C interfaces to GALAHAD RQS

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory

Wed May 3 2023

Chapter 1

GALAHAD C package rqs

1.1 Introduction

1.1.1 Purpose

Given real n by n symmetric matrices H and M (with M diagonally dominant), another real m by n matrix A, a real n vector c and scalars $\sigma > 0, p > 2$ and f, this package finds an **approximate minimizer of the regularised quadratic objective function** $\frac{1}{2}x^T Hx + c^T x + f + \frac{1}{p}\sigma\|x\|_M^p$**, where the vector** x **may additionally be required to satisfy** $Ax = 0$**,** and where the M-norm of x is $\|x\|_M = \sqrt{x^T Mx}$. This problem commonly occurs as a subproblem in nonlinear optimization calculations. The matrix M need not be provided in the commonly-occurring ℓ_2-regularisation case for which $M = I$, the n by n identity matrix.

Factorization of matrices of the form $H + \lambda M$—or

$$(1) \quad \left(\begin{array}{cc} H + \lambda M & A^T \\ A & 0 \end{array}\right)$$

in cases where $Ax = 0$ is imposed—for a succession of scalars λ will be required, so this package is most suited for the case where such a factorization may be found efficiently. If this is not the case, the GALAHAD package `GLRT` may be preferred.

1.1.2 Authors

N. I. M. Gould and H. S. Thorne, STFC-Rutherford Appleton Laboratory, England, and D. P. Robinson, Oxford University, 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

November 2008, C interface December 2021.

1.1.4 Method

The required solution x_* necessarily satisfies the optimality condition $Hx_* + \lambda_* M x_* + A^T y_* + c = 0$ and $Ax_* = 0$, where $\lambda_* = \sigma \|x_*\|^{p-2}$ is a Lagrange multiplier corresponding to the regularisation and y_* are Lagrange multipliers for the linear constraints $Ax = 0$, if any. In addition in all cases, the matrix $H + \lambda_* M$ will be positive semi-definite on the null-space of A; in most instances it will actually be positive definite, but in special `‘hard" cases singularity is a possibility.

The method is iterative, and proceeds in two phases. Firstly, lower and upper bounds, λ_L and λ_U, on λ_* are computed using Gershgorin's theorems and other eigenvalue bounds. The first phase of the computation proceeds by progressively shrinking the bound interval $[\lambda_L, \lambda_U]$ until a value λ for which $\|x(\lambda)\|_M \geq \sigma \|x(\lambda)\|_M^{p-2}$ is found. Here $x(\lambda)$ and its companion $y(\lambda)$ are defined to be a solution of

$$(2) \quad (H + \lambda M)x(\lambda) + A^T y(\lambda) = -c \text{ and } Ax(\lambda) = 0.$$

Once the terminating λ from the first phase has been discovered, the second phase consists of applying Newton or higher-order iterations to the nonlinear `‘secular" equation $\|x(\lambda)\|_M = \sigma \|x(\lambda)\|_M^{p-2}$ with the knowledge that such iterations are both globally and ultimately rapidly convergent. It is possible in the `‘hard" case that the interval in the first-phase will shrink to the single point λ_*, and precautions are taken, using inverse iteration with Rayleigh-quotient acceleration to ensure that this too happens rapidly.

The dominant cost is the requirement that we solve a sequence of linear systems (2). In the absence of linear constraints, an efficient sparse Cholesky factorization with precautions to detect indefinite $H + \lambda M$ is used. If $Ax = 0$ is required, a sparse symmetric, indefinite factorization of (1) is used rather than a Cholesky factorization.

1.1.5 Reference

The method is described in detail in

H. S. Dollar, N. I. M. Gould and D. P. Robinson. On solving trust-region and other regularised subproblems in optimization. Mathematical Programming Computation **2(1)** (2010) 21–57.

1.1.6 Call order

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

- rqs_initialize - provide default control parameters and set up initial data structures
- rqs_read_specfile (optional) - override control values by reading replacement values from a file
- rqs_import - set up problem data structures and fixed values
- rqs_import_m - (optional) set up problem data structures and fixed values for the scaling matrix M, if any
- rqs_import_a - (optional) set up problem data structures and fixed values for the constraint matrix A, if any
- rqs_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- rqs_solve_problem - solve the regularised quadratic problem
- rqs_information (optional) - recover information about the solution and solution process
- rqs_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.7 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.

