factorization is required. In the explicit case, the package is really little more than a wrapper for the GALAHAD symmetric, indefinite linear solver SLS in which the system matrix K_G is assembled from its constituents A, C and whichever G is requested by the user. Implicit-factorization preconditioners are more involved, and there is a large variety of different possibilities. The essential ideas are described in detail in

H. S. Dollar, N. I. M. Gould and A. J. Wathen. ``On implicit-factorization constraint preconditioners''. In Large Scale Nonlinear Optimization (G. Di Pillo and M. Roma, eds.) Springer Series on Nonconvex Optimization and Its Applications, Vol. 83, Springer Verlag (2006) 61–82

and

H. S. Dollar, N. I. M. Gould, W. H. A. Schilders and A. J. Wathen ``On iterative methods and implicit-factorization preconditioners for regularized saddle-point systems''. SIAM Journal on Matrix Analysis and Applications, 28(1) (2006) 170–189.

The range-space factorization is based upon the decomposition

$$K_G = \begin{pmatrix} G & 0 \\ A & I \end{pmatrix} \begin{pmatrix} G^{-1} & 0 \\ 0 & -S \end{pmatrix} \begin{pmatrix} G & A^T \\ 0 & I \end{pmatrix},$$

where the ``Schur complement'' $S = C + AG^{-1}A^T$. Such a method requires that S is easily invertible. This is often the case when G is a diagonal matrix, in which case S is frequently sparse, or when $m \ll n$ in which case S is small and a dense Cholesky factorization may be used.

When $C = 0$, the null-space factorization is based upon the decomposition

$$K_G = P \begin{pmatrix} G_{11} & 0 & I \\ G_{21} & I & A_2^T A_1^{-T} \\ A_1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & I \\ 0 & R & 0 \\ I & 0 & -G_{11} \end{pmatrix} \begin{pmatrix} G_{11} & G_{21}^T & A_1^T \\ 0 & I & 0 \\ I & A_1^{-1}A_2 & 0 \end{pmatrix} P^T,$$

where the ``reduced Hessian''

$$R = (-A_2^T A_1^{-T} \;\; I) \begin{pmatrix} G_{11} & G_{21}^T \\ G_{21} & G_{22} \end{pmatrix} \begin{pmatrix} -A_1^{-1}A_2 \\ I \end{pmatrix}$$

and P is a suitably-chosen permutation for which A_1 is invertible. The method is most useful when $m \approx n$ as then the dimension of R is small and a dense Cholesky factorization may be used.

1.1.5 Call order

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

- sbls_initialize - provide default control parameters and set up initial data structures
- sbls_read_specfile (optional) - override control values by reading replacement values from a file
- sbls_import - set up matrix data structures
- sbls_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- sbls_factorize_matrix - form and factorize the block matrix from its components
- sbls_solve_system - solve the block linear system of equations
- sbls_information (optional) - recover information about the solution and solution process
- sbls_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.6 Unsymmetric matrix storage formats

The unsymmetric m by n constraint 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.

1.1.6.1 Dense 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 $0 \leq i \leq m - 1, 0 \leq j \leq n - 1$.

1.1.6.2 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 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.

1.1.6.3 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 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 plus one. 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 predecessor.

1.1.7 Symmetric matrix storage formats

Likewise, the symmetric n by n matrix H, as well as the m by m matrix C, 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). We focus on H, but everything we say applies equally to C.

1.1.7.1 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 $0 \leq j \leq i \leq n - 1$.

1.1.7.2 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.

1.1.7.3 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.

1.1.7.4 Diagonal storage format

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

1.1.7.5 Multiples of the identity storage format

If H is a multiple of the identity matrix, (i.e., $H = \alpha I$ where I is the n by n identity matrix and α is a scalar), it suffices to store α as the first component of H_val.

1.1.7.6 The identity matrix format

If H is the identity matrix, no values need be stored.

