C interfaces to GALAHAD EQP

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Sun Mar 20 2022

Chapter 1

GALAHAD C package eqp

1.1 Introduction

1.1.1 Purpose

This package uses an iterative method to solve the **equality-constrained quadratic programming problem**

$$\text{minimize} \;\; q(x) = \frac{1}{2} x^T H x + g^T x + f$$

subject to the linear constraints

$$(1) \;\; Ax + c = 0,$$

where the n by n symmetric matrix H, the m by n matrix A, the vectors g and c Full advantage is taken of any zero coefficients in the matrices H and A.

The package may alternatively be used to minimize the (shifted) squared- least-distance objective

$$\frac{1}{2} \sum_{j=1}^{n} w_j^2 (x_j - x_j^0)^2 + g^T x + f,$$

subject to the linear constraint (1), for given vectors w and x^0.

1.1.2 Authors

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

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

1.1.3 Originally released

March 2006, C interface January 2021.

1.1.4 Terminology

The required solution x necessarily satisfies the primal optimality conditions

$$(2) \quad Ax + c = 0$$

and the dual optimality conditions

$$Hx + g - A^T y = 0 \ \ (\text{or } \ W^2(x - x^0) + g - A^T y = 0 \ \text{ for the shifted-least-distance type objective})$$

where the diagonal matrix W^2 has diagonal entries w_j^2, $j = 1, \ldots, n$, and where the vector y is known as the Lagrange multipliers for the linear constraints.

1.1.5 Method

A solution to the problem is found in two phases. In the first, a point x_F satisfying (2) is found. In the second, the required solution $x = x_F + s$ is determined by finding s to minimize $q(s) = \frac{1}{2}s^T Hs + g_F^T s + f_F$ subject to the homogeneous constraints $As = zero$, where $g_F = Hx_F + g$ and $f_F = \frac{1}{2}x_F^T Hx_F + g^T x_F + f$. The required constrained minimizer of $q(s)$ is obtained by implictly applying the preconditioned conjugate-gradient method in the null space of A. Any preconditioner of the form

$$K_G = \begin{pmatrix} G & A^T \\ A & 0 \end{pmatrix}$$

is suitable, and the GALAHAD package SBLS provides a number of possibilities. In order to ensure that the minimizer obtained is finite, an additional, precautionary trust-region constraint $\|s\| \leq \Delta$ for some suitable positive radius Δ is imposed, and the GALAHAD package GLTR is used to solve this additionally-constrained problem.

1.1.6 Reference

The preconditioning aspcets 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,

while the constrained conjugate-gradient method is discussed in

N. I. M. Gould, S. Lucidi, M. Roma and Ph. L. Toint, Solving the trust-region subproblem using the Lanczos method. SIAM Journal on Optimization **9:2** (1999), 504-525.

1.1.7 Call order

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

- eqp_initialize - provide default control parameters and set up initial data structures
- eqp_read_specfile (optional) - override control values by reading replacement values from a file
- eqp_import - set up problem data structures and fixed values
- eqp_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
  - eqp_solve_qp - solve the quadratic program
  - eqp_solve_sldqp - solve the shifted least-distance problem
- eqp_resolve_qp (optional) - resolve the problem with the same Hessian and Jacobian, but different g, f and/or c
- eqp_information (optional) - recover information about the solution and solution process
- eqp_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.8 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.

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

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

1.1.8.3 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.

1.1.9 Symmetric matrix storage formats

Likewise, the symmetric n by n objective Hessian matrix H 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).

1.1.9.1 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 $0 \leq j \leq i \leq n-1$.

1.1.9.2 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.

1.1.9.3 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.

1.1.9.4 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.

1.1.9.5 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.

1.1.9.6 The identity matrix format

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 eqp.h File Reference

```
#include <stdbool.h>
#include "galahad_precision.h"
#include "fdc.h"
#include "sbls.h"
#include "gltr.h"
```