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

1.1.7.2 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 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.

1.1.7.3 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 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), $\ldots$, 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.

1.1.8 Symmetric matrix storage formats

Likewise, the symmetric n by n objective Hessian matrix H and scaling matrix M 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). In what follows, we refer to H but this applies equally to M.

1.1.8.1 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 $0 \leq j \leq i \leq n-1$.

1.1.8.2 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.

1.1.8.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row i+1. For the i-th row of H the i-th component of the integer array H_ptr holds the position of the first entry in this row, while H_ptr(n) holds the total number of entries. The column indices j, $0 \leq j \leq i$, and values h_{ij} of the entries in the i-th row are stored in components l = H_ptr(i), . . ., H_ptr(i+1)-1 of the integer array H_col, and real array H_val, respectively. Note that as before only the entries in the lower triangle should be stored. For sparse matrices, this scheme almost always requires less storage than its predecessor.

1.1.8.4 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.

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_rqs.h File Reference

```
#include <stdbool.h>
#include <stdint.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_sls.h"
#include "galahad_ir.h"
```

Data Structures

- struct rqs_control_type
- struct rqs_time_type
- struct rqs_history_type
- struct rqs_inform_type

Functions

- void rqs_initialize (void **data, struct rqs_control_type *control, int *status)
- void rqs_read_specfile (struct rqs_control_type *control, const char specfile[ ])
- void rqs_import (struct rqs_control_type *control, void **data, int *status, int n, const char H_type[ ], int H_ne, const int H_row[ ], const int H_col[ ], const int H_ptr[ ])
- void rqs_import_m (void **data, int *status, int n, const char M_type[ ], int M_ne, const int M_row[ ], const int M_col[ ], const int M_ptr[ ])
- void rqs_import_a (void **data, int *status, int m, const char A_type[ ], int A_ne, const int A_row[ ], const int A_col[ ], const int A_ptr[ ])
- void rqs_reset_control (struct rqs_control_type *control, void **data, int *status)
- void rqs_solve_problem (void **data, int *status, int n, const real_wp_ power, const real_wp_ weight, const real_wp_ f, const real_wp_ c[ ], int H_ne, const real_wp_ H_val[ ], real_wp_ x[ ], int M_ne, const real_wp_ M_val[ ], int m, int A_ne, const real_wp_ A_val[ ], real_wp_ y[ ])
- void rqs_information (void **data, struct rqs_inform_type *inform, int *status)
- void rqs_terminate (void **data, struct rqs_control_type *control, struct rqs_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct rqs_control_type

control derived type as a C struct

Examples

rqst.c, and rqstf.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	problem	unit to write problem data into file problem_file
int	print_level	controls level of diagnostic output
int	dense_factorization	should the problem be solved by dense factorization? Possible values are • 0 sparse factorization will be used • 1 dense factorization will be used • other the choice is made automatically depending on the dimension and sparsity
int	new_h	how much of H has changed since the previous call. Possible values are • 0 unchanged • 1 values but not indices have changed • 2 values and indices have changed
int	new_m	how much of M has changed since the previous call. Possible values are • 0 unchanged • 1 values but not indices have changed • 2 values and indices have changed
int	new_a	how much of A has changed since the previous call. Possible values are 0 unchanged 1 values but not indices have changed 2 values and indices have changed
int	max_factorizations	the maximum number of factorizations (=iterations) allowed. -ve implies no limit
int	inverse_itmax	the number of inverse iterations performed in the "maybe hard" case
int	taylor_max_degree	maximum degree of Taylor approximant allowed
real_wp_	initial_multiplier	initial estimate of the Lagrange multipler
real_wp_	lower	lower and upper bounds on the multiplier, if known
real_wp_	upper	see lower