1.1.7.7 The zero matrix format

The same is true if H is the zero matrix.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 galahad_sbls.h File Reference

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

Data Structures

- struct sbls_control_type
- struct sbls_time_type
- struct sbls_inform_type

Functions

- void sbls_initialize (void **data, struct sbls_control_type *control, int *status)
- void sbls_read_specfile (struct sbls_control_type *control, const char specfile[ ])
- void sbls_import (struct sbls_control_type *control, void **data, int *status, int n, int m, const char H_type[ ], int H_ne, const int H_row[ ], const int H_col[ ], const int H_ptr[ ], const char A_type[ ], int A_ne, const int A_row[ ], const int A_col[ ], const int A_ptr[ ], const char C_type[ ], int C_ne, const int C_row[ ], const int C_col[ ], const int C_ptr[ ])
- void sbls_reset_control (struct sbls_control_type *control, void **data, int *status)
- void sbls_factorize_matrix (void **data, int *status, int n, int h_ne, const real_wp_ H_val[ ], int a_ne, const real_wp_ A_val[ ], int c_ne, const real_wp_ C_val[ ], const real_wp_ D[ ])
- void sbls_solve_system (void **data, int *status, int n, int m, real_wp_ sol[ ])
- void sbls_information (void **data, struct sbls_inform_type *inform, int *status)
- void sbls_terminate (void **data, struct sbls_control_type *control, struct sbls_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct sbls_control_type

control derived type as a C struct

Examples

sblst.c, and sblstf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	error	unit for error messages
int	out	unit for monitor output
int	print_level	controls level of diagnostic output
int	indmin	initial estimate of integer workspace for SLS (obsolete)
int	valmin	initial estimate of real workspace for SLS (obsolete)
int	len_ulsmin	initial estimate of workspace for ULS (obsolete)
int	itref_max	maximum number of iterative refinements with preconditioner allowed
int	maxit_pcg	maximum number of projected CG iterations allowed
int	new_a	how much has A changed since last factorization: 0 = not changed, 1 = values changed, 2 = structure changed
int	new_h	how much has H changed since last factorization: 0 = not changed, 1 = values changed, 2 = structure changed
int	new_c	how much has C changed since last factorization: 0 = not changed, 1 = values changed, 2 = structure changed
int	preconditioner	which preconditioner to use: • 0 selected automatically • 1 explicit with $G = I$ • 2 explicit with $G = H$ • 3 explicit with $G =$ diag(max(H,min_diag)) • 4 explicit with $G =$ band (H) • 5 explicit with $G =$ (optional, diagonal) D • 11 explicit with $G_{11} = 0$, $G_{21} = 0$, $G_{22} = H_{22}$ • 12 explicit with $G_{11} = 0$, $G_{21} = H_{21}$, $G_{22} = H_{22}$ • -1 implicit with $G_{11} = 0$, $G_{21} = 0$, $G_{22} = I$ • -2 implicit with $G_{11} = 0$, $G_{21} = 0$, $G_{22} = H_{22}$
int	semi_bandwidth	the semi-bandwidth for band(H)

Data Fields

int	factorization	the explicit factorization used: • 0 selected automatically • 1 Schur-complement if G is diagonal and successful otherwise augmented system • 2 augmented system • 3 null-space • 4 Schur-complement if G is diagonal and successful otherwise failure • 5 Schur-complement with pivoting if G is diagonal and successful otherwise failure
int	max_col	maximum number of nonzeros in a column of A for Schur-complement factorization
int	scaling	not used at present
int	ordering	see scaling
real_wp_	pivot_tol	the relative pivot tolerance used by ULS (obsolete)
real_wp_	pivot_tol_for_basis	the relative pivot tolerance used by ULS when determining the basis matrix
real_wp_	zero_pivot	the absolute pivot tolerance used by ULS (obsolete)
real_wp_	static_tolerance	not used at present
real_wp_	static_level	see static_tolerance
real_wp_	min_diagonal	the minimum permitted diagonal in diag(max(H,min_diag))
real_wp_	stop_absolute	the required absolute and relative accuracies
real_wp_	stop_relative	see stop_absolute
bool	remove_dependencies	preprocess equality constraints to remove linear dependencies
bool	find_basis_by_transpose	determine implicit factorization preconditioners using a basis of A found by examining A's transpose
bool	affine	can the right-hand side c be assumed to be zero?
bool	allow_singular	do we tolerate "singular" preconditioners?
bool	perturb_to_make_definite	if the initial attempt at finding a preconditioner is unsuccessful, should the diagonal be perturbed so that a second attempt succeeds?
bool	get_norm_residual	compute the residual when applying the preconditioner?
bool	check_basis	if an implicit or null-space preconditioner is used, assess and correct for ill conditioned basis matrices
bool	space_critical	if space is critical, ensure allocated arrays are no bigger than needed
bool	deallocate_error_fatal	exit if any deallocation fails
char	symmetric_linear_solver[31]	indefinite linear equation solver
char	definite_linear_solver[31]	definite linear equation solver
char	unsymmetric_linear_solver[31]	unsymmetric linear equation solver