Data Structures

- struct eqp_control_type
- struct eqp_time_type
- struct eqp_inform_type

Functions

- void eqp_initialize (void **data, struct eqp_control_type *control, int *status)
- void eqp_read_specfile (struct eqp_control_type *control, const char specfile[ ])
- void eqp_import (struct eqp_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[ ])
- void eqp_reset_control (struct eqp_control_type *control, void **data, int *status)
- void eqp_solve_qp (void **data, int *status, int n, int m, int h_ne, const real_wp_ H_val[ ], const real_wp_ g[ ], const real_wp_ f, int a_ne, const real_wp_ A_val[ ], real_wp_ c[ ], real_wp_ x[ ], real_wp_ y[ ])
- void eqp_solve_sldqp (void **data, int *status, int n, int m, const real_wp_ w[ ], const real_wp_ x0[ ], const real_wp_ g[ ], const real_wp_ f, int a_ne, const real_wp_ A_val[ ], real_wp_ c[ ], real_wp_ x[ ], real_wp_ y[ ])
- void eqp_resolve_qp (void **data, int *status, int n, int m, const real_wp_ g[ ], const real_wp_ f, real_wp_ c[ ], real_wp_ x[ ], real_wp_ y[ ])
- void eqp_information (void **data, struct eqp_inform_type *inform, int *status)
- void eqp_terminate (void **data, struct eqp_control_type *control, struct eqp_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct eqp_control_type

control derived type as a C struct

Examples

eqpt.c, and eqptf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	error	error and warning diagnostics occur on stream error
int	out	general output occurs on stream out
int	print_level	the level of output required is specified by print_level
int	factorization	the factorization to be used. Possible values are 0 automatic 1 Schur-complement factorization 2 augmented-system factorization (OBSOLETE
int	max_col	the maximum number of nonzeros in a column of A which is permitted with the Schur-complement factorization (OBSOLE
int	indmin	an initial guess as to the integer workspace required by SBLS (OBSOL
int	valmin	an initial guess as to the real workspace required by SBLS (OBSOL
int	len_ulsmin	an initial guess as to the workspace required by ULS (OBSOL
int	itref_max	the maximum number of iterative refinements allowed (OBSOL
int	cg_maxit	the maximum number of CG iterations allowed. If cg_maxit $<$ 0, this number will be reset to the dimension of the system + 1
int	preconditioner	the preconditioner to be used for the CG is defined by precon. Possible values are 0 automatic 1 no preconditioner, i.e, the identity within full factorization 2 full factorization 3 band within full factorization 4 diagonal using the barrier terms within full factorization (OBSOLETE 5 optionally supplied diagonal, G = D
int	semi_bandwidth	the semi-bandwidth of a band preconditioner, if appropriate (OBSOL
int	new_a	how much has A changed since last problem solved: 0 = not changed, 1 = values changed, 2 = structure changed
int	new_h	how much has H changed since last problem solved: 0 = not changed, 1 = values changed, 2 = structure changed
int	sif_file_device	specifies the unit number to write generated SIF file describing the current problem
real_wp_	pivot_tol	the threshold pivot used by the matrix factorization. See the documentation for SBLS for details (OBSOLE
real_wp_	pivot_tol_for_basis	the threshold pivot used by the matrix factorization when finding the ba See the documentation for ULS for details (OBSOLE
real_wp_	zero_pivot	any pivots smaller than zero_pivot in absolute value will be regarded to zero when attempting to detect linearly dependent constraints (OBSOLE
real_wp_	inner_fraction_opt	the computed solution which gives at least inner_fraction_opt times the optimal value will be found (OBSOLE
real_wp_	radius	an upper bound on the permitted step (-ve will be reset to an appropriat large value by eqp_solve)
real_wp_	min_diagonal	diagonal preconditioners will have diagonals no smaller than min_diagona (OBSOLETE)

Data Fields

real_wp_	max_infeasibility_relative	if the constraints are believed to be rank defficient and the residual at a "typical" feasible point is larger than max(max_infeasibility_relative ∗ norm A, max_infeasibility_absolute) the problem will be marked as infeasible
real_wp_	max_infeasibility_absolute	see max_infeasibility_relative
real_wp_	inner_stop_relative	the computed solution is considered as an acceptable approximation to th minimizer of the problem if the gradient of the objective in the preconditioning(inverse) norm is less than max(inner_stop_relative ∗ initial preconditioning(inverse) gradient norm, inner_stop_absolute)
real_wp_	inner_stop_absolute	see inner_stop_relative
real_wp_	inner_stop_inter	see inner_stop_relative
bool	find_basis_by_transpose	if .find_basis_by_transpose is true, implicit factorization precondition will be based on a basis of A found by examining A's transpose (OBSOLE
bool	remove_dependencies	if .remove_dependencies is true, the equality constraints will be preprocessed to remove any linear dependencies
bool	space_critical	if .space_critical true, every effort will be made to use as little space as possible. This may result in longer computation time
bool	deallocate_error_fatal	if .deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue
bool	generate_sif_file	if .generate_sif_file is .true. if a SIF file describing the current problem is to be generated
char	sif_file_name[31]	name of generated SIF file containing input problem
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 fdc_control_type	fdc_control	control parameters for FDC
struct sbls_control_type	sbls_control	control parameters for SBLS
struct gltr_control_type	gltr_control	control parameters for GLTR