Data Fields

real_wp_	stop_normal	stop when $\|\|\|x\|\| - (multiplier/\sigma)^(1/(p-2))\| \leq$ stop_normal $*$ max $(\|\|x\|\|, (multiplier/\sigma)^(1/(p-2)))$ REAL (KIND = wp) :: stop_normal = epsmch $**$ 0.75
real_wp_	stop_hard	stop when bracket on optimal multiplier $<=$ stop_hard $*$ max(bracket ends) REAL (KIND = wp) :: stop_hard = epsmch $**$ 0.75
real_wp_	start_invit_tol	start inverse iteration when bracket on optimal multiplier $<=$ stop_start_invit_tol $*$ max(bracket ends)
real_wp_	start_invitmax_tol	start full inverse iteration when bracket on multiplier $<=$ stop_start_invitmax_tol $*$ max(bracket ends)
bool	use_initial_multiplier	ignore initial_multiplier?
bool	initialize_approx_eigenvector	should a suitable initial eigenvector should be chosen or should a previous eigenvector may be used?
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	problem_file[31]	name of file into which to write problem data
char	symmetric_linear_solver[31]	symmetric (indefinite) linear equation solver
char	definite_linear_solver[31]	definite linear equation solver
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 the Cholesky factorization and solution (see sls_c documentation)
struct ir_control_type	ir_control	control parameters for iterative refinement (see ir_c documentation)

3.1.1.2 struct rqs_time_type

time derived type as a C struct

Data Fields

real_wp_	total	total CPU time spent in the package
real_wp_	assemble	CPU time spent building $H + \lambda M$.
real_wp_	analyse	CPU time spent reordering $H + \lambda M$ prior to factorization.
real_wp_	factorize	CPU time spent factorizing $H + \lambda M$.
real_wp_	solve	CPU time spent solving linear systems inolving $H + \lambda M$.
real_wp_	clock_total	total clock time spent in the package
real_wp_	clock_assemble	clock time spent building $H + \lambda M$
real_wp_	clock_analyse	clock time spent reordering $H + \lambda M$ prior to factorization
real_wp_	clock_factorize	clock time spent factorizing $H + \lambda M$
real_wp_	clock_solve	clock time spent solving linear systems inolving $H + \lambda M$

3.1.1.3 struct rqs_history_type

history derived type as a C struct

Data Fields

real_wp_	lambda	the value of λ
real_wp_	x_norm	the corresponding value of $\|x(\lambda)\|_M$

3.1.1.4 struct rqs_inform_type

inform derived type as a C struct

Examples

rqst.c, and rqstf.c.

Data Fields

int	status	reported return status: • 0 the solution has been found • -1 an array allocation has failed • -2 an array deallocation has failed • -3 n and/or sigma is not positive and/or p <= 2 • -9 the analysis phase of the factorization of $H + \lambda M$ failed • -10 the factorization of $H + \lambda M$ failed • -15 M does not appear to be strictly diagonally dominant • -16 ill-conditioning has prevented furthr progress
int	alloc_status	STAT value after allocate failure.
int	factorizations	the number of factorizations performed
int64_t	max_entries_factors	the maximum number of entries in the factors
int	len_history	the number of $(\|x\|_M, \lambda)$ pairs in the history
real_wp_	obj	the value of the quadratic function
real_wp_	obj_regularized	the value of the regularized quadratic function
real_wp_	x_norm	the M-norm of x, $\|x\|_M$
real_wp_	multiplier	the Lagrange multiplier corresponding to the regularization
real_wp_	pole	a lower bound max $(0, -\lambda_1)$, where λ_1 is the left-most eigenvalue of (H, M)
bool	dense_factorization	was a dense factorization used?
bool	hard_case	has the hard case occurred?
char	bad_alloc[81]	name of array which provoked an allocate failure
struct rqs_time_type	time	time information
struct rqs_history_type	history[100]	history information
struct sls_inform_type	sls_inform	cholesky information (see sls_c documentation)
struct ir_inform_type	ir_inform	iterative_refinement information (see ir_c documentation)

3.1.2 Function Documentation

3.1.2.1 rqs_initialize()

```
void rqs_initialize (
            void ** data,
            struct rqs_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 rqs_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

rqst.c, and rqstf.c.

3.1.2.2 rqs_read_specfile()

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

Parameters

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

3.1.2.3 rqs_import()

```
void rqs_import (
```

```
struct rqs_control_type * control,
void ** data,
int * status,
int n,
const char H_type[ ],
int H_ne,
const int H_row[ ],
const int H_col[ ],
const int H_ptr[ ] )
```

Import problem data into internal storage prior to solution.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see rqs_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 and m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', diagonal' or 'identity' has been violated.
in	*n*	is a scalar variable of type int, that holds the number of rows (and columns) of H.
in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian, H. It should be one of 'coordinate', 'sparse_by_rows', 'dense', or 'diagonal'; 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 or diagonal storage schemes are used, and in this case can be NULL.
in	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of H, as well as the total number of entries, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.

Examples

rqst.c, and rqstf.c.

3.1.2.4 rqs_import_m()