Data Fields

char	prefix[31]	all output lines will be prefixed by prefix(2:LEN(TRIM(.prefix))-1) where prefix contains the required string enclosed in quotes, e.g. "string" or 'string'
struct sls_control_type	sls_control	control parameters for SLS
struct uls_control_type	uls_control	control parameters for ULS

3.1.1.2 struct sbls_time_type

time derived type as a C struct

Data Fields

real_wp_	total	total cpu time spent in the package
real_wp_	form	cpu time spent forming the preconditioner K_G
real_wp_	factorize	cpu time spent factorizing K_G
real_wp_	apply	cpu time spent solving linear systems inolving K_G
real_wp_	clock_total	total clock time spent in the package
real_wp_	clock_form	clock time spent forming the preconditioner K_G
real_wp_	clock_factorize	clock time spent factorizing K_G
real_wp_	clock_apply	clock time spent solving linear systems inolving K_G

3.1.1.3 struct sbls_inform_type

inform derived type as a C struct

Examples

sblst.c, and sblstf.c.

Data Fields

int	status	return status. See SBLS_form_and_factorize 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	sils_analyse_status	obsolete return status from the factorization routines
int	sils_factorize_status	see sils_analyse_status
int	sils_solve_status	see sils_analyse_status
int	sls_analyse_status	see sils_analyse_status
int	sls_factorize_status	see sils_analyse_status
int	sls_solve_status	see sils_analyse_status
int	uls_analyse_status	see sils_analyse_status
int	uls_factorize_status	see sils_analyse_status
int	uls_solve_status	see sils_analyse_status

Data Fields

int	sort_status	the return status from the sorting routines
long int	factorization_integer	the total integer workspace required for the factorization
long int	factorization_real	the total real workspace required for the factorization
int	preconditioner	the preconditioner used
int	factorization	the factorization used
int	d_plus	how many of the diagonals in the factorization are positive
int	rank	the computed rank of A
bool	rank_def	is the matrix A rank defficient?
bool	perturbed	has the used preconditioner been perturbed to guarantee correct inertia?
int	iter_pcg	the total number of projected CG iterations required
real_wp_	norm_residual	the norm of the residual
bool	alternative	has an "alternative" y: $Ky = 0$ and $y^T c > 0$ been found when trying to solve $Ky = c$ for generic K?
struct sbls_time_type	time	timings (see above)
struct sls_inform_type	sls_inform	inform parameters from the GALAHAD package SLS used
struct uls_inform_type	uls_inform	inform parameters from the GALAHAD package ULS used

3.1.2 Function Documentation

3.1.2.1 sbls_initialize()

```
void sbls_initialize (
            void ** data,
            struct sbls_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 sbls_control_type)
in,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

sblst.c, and sblstf.c.

3.1.2.2 sbls_read_specfile()

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

Parameters

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

3.1.2.3 sbls_import()

```
void sbls_import (
             struct sbls_control_type * control,
             void ** data,
             int * status,
             int n,
             int m,
             const char H_type[],
             int H_ne,
             const int H_row[],
             const int H_col[],
             const int H_ptr[],
             const char A_type[],
             int A_ne,
             const int A_row[],
             const int A_col[],
             const int A_ptr[],
             const char C_type[],
             int C_ne,
             const int C_row[],
             const int C_col[],
             const int C_ptr[] )
```

Import structural matrix data into internal storage prior to solution.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see sbls_control_type)
in,out	*data*	holds private internal data

Parameters

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 restrictions n $>$ 0 or m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none' has been violated.
in	*n*	is a scalar variable of type int, that holds the number of rows in the symmetric matrix H.
in	*m*	is a scalar variable of type int, that holds the number of rows in the symmetric matrix C.
in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the matrix H. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none', the latter pair if $H = 0$; lower or upper case variants are allowed.
in	*H_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 schemes.
in	*H_row*	is a one-dimensional array of size H_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 H_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, diagonal or (scaled) identity 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.
in	*A_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the matrix A. It should be one of 'coordinate', 'sparse_by_rows', 'dense' or 'absent', the latter if access to the Jacobian is via matrix-vector products; 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 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 when the dense or diagonal storage schemes are used, and in this case can be NULL.
in	*A_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of A, 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.

Parameters

in	*C_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the matrix C. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none', the latter pair if $C = 0$; lower or upper case variants are allowed.
in	*C_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of C in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*C_row*	is a one-dimensional array of size C_ne and type int, that holds the row indices of the lower triangular part of C 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	*C_col*	is a one-dimensional array of size C_ne and type int, that holds the column indices of the lower triangular part of C in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense, diagonal or (scaled) identity storage schemes are used, and in this case can be NULL.
in	*C_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 C, 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

sblst.c, and sblstf.c.

3.1.2.4 sbls_reset_control()