3.1.1.2 struct eqp_time_type

time derived type as a C struct

Data Fields

real_wp_	total	the total CPU time spent in the package
real_wp_	find_dependent	the CPU time spent detecting linear dependencies
real_wp_	factorize	the CPU time spent factorizing the required matrices
real_wp_	solve	the CPU time spent computing the search direction
real_wp_	solve_inter	see solve
real_wp_	clock_total	the total clock time spent in the package

Data Fields

real_wp_	clock_find_dependent	the clock time spent detecting linear dependencies
real_wp_	clock_factorize	the clock time spent factorizing the required matrices
real_wp_	clock_solve	the clock time spent computing the search direction

3.1.1.3 struct eqp_inform_type

inform derived type as a C struct

Examples

eqpt.c, and eqptf.c.

Data Fields

int	status	return status. See EQP_solve for details
int	alloc_status	the status of the last attempted allocation/deallocation
char	bad_alloc[81]	the name of the array for which an allocation/deallocation error ocurred
int	cg_iter	the total number of conjugate gradient iterations required
int	cg_iter_inter	see cg_iter
int	factorization_integer	the total integer workspace required for the factorization
int	factorization_real	the total real workspace required for the factorization
real_wp_	obj	the value of the objective function at the best estimate of the solution determined by QPB_solve
struct eqp_time_type	time	timings (see above)
struct fdc_inform_type	fdc_inform	inform parameters for FDC
struct sbls_inform_type	sbls_inform	inform parameters for SBLS
struct gltr_inform_type	gltr_inform	return information from GLTR

3.1.2 Function Documentation

3.1.2.1 eqp_initialize()

```
void eqp_initialize (
            void ** data,
            struct eqp_control_type * control,
            int * status )
```

Set default control values and initialize private data

Parameters

in,out	*data*	holds private internal data

Parameters

out	*control*	is a struct containing control information (see eqp_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

eqpt.c, and eqptf.c.

3.1.2.2 eqp_read_specfile()

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

Parameters

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

3.1.2.3 eqp_import()

```
void eqp_import (
             struct eqp_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[] )
```

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 eqp_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 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. • -23. An entry from the strict upper triangle of H has been specified.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints.
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', '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 unsymmetric storage scheme used for the constraint Jacobian, A. 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 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.

Parameters

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.

Examples

eqpt.c, and eqptf.c.

3.1.2.4 eqp_reset_control()

```
void eqp_reset_control (
             struct eqp_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 eqp_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 eqp_solve_qp()

```
void eqp_solve_qp (
             void ** data,
             int * status,
             int n,
             int m,
             int h_ne,
             const real_wp_ H_val[],
             const real_wp_ g[],
             const real_wp_ f,
             int a_ne,
             const real_wp_ A_val[],
             real_wp_ c[],
             real_wp_ x[],
             real_wp_ y[] )
```

Solve the quadratic program when the Hessian H is available.

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. Possible exit are: • 0. The run was succesful. • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The 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. • -7. The constraints appear to have no feasible point. • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -17. The step is too small to make further impact. • -18. Too many iterations have been performed. This may happen if control.maxit is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -23. An entry from the strict upper triangle of H has been specified.
in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints.
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.
in	*g*	is a one-dimensional array of size n and type double, that holds the linear term g of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
in	*f*	is a scalar of type double, that holds the constant term f of the objective function.
in	*a_ne*	is a scalar variable of type int, that holds the number of entries in the constraint Jacobian 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 constraint Jacobian matrix A in any of the available storage schemes.

Parameters

in	*c*	is a one-dimensional array of size m and type double, that holds the linear term c in the constraints. The i-th component of c, i = 0, ... , m-1, contains c_i.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in,out	*y*	is a one-dimensional array of size n and type double, that holds the values y of the Lagrange multipliers for the linear constraints. The j-th component of y, i = 0, ... , m-1, contains y_i.

Examples

eqpt.c, and eqptf.c.

3.1.2.6 eqp_solve_sldqp()