```
void rqs_import_m (
            void ** data,
            int * status,
            int n,
            const char M_type[],
            int M_ne,
            const int M_row[],
            const int M_col[],
            const int M_ptr[] )
```

Import data for the scaling matrix M into internal storage prior to solution.

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 and m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', diagonal' or 'identity' has been violated.
in	*n*	is a scalar variable of type int, that holds the number of rows (and columns) of M.
in	*M_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the scaling matrix, M. It should be one of 'coordinate', 'sparse_by_rows', 'dense', or 'diagonal'; lower or upper case variants are allowed.
in	*M_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of M in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*M_row*	is a one-dimensional array of size M_ne and type int, that holds the row indices of the lower triangular part of M 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	*M_col*	is a one-dimensional array of size M_ne and type int, that holds the column indices of the lower triangular part of M in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense, diagonal or identity storage schemes are used, and in this case can be NULL.
in	*M_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 M, as well as the total number of entries, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.

Examples

rqst.c, and rqstf.c.

3.1.2.5 rqs_import_a()

```
void rqs_import_a (
            void ** data,
            int * status,
            int m,
            const char A_type[],
            int A_ne,
            const int A_row[],
            const int A_col[],
            const int A_ptr[] )
```

Import data for the constraint matrix A into internal storage prior to solution.

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 and m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate' or 'sparse_by_rows' has been violated.
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints, i.e., the number of rows of A, if any. m must be non-negative.
in	*A_type*	is a one-dimensional array of type char that specifies the unsymmetric storage scheme used for the constraint Jacobian, A if any. It should be one of 'coordinate', 'sparse_by_rows' or 'dense'; 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, if used, 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, 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

rqst.c, and rqstf.c.

3.1.2.6 rqs_reset_control()

```
void rqs_reset_control (
            struct rqs_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 rqs_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.

3.1.2.7 rqs_solve_problem()

```
void rqs_solve_problem (
            void ** data,
            int * status,
            int n,
            const real_wp_ power,
            const real_wp_ weight,
            const real_wp_ f,
            const real_wp_ c[ ],
            int H_ne,
            const real_wp_ H_val[ ],
            real_wp_ x[ ],
            int M_ne,
            const real_wp_ M_val[ ],
            int m,
            int A_ne,
            const real_wp_ A_val[ ],
            real_wp_ y[ ] )
```

Solve the regularised quadratic problem.

Parameters

in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful. • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0, power $>$ 2, weight $>$ 0 and m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'identity' has been violated. • -9. The analysis phase of the factorization of the matrix (1) failed. • -10. The factorization of the matrix (1) failed. • -15. The matrix M appears not to be diagonally dominant. • -16. The problem is so ill-conditioned that further progress is impossible. • -18. Too many factorizations have been required. This may happen if control.max factorizations 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	*power*	is a scalar of type double, that holds the order of regularisation, p, used. power must be no smaller than 2.
in	*weight*	is a scalar of type double, that holds the regularisation weight, σ, used. weight must be strictly positive.
in	*c*	is a one-dimensional array of size n and type double, that holds the linear term c of the objective function. The j-th component of c, j = 0, ... , n-1, contains c_j.
in	*f*	is a scalar of type double, that holds the constant term f of the objective function.
in	*H_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix H.
in	*H_val*	is a one-dimensional array of size h_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.
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	*M_ne*	is a scalar variable of type int, that holds the number of entries in the scaling matrix M if it not the identity matrix.
in	*M_val*	is a one-dimensional array of size M_ne and type double, that holds the values of the entries of the scaling matrix M, if it is not the identity matrix, in any of the available storage schemes. If M_val is NULL, M will be taken to be the identity matrix.
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints, if any. m must be non-negative.
in	*A_ne*	is a scalar variable of type int, that holds the number of entries in the constraint Jacobian matrix A if used. A_ne must be non-negative.

Parameters

in	*A_val*	is a one-dimensional array of size A_ne and type double, that holds the values of the entries of the constraint Jacobian matrix A, if used, in any of the available storage schemes. If A_val is NULL, no constraints will be enforced.
out	*y*	is a one-dimensional array of size m and type double, that holds the values y of the Lagrange multipliers for the equality constraints $Ax = 0$ if used. The i-th component of y, i = 0, ... , m-1, contains y_i.

