C interfaces to GALAHAD DQP

Generated by Doxygen 1.8.17

Chapter 1

GALAHAD C package dqp

1.1 Introduction

1.1.1 Purpose

This package uses a dual gradient-projection interior-point method to solve the **strictly convex quadratic programming problem**

$$(0) \quad \text{minimize} \;\; q(x) = \frac{1}{2} x^T H x + g^T x + f$$

or the **shifted least-distance problem**

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

subject to the general linear constraints

$$c_i^l \leq a_i^T x \leq c_i^u, \quad i = 1, \ldots, m,$$

and the simple bound constraints

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

where the n by n symmetric, positive-definite matrix H, the vectors g, w, x^0, a_i, c^l, c^u, x^l, x^u and the scalar f are given. Any of the constraint bounds c_i^l, c_i^u, x_j^l and x_j^u may be infinite. Full advantage is taken of any zero coefficients in the matrix H or the matrix A of vectors a_i.

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

August 2012, C interface December 2021.

1.1.4 Terminology

The required solution x necessarily satisfies the primal optimality conditions

(1a) $$Ax = c$$

and

(1b) $$c^l \leq c \leq c^u, \;\; x^l \leq x \leq x^u,$$

the dual optimality conditions

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

where

(2b) $$y = y^l + y^u, \;\; z = z^l + z^u, \;\; y^l \geq 0, \;\; y^u \leq 0, \;\; z^l \geq 0 \;\; \text{and} \;\; z^u \leq 0,$$

and the complementary slackness conditions

(3) $$(Ax - c^l)^T y^l = 0, \;\; (Ax - c^u)^T y^u = 0, \;\; (x - x^l)^T z^l = 0 \;\; \text{and} \;\; (x - x^u)^T z^u = 0,$$

where the diagonal matrix W^2 has diagonal entries w_j^2, $j = 1, \ldots, n$, where the vectors y and z are known as the Lagrange multipliers for the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold component-wise.

1.1.5 Method

Dual gradient-projection methods solve (0) by instead solving the dual quadratic program

(4) $$\begin{array}{ll} \text{minimize} & q^D(y^l, y^u, z^l, z^u) = \frac{1}{2}[(y^l + y^u)^T A + (z^l + z^u]^T) H^{-1} [A^T (y^l + y^u) + z^l + z^u] \\ & \quad -[(y^l + y^u)^T A + (z^l + z^u]^T) H^{-1} g - (c^{lT} y^l + c^{uT} y^u + x^{lT} z^l + x^{uT} z^u) \\ \text{subject to} & (y^l, z^l) \geq 0 \;\; \text{and} \;\; (y^u, z^u) \leq 0, \end{array}$$

and then recovering the required solution from the linear system

$$Hx = -g + A^T (y^l + y^u) + z^l + z^u.$$

The dual problem (4) is solved by an accelerated gradient-projection method comprising of alternating phases in which (i) the current projected dual gradient is traced downhill (the 'arc search') as far as possible and (ii) the dual variables that are currently on their bounds are temporarily fixed and the unconstrained minimizer of $q^D(y^l, y^u, z^l, z^u)$ with respect to the remaining variables is sought; the minimizer in the second phase may itself need to be projected back into the dual feasible region (either using a brute-force backtrack or a second arc search).

Both phases require the solution of sparse systems of symmetric linear equations, and these are handled by the GALAHAD matrix factorization package SBLS or the GALAHAD conjugate-gradient package GLTR. The systems are commonly singular, and this leads to a requirement to find the Fredholm Alternative for the given matrix and its right-hand side. In the non-singular case, there is an option to update existing factorizations using the "Schur-complement" approach given by the GALAHAD package SCU.

Optionally, the problem may be pre-processed temporarily to eliminate dependentconstraints using the GALAHAD package FDC. This may improve the performance of the subsequent iteration.

1.1.6 Reference

The basic algorithm is described in

N. I. M. Gould and D. P. Robinson, ``A dual gradient-projection method for large-scale strictly-convex quadratic problems'', Computational Optimization and Applications **67(1)** (2017) 1-38.