```
void eqp_solve_sldqp (
             void ** data,
             int * status,
             int n,
             int m,
             const real_wp_ w[],
             const real_wp_ x0[],
             const real_wp_ g[],
             const real_wp_ f,
             int a_ne,
             const real_wp_ A_val[],
             real_wp_ c[],
             real_wp_ x[],
             real_wp_ y[] )
```

Solve the shifted least-distance quadratic program

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. 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 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. • -7. The constraints appear to have no feasible point. • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -17. The step is too small to make further impact. • -18. Too many iterations have been performed. This may happen if control.maxit is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -23. An entry from the strict upper triangle of H has been specified.
in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints.
in	*w*	is a one-dimensional array of size n and type double, that holds the values of the weights w.
in	*x0*	is a one-dimensional array of size n and type double, that holds the values of the shifts x^0.
in	*g*	is a one-dimensional array of size n and type double, that holds the linear term g of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
in	*f*	is a scalar of type double, that holds the constant term f of the objective function.
in	*a_ne*	is a scalar variable of type int, that holds the number of entries in the constraint Jacobian 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 constraint Jacobian matrix A in any of the available storage schemes.
in	*c*	is a one-dimensional array of size m and type double, that holds the linear term c in the constraints. The i-th component of c, i = 0, ... , m-1, contains c_i.

Parameters

in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in,out	*y*	is a one-dimensional array of size n and type double, that holds the values y of the Lagrange multipliers for the linear constraints. The j-th component of y, i = 0, ... , m-1, contains y_i.

Examples

eqpt.c, and eqptf.c.

3.1.2.7 eqp_resolve_qp()

```
void eqp_resolve_qp (
             void ** data,
             int * status,
             int n,
             int m,
             const real_wp_ g[],
             const real_wp_ f,
             real_wp_ c[],
             real_wp_ x[],
             real_wp_ y[] )
```

Resolve the quadratic program or shifted least-distance quadratic program when some or all of the data g, f and c has changed

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. 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 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. • -7. The constraints appear to have no feasible point. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -17. The step is too small to make further impact. • -18. Too many iterations have been performed. This may happen if control.maxit is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -23. An entry from the strict upper triangle of H has been specified.
in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints.
in	*g*	is a one-dimensional array of size n and type double, that holds the linear term g of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
in	*f*	is a scalar of type double, that holds the constant term f of the objective function.
in	*c*	is a one-dimensional array of size m and type double, that holds the linear term c in the constraints. The i-th component of c, i = 0, ... , m-1, contains c_i.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in,out	*y*	is a one-dimensional array of size n and type double, that holds the values y of the Lagrange multipliers for the linear constraints. The j-th component of y, i = 0, ... , m-1, contains y_i.

3.1.2.8 eqp_information()

```
void eqp_information (
             void ** data,
```

```
            struct eqp_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 eqp_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

eqpt.c, and eqptf.c.

3.1.2.9 eqp_terminate()

```
void eqp_terminate (
            void ** data,
            struct eqp_control_type * control,
            struct eqp_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 eqp_control_type)
out	*inform*	is a struct containing output information (see eqp_inform_type)

Examples

eqpt.c, and eqptf.c.

Chapter 4

Example Documentation

4.1 eqpt.c

This is an example of how to use the package to solve a quadratic program. A variety of supported Hessian 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`.

```
/* eqpt.c */
/* Full test for the EQP C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "eqp.h"
int main(void) {
    // Derived types
    void *data;
    struct eqp_control_type control;
    struct eqp_inform_type inform;
    // Set problem data
    int n = 3; // dimension
    int m = 2; // number of general constraints
    int H_ne = 3; // Hesssian elements
    int H_row[] = {0, 1, 2 };   // row indices, NB lower triangle
    int H_col[] = {0, 1, 2};    // column indices, NB lower triangle
    int H_ptr[] = {0, 1, 2, 3}; // row pointers
    double H_val[] = {1.0, 1.0, 1.0 };   // values
    double g[] = {0.0, 2.0, 0.0};   // linear term in the objective
    double f = 1.0;  // constant term in the objective
    int A_ne = 4; // Jacobian elements
    int A_row[] = {0, 0, 1, 1}; // row indices
    int A_col[] = {0, 1, 1, 2}; // column indices
    int A_ptr[] = {0, 2, 4}; // row pointers
    double A_val[] = {2.0, 1.0, 1.0, 1.0 }; // values
    // Set output storage
    double c[m]; // constraint values
    int x_stat[n]; // variable status
    int c_stat[m]; // constraint status
    char st;
    int status;
    printf(" C sparse matrix indexing\n\n");
    printf(" basic tests of qp storage formats\n\n");
    for( int d=1; d <= 7; d++){
        // Initialize EQP
        eqp_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = false; // C sparse matrix indexing
        // Start from 0
        double x[] = {0.0,0.0,0.0};
        double y[] = {0.0,0.0};
        double z[] = {0.0,0.0,0.0};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                eqp_import( &control, &data, &status, n, m,
                            "coordinate", H_ne, H_row, H_col, NULL,
```