Examples

rqst.c, and rqstf.c.

3.1.2.8 rqs_information()

```
void rqs_information (
            void ** data,
            struct rqs_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 rqs_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

rqst.c, and rqstf.c.

3.1.2.9 rqs_terminate()

```
void rqs_terminate (
            void ** data,
            struct rqs_control_type * control,
            struct rqs_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 rqs_control_type)
`out`	*inform*	is a struct containing output information (see rqs_inform_type)

Examples

rqst.c, and rqstf.c.

Chapter 4

Example Documentation

4.1 rqst.c

This is an example of how to use the package to solve a regularised quadratic problem. A variety of supported Hessian, scaling and constraint matrix storage formats are shown.

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

```
/* rqst.c */
/* Full test for the RQS C interface using C sparse matrix indexing */
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_rqs.h"
int main(void) {
    // Derived types
    void *data;
    struct rqs_control_type control;
    struct rqs_inform_type inform;
    // Set problem data
    int n = 3; // dimension of H
    int m = 1; // dimension of A
    int H_ne = 4; // number of elements of H
    int M_ne = 3; // number of elements of M
    int A_ne = 3; // number of elements of A
    int H_dense_ne = 6; // number of elements of H
    int M_dense_ne = 6; // number of elements of M
    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 M_row[] = {0, 1, 2}; // row indices, NB lower triangle
    int M_col[] = {0, 1, 2};
    int M_ptr[] = {0, 1, 2, 3};
    int A_row[] = {0, 0, 0} ;
    int A_col[] = {0, 1, 2};
    int A_ptr[] = {0, 3};
    real_wp_ H_val[] = {1.0, 2.0, 3.0, 4.0};
    real_wp_ M_val[] = {1.0, 2.0, 1.0};
    real_wp_ A_val[] = {1.0, 1.0, 1.0};
    real_wp_ H_dense[] = {1.0, 0.0, 2.0, 4.0, 0.0, 3.0};
    real_wp_ M_dense[] = {1.0, 0.0, 2.0, 0.0, 0.0, 1.0};
    real_wp_ H_diag[] = {1.0, 0.0, 2.0};
    real_wp_ M_diag[] = {1.0, 2.0, 1.0};
    real_wp_ f = 0.96;
    real_wp_ power = 3.0;
    real_wp_ weight = 1.0;
    real_wp_ c[] = {0.0, 2.0, 0.0};
    char st;
    int status;
    real_wp_ x[n];
    char ma[3];
    printf(" C sparse matrix indexing\n\n");
    printf(" basic tests of storage formats\n\n");
```

```
for( int a_is=0; a_is <= 1; a_is++){ // add a linear constraint?
  for( int m_is=0; m_is <= 1; m_is++){ // include a scaling matrix?
    if (a_is == 1 && m_is == 1 ) {
      strcpy(ma, "MA");
    }
    else if (a_is == 1) {
      strcpy(ma, "A ");
    }
    else if (m_is == 1) {
      strcpy(ma, "M ");
    }
    else {
      strcpy(ma, "  ");
    }
    for( int storage_type=1; storage_type <= 4; storage_type++){
      // Initialize RQS
      rqs_initialize( &data, &control, &status );
      // Set user-defined control options
      control.f_indexing = false; // C sparse matrix indexing
      switch(storage_type){
          case 1: // sparse co-ordinate storage
              st = 'C';
              // import the control parameters and structural data
              rqs_import( &control, &data, &status, n,
                         "coordinate", H_ne, H_row, H_col, NULL );
              if (m_is == 1) {
                rqs_import_m( &data, &status, n,
                              "coordinate", M_ne, M_row, M_col, NULL );
              }
              if (a_is == 1) {
                rqs_import_a( &data, &status, m,
                              "coordinate", A_ne, A_row, A_col, NULL );
              }
              // solve the problem
              if (a_is == 1 && m_is == 1 ) {
                rqs_solve_problem( &data, &status, n,
                                   power, weight, f, c, H_ne, H_val, x,
                                   M_ne, M_val, m, A_ne, A_val, NULL );
              }
              else if (a_is == 1) {
                rqs_solve_problem( &data, &status, n,
                                   power, weight, f, c, H_ne, H_val, x,
                                   0, NULL, m, A_ne, A_val, NULL );
              }
              else if (m_is == 1) {
                rqs_solve_problem( &data, &status, n,
                                   power, weight, f, c, H_ne, H_val, x,
                                   M_ne, M_val, 0, 0, NULL, NULL );
              }
              else {
                rqs_solve_problem( &data, &status, n,
                                   power, weight, f, c, H_ne, H_val, x,
                                   0, NULL, 0, 0, NULL, NULL );
              }
              break;
          printf(" case %1i break\n", storage_type );
          case 2: // sparse by rows
              st = 'R';
              // import the control parameters and structural data
              rqs_import( &control, &data, &status, n,
                          "sparse_by_rows", H_ne, NULL, H_col, H_ptr );