```
void sbls_reset_control (
             struct sbls_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 sbls_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.

3.1.2.5 sbls_factorize_matrix()

```
void sbls_factorize_matrix (
```

```
void ** data,
int * status,
int n,
int h_ne,
const real_wp_ H_val[],
int a_ne,
const real_wp_ A_val[],
int c_ne,
const real_wp_ C_val[],
const real_wp_ D[] )
```

Form and factorize the block matrix

$$K_G = \begin{pmatrix} G & A^T \\ A & -C \end{pmatrix}$$

for some appropriate matrix G.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The factors were generated succesfully. • -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 and m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none' has been violated. • -9. An error was reported by SLS analyse. The return status from SLS analyse is given in inform.sls_inform.status. See the documentation for the GALAHAD package SLS for further details. • -10. An error was reported by SLS_factorize. The return status from SLS factorize is given in inform.sls_inform.status. See the documentation for the GALAHAD package SLS for further details. • -13. An error was reported by ULS_factorize. The return status from ULS_factorize is given in inform.uls_factorize_status. See the documentation for the GALAHAD package ULS for further details. • -15. The computed preconditioner K_G is singular and is thus unsuitable. • -20. The computed preconditioner K_G has the wrong inertia and is thus unsuitable. • -24. An error was reported by the GALAHAD package SORT_reorder_by_rows. The return status from SORT_reorder_by_rows is given in inform.sort_status. See the documentation for the GALAHAD package SORT for further details.
in	*n*	is a scalar variable of type int, that holds the number of rows in the symmetric matrix H.

Parameters

in	*h_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the symmetric matrix H.
in	*H_val*	is a one-dimensional array of size h_ne and type double, that holds the values of the entries of the lower triangular part of the symmetric matrix H in any of the available storage schemes
in	*a_ne*	is a scalar variable of type int, that holds the number of entries in the unsymmetric matrix A.
in	*A_val*	is a one-dimensional array of size a_ne and type double, that holds the values of the entries of the unsymmetric matrix A in any of the available storage schemes.
in	*c_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the symmetric matrix C.
in	*C_val*	is a one-dimensional array of size c_ne and type double, that holds the values of the entries of the lower triangular part of the symmetric matrix C in any of the available storage schemes
in	*D*	is a one-dimensional array of size n and type double, that holds the values of the entries of the diagonal matrix D that is required if the user has specified control.preconditioner = 5. It need not be set otherwise.

Examples

sblst.c, and sblstf.c.

3.1.2.6 sbls_solve_system()

```
void sbls_solve_system (
             void ** data,
             int * status,
             int n,
             int m,
             real_wp_ sol[] )
```

Solve the block linear system

$$\begin{pmatrix} G & A^T \\ A & -C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}.$$

Parameters

in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The required solution was obtained. • -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. • -11. An error was reported by SLS_solve. The return status from SLS solve is given in inform.sls_inform.status. See the documentation for the GALAHAD package SLS for further details. • -14. An error was reported by ULS_solve. The return status from ULS_solve is given in inform.uls_solve_status. See the documentation for the GALAHAD package ULS for further details.
in	*n*	is a scalar variable of type int, that holds the number of entries in the vector a.
in	*m*	is a scalar variable of type int, that holds the number of entries in the vector b.
in,out	*sol*	is a one-dimensional array of size n + m and type double. on entry, its first n entries must hold the vector a, and the following entries must hold the vector b. On a successful exit, its first n entries contain the solution components x, and the following entries contain the components y.