```
                    "coordinate", A_ne, A_row, A_col, NULL );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        printf(" case %1i break\n",d);
        case 2: // sparse by rows
            st = 'R';
            eqp_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 );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 3: // dense
            st = 'D';
            int H_dense_ne = 6; // number of elements of H
            int A_dense_ne = 6; // number of elements of A
            double H_dense[] = {1.0, 0.0, 1.0, 0.0, 0.0, 1.0};
            double A_dense[] = {2.0, 1.0, 0.0, 0.0, 1.0, 1.0};
            eqp_import( &control, &data, &status, n, m,
                        "dense", H_ne, NULL, NULL, NULL,
                        "dense", A_ne, NULL, NULL, NULL );
            eqp_solve_qp( &data, &status, n, m, H_dense_ne, H_dense, g, f,
                          A_dense_ne, A_dense, c, x, y );
            break;
        case 4: // diagonal
            st = 'L';
            eqp_import( &control, &data, &status, n, m,
                        "diagonal", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 5: // scaled identity
            st = 'S';
            eqp_import( &control, &data, &status, n, m,
                        "scaled_identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 6: // identity
            st = 'I';
            eqp_import( &control, &data, &status, n, m,
                        "identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 7: // zero
            st = 'Z';
            eqp_import( &control, &data, &status, n, m,
                        "zero", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        }
    eqp_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c:%6i cg iterations. Optimal objective value = %5.2f status = %1i\n",
               st, inform.cg_iter, inform.obj, inform.status);
    }else{
        printf("%c: EQP_solve exit status = %1i\n", st, inform.status);
    }
    //printf("x: ");
    //for( int i = 0; i < n; i++) printf("%f ", x[i]);
    //printf("\n");
    //printf("gradient: ");
    //for( int i = 0; i < n; i++) printf("%f ", g[i]);
    //printf("\n");
    // Delete internal workspace
    eqp_terminate( &data, &control, &inform );
}
// test shifted least-distance interface
for( int d=1; d <= 1; d++){
    // Initialize EQP
    eqp_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    // Start from 0
    double x[] = {0.0,0.0,0.0};
    double y[] = {0.0,0.0};
    double z[] = {0.0,0.0,0.0};
    // Set shifted least-distance data
    double w[] = {1.0,1.0,1.0};
    double x_0[] = {0.0,0.0,0.0};
```

```
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'W';
                eqp_import( &control, &data, &status, n, m,
                            "shifted_least_distance", H_ne, NULL, NULL, NULL,
                            "coordinate", A_ne, A_row, A_col, NULL );
                eqp_solve_sldqp( &data, &status, n, m, w, x_0, g, f,
                                 A_ne, A_val, c, x, y );
                break;
            }
        eqp_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i cg iterations. Optimal objective value = %5.2f status = %1i\n",
                   st, inform.cg_iter, inform.obj, inform.status);
        }else{
            printf("%c: EQP_solve exit status = %1i\n", st, inform.status);
        }
        //printf("x: ");
        //for( int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
        //for( int i = 0; i < n; i++) printf("%f ", g[i]);
        //printf("\n");
        // Delete internal workspace
        eqp_terminate( &data, &control, &inform );
    }
}
```

4.2 eqptf.c

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