              if (m_is == 1) {
                rqs_import_m( &data, &status, n,
                              "sparse_by_rows", M_ne, NULL, M_col, M_ptr );
              }
              if (a_is == 1) {
                rqs_import_a( &data, &status, m,
                             "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
              }
              // solve the problem
              if (a_is == 1 && m_is == 1 ) {
                rqs_solve_problem( &data, &status, n,
                                   power, weight, f, c, H_ne, H_val, x,
                                   M_ne, M_val, m, A_ne, A_val, NULL );
              }
              else if (a_is == 1) {
                rqs_solve_problem( &data, &status, n,
                                   power, weight, f, c, H_ne, H_val, x,
                                   0, NULL, m, A_ne, A_val, NULL );
              }
              else if (m_is == 1) {
                rqs_solve_problem( &data, &status, n,
                                   power, weight, f, c, H_ne, H_val, x,
                                   M_ne, M_val, 0, 0, NULL, NULL );
              }
              else {
                rqs_solve_problem( &data, &status, n,
```

```
                              power, weight, f, c, H_ne, H_val, x,
                              0, NULL, 0, 0, NULL, NULL );
            }
            break;
        case 3: // dense
            st = 'D';
            // import the control parameters and structural data
            rqs_import( &control, &data, &status, n,
                        "dense", H_ne, NULL, NULL, NULL );
            if (m_is == 1) {
              rqs_import_m( &data, &status, n,
                            "dense", M_ne, NULL, NULL, NULL );
            }
            if (a_is == 1) {
              rqs_import_a( &data, &status, m,
                            "dense", A_ne, NULL, NULL, NULL );
            }
            // solve the problem
            if (a_is == 1 && m_is == 1 ) {
              rqs_solve_problem( &data, &status, n, power, weight,
                                 f, c, H_dense_ne, H_dense, x,
                                 M_dense_ne, M_dense, m, A_ne, A_val,
                                 NULL );
            }
            else if (a_is == 1) {
              rqs_solve_problem( &data, &status, n, power, weight,
                                 f, c, H_dense_ne, H_dense, x,
                                 0, NULL, m, A_ne, A_val, NULL );
            }
            else if (m_is == 1) {
              rqs_solve_problem( &data, &status, n, power, weight,
                                 f, c, H_dense_ne, H_dense, x,
                                 M_dense_ne, M_dense, 0, 0, NULL, NULL );
            }
            else {
              rqs_solve_problem( &data, &status, n, power, weight,
                                 f, c, H_dense_ne, H_dense, x,
                                 0, NULL, 0, 0, NULL, NULL );
            }
            break;
        case 4: // diagonal
            st = 'L';
            // import the control parameters and structural data
            rqs_import( &control, &data, &status, n,
                        "diagonal", H_ne, NULL, NULL, NULL );
            if (m_is == 1) {
              rqs_import_m( &data, &status, n,
                            "diagonal", M_ne, NULL, NULL, NULL );
            }
            if (a_is == 1) {
              rqs_import_a( &data, &status, m,
                            "dense", A_ne, NULL, NULL, NULL );
            }
            // solve the problem
            if (a_is == 1 && m_is == 1 ) {
              rqs_solve_problem( &data, &status, n,
                                 power, weight, f, c, n, H_diag, x,
                                 n, M_diag, m, A_ne, A_val, NULL );
            }
            else if (a_is == 1) {
              rqs_solve_problem( &data, &status, n,
                                 power, weight, f, c, n, H_diag, x,
                                 0, NULL, m, A_ne, A_val, NULL );
            }
            else if (m_is == 1) {
              rqs_solve_problem( &data, &status, n,
                                 power, weight, f, c, n, H_diag, x,
                                 n, M_diag, 0, 0, NULL, NULL );
            }
            else {
              rqs_solve_problem( &data, &status, n,
                                 power, weight, f, c, n, H_diag, x,
                                 0, NULL, 0, 0, NULL, NULL );
            }
            break;
        }
    rqs_information( &data, &inform, &status );
    printf("format %c%s: RQS_solve_problem exit status = %1i, f = %.2f\n",
           st, ma, inform.status, inform.obj_regularized );
    //printf("x: ");
    //for( int i = 0; i < n+m; i++) printf("%f ", x[i]);
    // Delete internal workspace
    rqs_terminate( &data, &control, &inform );
  }
 }
}
}
```