1.1.7 Call order

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

- dqp_initialize - provide default control parameters and set up initial data structures
- dqp_read_specfile (optional) - override control values by reading replacement values from a file
- dqp_import - set up problem data structures and fixed values
- dqp_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
  - dqp_solve_qp - solve the quadratic program
  - dqp_solve_sldqp - solve the shifted least-distance problem
- dqp_information (optional) - recover information about the solution and solution process
- dqp_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 dqp.h File Reference

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

Data Structures

- struct dqp_control_type
- struct dqp_time_type
- struct dqp_inform_type

Functions

- void dqp_initialize (void **data, struct dqp_control_type *control, int *status)
- void dqp_read_specfile (struct dqp_control_type *control, const char specfile[ ])
- void dqp_import (struct dqp_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 dqp_reset_control (struct dqp_control_type *control, void **data, int *status)
- void dqp_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[ ], const real_wp_ c_l[ ], const real_wp_ c_u[ ], const real_wp_ x_l[ ], const real_wp_ x_u[ ], real_wp_ x[ ], real_wp_ c[ ], real_wp_ y[ ], real_wp_ z[ ], int x_stat[ ], int c_stat[ ])
- void dqp_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[ ], const real_wp_ c_l[ ], const real_wp_ c_↩ u[ ], const real_wp_ x_l[ ], const real_wp_ x_u[ ], real_wp_ x[ ], real_wp_ c[ ], real_wp_ y[ ], real_wp_ z[ ], int x_stat[ ], int c_stat[ ])
- void dqp_information (void **data, struct dqp_inform_type *inform, int *status)
- void dqp_terminate (void **data, struct dqp_control_type *control, struct dqp_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct dqp_control_type

control derived type as a C struct

Examples

dqpt.c, and dqptf.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	start_print	any printing will start on this iteration
int	stop_print	any printing will stop on this iteration
int	print_gap	printing will only occur every print_gap iterations
int	dual_starting_point	which starting point should be used for the dual problem • -1 user supplied comparing primal vs dual variables • 0 user supplied • 1 minimize linearized dual • 2 minimize simplified quadratic dual • 3 all free (= all active primal costraints) • 4 all fixed on bounds (= no active primal costraints)
int	maxit	at most maxit inner iterations are allowed
int	max_sc	the maximum permitted size of the Schur complement before a refactorization is performed (used in the case where there is no Fredholm Alternative, 0 = refactor every iteration)
int	cauchy_only	a subspace step will only be taken when the current Cauchy step has changed no more than than cauchy_only active constraints; the subspace step will always be taken if cauchy_only < 0
int	arc_search_maxit	how many iterations are allowed per arc search (-ve = as many as require
int	cg_maxit	how many CG iterations to perform per DQP iteration (-ve reverts to n+1)

Data Fields

int	explore_optimal_subspace	once a potentially optimal subspace has been found, investigate it • 0 as per an ordinary subspace • 1 by increasing the maximum number of allowed CG iterations • 2 by switching to a direct method
int	restore_problem	indicate whether and how much of the input problem should be restored on output. Possible values are • 0 nothing restored • 1 scalar and vector parameters • 2 all parameters
int	sif_file_device	specifies the unit number to write generated SIF file describing the current problem
int	qplib_file_device	specifies the unit number to write generated QPLIB file describing the current problem
real_wp_	rho	the penalty weight, rho. The general constraints are not enforced explicitly, but instead included in the objective as a penalty term weighted by rho when rho $>$ 0. If rho $<=$ 0, the general constraints are explicit (that is, there is no penalty term in the objective function)
real_wp_	infinity	any bound larger than infinity in modulus will be regarded as infinite
real_wp_	stop_abs_p	the required absolute and relative accuracies for the primal infeasibilies
real_wp_	stop_rel_p	see stop_abs_p
real_wp_	stop_abs_d	the required absolute and relative accuracies for the dual infeasibility
real_wp_	stop_rel_d	see stop_abs_d
real_wp_	stop_abs_c	the required absolute and relative accuracies for the complementarity
real_wp_	stop_rel_c	see stop_abs_c
real_wp_	stop_cg_relative	the CG iteration will be stopped as soon as the current norm of the preconditioned gradient is smaller than max(stop_cg_relative $*$ initial preconditioned gradient, stop_cg_absolute)
real_wp_	stop_cg_absolute	see stop_cg_relative
real_wp_	cg_zero_curvature	threshold below which curvature is regarded as zero if CG is used
real_wp_	max_growth	maximum growth factor allowed without a refactorization
real_wp_	identical_bounds_tol	any pair of constraint bounds (c_l,c_u) or (x_l,x_u) that are closer than identical_bounds_tol will be reset to the average of their values
real_wp_	cpu_time_limit	the maximum CPU time allowed (-ve means infinite)