Examples

sblst.c, and sblstf.c.

3.1.2.7 sbls_information()

```
void sbls_information (
            void ** data,
            struct sbls_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 sbls_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

sblst.c, and sblstf.c.

3.1.2.8 sbls_terminate()

```
void sbls_terminate (
             void ** data,
             struct sbls_control_type * control,
             struct sbls_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 sbls_control_type)
out	*inform*	is a struct containing output information (see sbls_inform_type)

Examples

sblst.c, and sblstf.c.

Chapter 4

Example Documentation

4.1 sblst.c

This is an example of how to use the package.
A variety of supported matrix storage formats are illustrated.

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

```
/* sblst.c */
/* Full test for the SBLS C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_sbls.h"
int main(void) {
    // Derived types
    void *data;
    struct sbls_control_type control;
    struct sbls_inform_type inform;
    // Set problem data
    int n = 3; // dimension of H
    int m = 2; // dimension of C
    int H_ne = 4; // number of elements of H
    int A_ne = 3; // number of elements of A
    int C_ne = 3; // number of elements of C
    int H_dense_ne = 6; // number of elements of H
    int A_dense_ne = 6; // number of elements of A
    int C_dense_ne = 3; // number of elements of C
    int H_row[] = {0, 1, 2, 2}; // row indices, NB lower triangle
    int H_col[] = {0, 1, 2, 0};
    int H_ptr[] = {0, 1, 2, 4};
    int A_row[] = {0, 0, 1};
    int A_col[] = {0, 1, 2};
    int A_ptr[] = {0, 2, 3};
    int C_row[] = {0, 1, 1}; // row indices, NB lower triangle
    int C_col[] = {0, 0, 1};
    int C_ptr[] = {0, 1, 3};
    double H_val[] = {1.0, 2.0, 3.0, 1.0};
    double A_val[] = {2.0, 1.0, 1.0};
    double C_val[] = {4.0, 1.0, 2.0};
    double H_dense[] = {1.0, 0.0, 2.0, 1.0, 0.0, 3.0};
    double A_dense[] = {2.0, 1.0, 0.0, 0.0, 0.0, 1.0};
    double C_dense[] = {4.0, 1.0, 2.0};
    double H_diag[] = {1.0, 1.0, 2.0};
    double C_diag[] = {4.0, 2.0};
    double H_scid[] = {2.0};
    double C_scid[] = {2.0};
    char st;
    int status;
    printf(" C sparse matrix indexing\n\n");
    printf(" basic tests of storage formats\n\n");
    for( int d=1; d <= 7; d++){
        // Initialize SBLS
        sbls_initialize( &data, &control, &status );
        control.preconditioner = 2;
        control.factorization = 2;
```

```
control.get_norm_residual = true;
// Set user-defined control options
control.f_indexing = false; // C sparse matrix indexing
switch(d){
    case 1: // sparse co-ordinate storage
        st = 'C';
        sbls_import( &control, &data, &status, n, m,
                     "coordinate", H_ne, H_row, H_col, NULL,
                     "coordinate", A_ne, A_row, A_col, NULL,
                     "coordinate", C_ne, C_row, C_col, NULL );
        sbls_factorize_matrix( &data, &status, n,
                               H_ne, H_val,
                               A_ne, A_val,
                               C_ne, C_val, NULL );
        break;
    printf(" case %1i break\n",d);
    case 2: // sparse by rows
        st = 'R';
        sbls_import( &control, &data, &status, n, m,
                     "sparse_by_rows", H_ne, NULL, H_col, H_ptr,
                     "sparse_by_rows", A_ne, NULL, A_col, A_ptr,