4.2 rqstf.c

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

```
/* rqstf.c */
/* Full test for the RQS C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_rqs.h"
int main(void) {
    // Derived types
    void *data;
    struct rqs_control_type control;
    struct rqs_inform_type inform;
    // Set problem data
    int n = 3; // dimension of H
    int m = 1; // dimension of A
    int H_ne = 4; // number of elements of H
    int M_ne = 3; // number of elements of M
    int A_ne = 3; // number of elements of A
    int H_dense_ne = 6; // number of elements of H
    int M_dense_ne = 6; // number of elements of M
    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 M_row[] = {1, 2, 3}; // row indices, NB lower triangle
    int M_col[] = {1, 2, 3};
    int M_ptr[] = {1, 2, 3, 4};
    int A_row[] = {1, 1, 1} ;
    int A_col[] = {1, 2, 3};
    int A_ptr[] = {1, 4};
    real_wp_ H_val[] = {1.0, 2.0, 3.0, 4.0};
    real_wp_ M_val[] = {1.0, 2.0, 1.0};
    real_wp_ A_val[] = {1.0, 1.0, 1.0};
    real_wp_ H_dense[] = {1.0, 0.0, 2.0, 4.0, 0.0, 3.0};
    real_wp_ M_dense[] = {1.0, 0.0, 2.0, 0.0, 0.0, 1.0};
    real_wp_ H_diag[] = {1.0, 0.0, 2.0};
    real_wp_ M_diag[] = {1.0, 2.0, 1.0};
    real_wp_ f = 0.96;
    real_wp_ power = 3.0;
    real_wp_ weight = 1.0;
    real_wp_ c[] = {0.0, 2.0, 0.0};
    char st;
    int status;
    real_wp_ x[n];
    char ma[3];
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" basic tests of storage formats\n\n");
    for( int a_is=0; a_is <= 1; a_is++){ // add a linear constraint?
      for( int m_is=0; m_is <= 1; m_is++){ // include a scaling matrix?
        if (a_is == 1 && m_is == 1 ) {
          strcpy(ma, "MA");
        }
        else if (a_is == 1) {
          strcpy(ma, "A ");
        }
        else if (m_is == 1) {
          strcpy(ma, "M ");
        }
        else {
          strcpy(ma, "  ");
        }
        for( int storage_type=1; storage_type <= 4; storage_type++){
          // Initialize RQS
          rqs_initialize( &data, &control, &status );
          // Set user-defined control options
          control.f_indexing = true; // fortran sparse matrix indexing
          switch(storage_type){
              case 1: // sparse co-ordinate storage
                  st = 'C';
                  // import the control parameters and structural data
                  rqs_import( &control, &data, &status, n,
                             "coordinate", H_ne, H_row, H_col, NULL );
                  if (m_is == 1) {
                    rqs_import_m( &data, &status, n,
                                 "coordinate", M_ne, M_row, M_col, NULL );
                  }
                  if (a_is == 1) {
                    rqs_import_a( &data, &status, m,
                                 "coordinate", A_ne, A_row, A_col, NULL );
                  }
                  // solve the problem
```

```
        if (a_is == 1 && m_is == 1 ) {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             M_ne, M_val, m, A_ne, A_val, NULL );
        }
        else if (a_is == 1) {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             0, NULL, m, A_ne, A_val, NULL );
        }
        else if (m_is == 1) {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             M_ne, M_val, 0, 0, NULL, NULL );
        }
        else {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             0, NULL, 0, 0, NULL, NULL );
        }
        break;
    printf(" case %1i break\n", storage_type );
    case 2: // sparse by rows
        st = 'R';
        // import the control parameters and structural data
        rqs_import( &control, &data, &status, n,
                    "sparse_by_rows", H_ne, NULL, H_col, H_ptr );
        if (m_is == 1) {
          rqs_import_m( &data, &status, n,
                        "sparse_by_rows", M_ne, NULL, M_col, M_ptr );
        }
        if (a_is == 1) {
          rqs_import_a( &data, &status, m,
                       "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
        }
        // solve the problem
        if (a_is == 1 && m_is == 1 ) {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             M_ne, M_val, m, A_ne, A_val, NULL );