```
/* eqptf.c */
/* Full test for the EQP C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "eqp.h"
int main(void) {
    // Derived types
    void *data;
    struct eqp_control_type control;
    struct eqp_inform_type inform;
    // Set problem data
    int n = 3; // dimension
    int m = 2; // number of general constraints
    int H_ne = 3; // Hesssian elements
    int H_row[] = {1, 2, 3 };   // row indices, NB lower triangle
    int H_col[] = {1, 2, 3};    // column indices, NB lower triangle
    int H_ptr[] = {1, 2, 3, 4}; // row pointers
    double H_val[] = {1.0, 1.0, 1.0 };   // values
    double g[] = {0.0, 2.0, 0.0};   // linear term in the objective
    double f = 1.0;  // constant term in the objective
    int A_ne = 4; // Jacobian elements
    int A_row[] = {1, 1, 2, 2}; // row indices
    int A_col[] = {1, 2, 2, 3}; // column indices
    int A_ptr[] = {1, 3, 5}; // row pointers
    double A_val[] = {2.0, 1.0, 1.0, 1.0 }; // values
    // Set output storage
    double c[m]; // constraint values
    int x_stat[n]; // variable status
    int c_stat[m]; // constraint status
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" basic tests of qp storage formats\n\n");
    for( int d=1; d <= 7; d++){
        // Initialize EQP
        eqp_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        // Start from 0
        double x[] = {0.0,0.0,0.0};
        double y[] = {0.0,0.0};
        double z[] = {0.0,0.0,0.0};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                eqp_import( &control, &data, &status, n, m,
                            "coordinate", H_ne, H_row, H_col, NULL,
                            "coordinate", A_ne, A_row, A_col, NULL );
                eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
```

```
                          A_ne, A_val, c, x, y );
            break;
        printf(" case %1i break\n",d);
        case 2: // sparse by rows
            st = 'R';
            eqp_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 );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 3: // dense
            st = 'D';
            int H_dense_ne = 6; // number of elements of H
            int A_dense_ne = 6; // number of elements of A
            double H_dense[] = {1.0, 0.0, 1.0, 0.0, 0.0, 1.0};
            double A_dense[] = {2.0, 1.0, 0.0, 0.0, 1.0, 1.0};
            eqp_import( &control, &data, &status, n, m,
                        "dense", H_ne, NULL, NULL, NULL,
                        "dense", A_ne, NULL, NULL, NULL );
            eqp_solve_qp( &data, &status, n, m, H_dense_ne, H_dense, g, f,
                          A_dense_ne, A_dense, c, x, y );
            break;
        case 4: // diagonal
            st = 'L';
            eqp_import( &control, &data, &status, n, m,
                        "diagonal", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 5: // scaled identity
            st = 'S';
            eqp_import( &control, &data, &status, n, m,
                        "scaled_identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 6: // identity
            st = 'I';
            eqp_import( &control, &data, &status, n, m,
                        "identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        case 7: // zero
            st = 'Z';
            eqp_import( &control, &data, &status, n, m,
                        "zero", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            eqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c, x, y );
            break;
        }
    eqp_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c:%6i cg iterations. Optimal objective value = %5.2f status = %1i\n",
               st, inform.cg_iter, inform.obj, inform.status);
    }else{
        printf("%c: EQP_solve exit status = %1i\n", st, inform.status);
    }
    //printf("x: ");
    //for( int i = 0; i < n; i++) printf("%f ", x[i]);
    //printf("\n");
    //printf("gradient: ");
    //for( int i = 0; i < n; i++) printf("%f ", g[i]);
    //printf("\n");
    // Delete internal workspace
    eqp_terminate( &data, &control, &inform );
}
// test shifted least-distance interface
for( int d=1; d <= 1; d++){
    // Initialize EQP
    eqp_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = true; // Fortran sparse matrix indexing
    // Start from 0
    double x[] = {0.0,0.0,0.0};
    double y[] = {0.0,0.0};
    double z[] = {0.0,0.0,0.0};
    // Set shifted least-distance data
    double w[] = {1.0,1.0,1.0};
    double x_0[] = {0.0,0.0,0.0};
    switch(d){
        case 1: // sparse co-ordinate storage
```

```
                st = 'W';
                eqp_import( &control, &data, &status, n, m,
                            "shifted_least_distance", H_ne, NULL, NULL, NULL,
                            "coordinate", A_ne, A_row, A_col, NULL );
                eqp_solve_sldqp( &data, &status, n, m, w, x_0, g, f,
                                 A_ne, A_val, c, x, y );
                break;
            }
        eqp_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i cg iterations. Optimal objective value = %5.2f status = %1i\n",
                   st, inform.cg_iter, inform.obj, inform.status);
        }else{
            printf("%c: EQP_solve exit status = %1i\n", st, inform.status);
        }
        //printf("x: ");
        //for( int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
        //for( int i = 0; i < n; i++) printf("%f ", g[i]);
        //printf("\n");
        // Delete internal workspace
        eqp_terminate( &data, &control, &inform );
    }
}
```

Index