                     "sparse_by_rows", C_ne, NULL, C_col, C_ptr );
        sbls_factorize_matrix( &data, &status, n,
                               H_ne, H_val,
                               A_ne, A_val,
                               C_ne, C_val, NULL );
        break;
    case 3: // dense
        st = 'D';
        sbls_import( &control, &data, &status, n, m,
                     "dense", H_ne, NULL, NULL, NULL,
                     "dense", A_ne, NULL, NULL, NULL,
                     "dense", C_ne, NULL, NULL, NULL );
        sbls_factorize_matrix( &data, &status, n,
                               H_dense_ne, H_dense,
                               A_dense_ne, A_dense,
                               C_dense_ne, C_dense,
                               NULL );
        break;
    case 4: // diagonal
        st = 'L';
        sbls_import( &control, &data, &status, n, m,
                     "diagonal", H_ne, NULL, NULL, NULL,
                     "dense", A_ne, NULL, NULL, NULL,
                     "diagonal", C_ne, NULL, NULL, NULL );
        sbls_factorize_matrix( &data, &status, n,
                               n, H_diag,
                               A_dense_ne, A_dense,
                               m, C_diag,
                               NULL );
        break;
    case 5: // scaled identity
        st = 'S';
        sbls_import( &control, &data, &status, n, m,
                     "scaled_identity", H_ne, NULL, NULL, NULL,
                     "dense", A_ne, NULL, NULL, NULL,
                     "scaled_identity", C_ne, NULL, NULL, NULL );
        sbls_factorize_matrix( &data, &status, n,
                               1, H_scid,
                               A_dense_ne, A_dense,
                               1, C_scid,
                               NULL );
        break;
    case 6: // identity
        st = 'I';
        sbls_import( &control, &data, &status, n, m,
                     "identity", H_ne, NULL, NULL, NULL,
                     "dense", A_ne, NULL, NULL, NULL,
                     "identity", C_ne, NULL, NULL, NULL );
        sbls_factorize_matrix( &data, &status, n,
                               0, H_val,
                               A_dense_ne, A_dense,
                               0, C_val, NULL );
        break;
    case 7: // zero
        st = 'Z';
        sbls_import( &control, &data, &status, n, m,
                     "identity", H_ne, NULL, NULL, NULL,
                     "dense", A_ne, NULL, NULL, NULL,
                     "zero", C_ne, NULL, NULL, NULL );
        sbls_factorize_matrix( &data, &status, n,
                               0, H_val,
                               A_dense_ne, A_dense,
                               0, NULL, NULL );
        break;
    }
// Set right-hand side ( a, b )
```

```
        double sol[] = {3.0, 2.0, 4.0, 2.0, 0.0};   // values
        sbls_solve_system( &data, &status, n, m, sol );
        sbls_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c: residual = %9.1e status = %1i\n",
                   st, inform.norm_residual, inform.status);
        }else{
            printf("%c: SBLS_solve exit status = %1i\n", st, inform.status);
        }
        //printf("sol: ");
        //for( int i = 0; i < n+m; i++) printf("%f ", x[i]);
        // Delete internal workspace
        sbls_terminate( &data, &control, &inform );
    }
}
```