        }
        else if (a_is == 1) {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             0, NULL, m, A_ne, A_val, NULL );
        }
        else if (m_is == 1) {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             M_ne, M_val, 0, 0, NULL, NULL );
        }
        else {
          rqs_solve_problem( &data, &status, n,
                             power, weight, f, c, H_ne, H_val, x,
                             0, NULL, 0, 0, NULL, NULL );
        }
        break;
    case 3: // dense
        st = 'D';
        // import the control parameters and structural data
        rqs_import( &control, &data, &status, n,
                    "dense", H_ne, NULL, NULL, NULL );
        if (m_is == 1) {
          rqs_import_m( &data, &status, n,
                       "dense", M_ne, NULL, NULL, NULL );
        }
        if (a_is == 1) {
          rqs_import_a( &data, &status, m,
                       "dense", A_ne, NULL, NULL, NULL );
        }
        // solve the problem
        if (a_is == 1 && m_is == 1 ) {
          rqs_solve_problem( &data, &status, n, power, weight,
                             f, c, H_dense_ne, H_dense, x,
                             M_dense_ne, M_dense, m, A_ne, A_val,
                             NULL );
        }
        else if (a_is == 1) {
          rqs_solve_problem( &data, &status, n, power, weight,
                             f, c, H_dense_ne, H_dense, x,
                             0, NULL, m, A_ne, A_val, NULL );
        }
        else if (m_is == 1) {
          rqs_solve_problem( &data, &status, n, power, weight,
                             f, c, H_dense_ne, H_dense, x,
                             M_dense_ne, M_dense, 0, 0, NULL, NULL );
        }
```

```
                else {
                  rqs_solve_problem( &data, &status, n, power, weight,
                                     f, c, H_dense_ne, H_dense, x,
                                     0, NULL, 0, 0, NULL, NULL );
                }
                break;
            case 4: // diagonal
                st = 'L';
                // import the control parameters and structural data
                rqs_import( &control, &data, &status, n,
                            "diagonal", H_ne, NULL, NULL, NULL );
                if (m_is == 1) {
                  rqs_import_m( &data, &status, n,
                                "diagonal", M_ne, NULL, NULL, NULL );
                }
                if (a_is == 1) {
                  rqs_import_a( &data, &status, m,
                                "dense", A_ne, NULL, NULL, NULL );
                }
                // solve the problem
                if (a_is == 1 && m_is == 1 ) {
                  rqs_solve_problem( &data, &status, n,
                                     power, weight, f, c, n, H_diag, x,
                                     n, M_diag, m, A_ne, A_val, NULL );
                }
                else if (a_is == 1) {
                  rqs_solve_problem( &data, &status, n,
                                     power, weight, f, c, n, H_diag, x,
                                     0, NULL, m, A_ne, A_val, NULL );
                }
                else if (m_is == 1) {
                  rqs_solve_problem( &data, &status, n,
                                     power, weight, f, c, n, H_diag, x,
                                     n, M_diag, 0, 0, NULL, NULL );
                }
                else {
                  rqs_solve_problem( &data, &status, n,
                                     power, weight, f, c, n, H_diag, x,
                                     0, NULL, 0, 0, NULL, NULL );
                }
                break;
            }
        rqs_information( &data, &inform, &status );
        printf("format %c%s: RQS_solve_problem exit status = %1i, f = %.2f\n",
               st, ma, inform.status, inform.obj_regularized );
        //printf("x: ");
        //for( int i = 0; i < n+m; i++) printf("%f ", x[i]);
        // Delete internal workspace
        rqs_terminate( &data, &control, &inform );
      }
    }
  }
}
```