4.2 sblstf.c

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

```
/* sblstf.c */
/* Full test for the SBLS C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_sbls.h"
int main(void) {
    // Derived types
    void *data;
    struct sbls_control_type control;
    struct sbls_inform_type inform;
    // Set problem data
    int n = 3; // dimension of H
    int m = 2; // dimension of C
    int H_ne = 4; // number of elements of H
    int A_ne = 3; // number of elements of A
    int C_ne = 3; // number of elements of C
    int H_dense_ne = 6; // number of elements of H
    int A_dense_ne = 6; // number of elements of A
    int C_dense_ne = 3; // number of elements of C
    int H_row[] = {1, 2, 3, 3}; // row indices, NB lower triangle
    int H_col[] = {1, 2, 3, 1};
    int H_ptr[] = {1, 2, 3, 5};
    int A_row[] = {1, 1, 2};
    int A_col[] = {1, 2, 3};
    int A_ptr[] = {1, 3, 4};
    int C_row[] = {1, 2, 2}; // row indices, NB lower triangle
    int C_col[] = {1, 1, 2};
    int C_ptr[] = {1, 2, 4};
    double H_val[] = {1.0, 2.0, 3.0, 1.0};
    double A_val[] = {2.0, 1.0, 1.0};
    double C_val[] = {4.0, 1.0, 2.0};
    double H_dense[] = {1.0, 0.0, 2.0, 1.0, 0.0, 3.0};
    double A_dense[] = {2.0, 1.0, 0.0, 0.0, 0.0, 1.0};
    double C_dense[] = {4.0, 1.0, 2.0};
    double H_diag[] = {1.0, 1.0, 2.0};
    double C_diag[] = {4.0, 2.0};
    double H_scid[] = {2.0};
    double C_scid[] = {2.0};
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" basic tests of storage formats\n\n");
    for( int d=1; d <= 7; d++){
        // Initialize SBLS
        sbls_initialize( &data, &control, &status );
        control.preconditioner = 2;
        control.factorization = 2;
        control.get_norm_residual = true;
        // Set user-defined control options
        control.f_indexing = true; // fortran sparse matrix indexing
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                sbls_import( &control, &data, &status, n, m,
                             "coordinate", H_ne, H_row, H_col, NULL,
                             "coordinate", A_ne, A_row, A_col, NULL,
                             "coordinate", C_ne, C_row, C_col, NULL );
                sbls_factorize_matrix( &data, &status, n,
                                       H_ne, H_val,
                                       A_ne, A_val,
                                       C_ne, C_val, NULL );
```

```
            break;
        printf(" case %1i break\n",d);
        case 2: // sparse by rows
            st = 'R';
            sbls_import( &control, &data, &status, n, m,
                         "sparse_by_rows", H_ne, NULL, H_col, H_ptr,
                         "sparse_by_rows", A_ne, NULL, A_col, A_ptr,
                         "sparse_by_rows", C_ne, NULL, C_col, C_ptr );
            sbls_factorize_matrix( &data, &status, n,
                                   H_ne, H_val,
                                   A_ne, A_val,
                                   C_ne, C_val, NULL );
            break;
        case 3: // dense
            st = 'D';
            sbls_import( &control, &data, &status, n, m,
                         "dense", H_ne, NULL, NULL, NULL,
                         "dense", A_ne, NULL, NULL, NULL,
                         "dense", C_ne, NULL, NULL, NULL );
            sbls_factorize_matrix( &data, &status, n,
                                   H_dense_ne, H_dense,
                                   A_dense_ne, A_dense,
                                   C_dense_ne, C_dense,
                                   NULL );
            break;
        case 4: // diagonal
            st = 'L';
            sbls_import( &control, &data, &status, n, m,
                         "diagonal", H_ne, NULL, NULL, NULL,
                         "dense", A_ne, NULL, NULL, NULL,
                         "diagonal", C_ne, NULL, NULL, NULL );
            sbls_factorize_matrix( &data, &status, n,
                                   n, H_diag,
                                   A_dense_ne, A_dense,
                                   m, C_diag,
                                   NULL );
            break;
        case 5: // scaled identity
            st = 'S';
            sbls_import( &control, &data, &status, n, m,
                         "scaled_identity", H_ne, NULL, NULL, NULL,
                         "dense", A_ne, NULL, NULL, NULL,
                         "scaled_identity", C_ne, NULL, NULL, NULL );
            sbls_factorize_matrix( &data, &status, n,
                                   1, H_scid,
                                   A_dense_ne, A_dense,
                                   1, C_scid,
                                   NULL );
            break;
        case 6: // identity
            st = 'I';
            sbls_import( &control, &data, &status, n, m,
                         "identity", H_ne, NULL, NULL, NULL,
                         "dense", A_ne, NULL, NULL, NULL,
                         "identity", C_ne, NULL, NULL, NULL );
            sbls_factorize_matrix( &data, &status, n,
                                   0, H_val,
                                   A_dense_ne, A_dense,
                                   0, C_val, NULL );
            break;
        case 7: // zero
            st = 'Z';
            sbls_import( &control, &data, &status, n, m,
                         "identity", H_ne, NULL, NULL, NULL,
                         "dense", A_ne, NULL, NULL, NULL,
                         "zero", C_ne, NULL, NULL, NULL );
            sbls_factorize_matrix( &data, &status, n,
                                   0, H_val,
                                   A_dense_ne, A_dense,
                                   0, NULL, NULL );
            break;
        }
    // Set right-hand side ( a, b )
    double sol[] = {3.0, 2.0, 4.0, 2.0, 0.0};   // values
    sbls_solve_system( &data, &status, n, m, sol );
    sbls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c: residual = %9.1e status = %1i\n",
               st, inform.norm_residual, inform.status);
    }else{
        printf("%c: SBLS_solve exit status = %1i\n", st, inform.status);
    }
    //printf("sol: ");
    //for( int i = 0; i < n+m; i++) printf("%f ", x[i]);
    // Delete internal workspace
    sbls_terminate( &data, &control, &inform );
}
```

```
}
```

Index