Data Fields

real_wp_	clock_time_limit	the maximum elapsed clock time allowed (-ve means infinite)
real_wp_	initial_perturbation	the initial penalty weight (for DLP only)
real_wp_	perturbation_reduction	the penalty weight reduction factor (for DLP only)
real_wp_	final_perturbation	the final penalty weight (for DLP only)
bool	factor_optimal_matrix	are the factors of the optimal augmented matrix required? (for DLP only)
bool	remove_dependencies	the equality constraints will be preprocessed to remove any linear dependencies if true
bool	treat_zero_bounds_as_general	any problem bound with the value zero will be treated as if it were a general value if true
bool	exact_arc_search	if .exact_arc_search is true, an exact piecewise arc search will be performed. Otherwise an ineaxt search using a backtracing Armijo strategy will be employed
bool	subspace_direct	if .subspace_direct is true, the subspace step will be calculated using a direct (factorization) method, while if it is false, an iterative (conjugate-gradient) method will be used.
bool	subspace_alternate	if .subspace_alternate is true, the subspace step will alternate between a direct (factorization) method and an iterative (GLTR conjugate-gradient) method. This will override .subspace_direct
bool	subspace_arc_search	if .subspace_arc_search is true, a piecewise arc search will be performed along the subspace step. Otherwise the search will stop at the firstconstraint encountered
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
bool	generate_qplib_file	if .generate_qplib_file is .true. if a QPLIB file describing the current problem is to be generated
char	symmetric_linear_solver[31]	indefinite linear equation solver set in symmetric_linear_solver
char	definite_linear_solver[31]	definite linear equation solver
char	unsymmetric_linear_solver[31]	unsymmetric linear equation solver
char	sif_file_name[31]	name of generated SIF file containing input problem
char	qplib_file_name[31]	name of generated QPLIB 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'

Data Fields

struct sls_control_type	sls_control	control parameters for FDC struct fdc_control_type fdc_control; control parameters for SLS
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 dqp_time_type

time derived type as a C struct

Data Fields

real_wp_	total	the total CPU time spent in the package
real_wp_	preprocess	the CPU time spent preprocessing the problem
real_wp_	find_dependent	the CPU time spent detecting linear dependencies
real_wp_	analyse	the CPU time spent analysing the required matrices prior to factorization
real_wp_	factorize	the CPU time spent factorizing the required matrices
real_wp_	solve	the CPU time spent computing the search direction
real_wp_	search	the CPU time spent in the linesearch
real_wp_	clock_total	the total clock time spent in the package
real_wp_	clock_preprocess	the clock time spent preprocessing the problem
real_wp_	clock_find_dependent	the clock time spent detecting linear dependencies
real_wp_	clock_analyse	the clock time spent analysing the required matrices prior to factorization
real_wp_	clock_factorize	the clock time spent factorizing the required matrices
real_wp_	clock_solve	the clock time spent computing the search direction
real_wp_	clock_search	the clock time spent in the linesearch

3.1.1.3 struct dqp_inform_type

inform derived type as a C struct

Examples

dqpt.c, and dqptf.c.

Data Fields

int	status	return status. See DQP_solve for details
int	alloc_status	the status of the last attempted allocation/deallocation
char	bad_alloc[81]	the name of the array for which an allocation/deallocation error ocurred
int	iter	the total number of iterations required
int	cg_iter	the total number of iterations required
int	factorization_status	the return status from the factorization
int	factorization_integer	the total integer workspace required for the factorization

Data Fields

int	factorization_real	the total real workspace required for the factorization
int	nfacts	the total number of factorizations performed
int	threads	the number of threads used
real_wp_	obj	the value of the objective function at the best estimate of the solution determined by DQP_solve
real_wp_	primal_infeasibility	the value of the primal infeasibility
real_wp_	dual_infeasibility	the value of the dual infeasibility
real_wp_	complementary_slackness	the value of the complementary slackness
real_wp_	non_negligible_pivot	the smallest pivot that was not judged to be zero when detecting linearly dependent constraints
bool	feasible	is the returned "solution" feasible?
int	checkpointsIter[16]	checkpoints(i) records the iteration at which the criticality measures first fall below 10^{-i}, i = 1, ..., 16 (-1 means not achieved)
real_wp_	checkpointsTime[16]	see checkpointsIter
struct dqp_time_type	time	timings (see above)
struct sls_inform_type	sls_inform	inform parameters for FDC struct fdc_inform_type fdc_inform; inform parameters for SLS
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 dqp_initialize()

```
void dqp_initialize (
            void ** data,
            struct dqp_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 dqp_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

dqpt.c, and dqptf.c.

3.1.2.2 dqp_read_specfile()

```
void dqp_read_specfile (
             struct dqp_control_type * control,
             const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters

Parameters

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

3.1.2.3 dqp_import()

```
void dqp_import (
             struct dqp_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 dqp_control_type)
in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 or m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal', 'scaled_identity' or 'identity' 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', or 'identity'; 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.
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

dqpt.c, and dqptf.c.

3.1.2.4 dqp_reset_control()

```
void dqp_reset_control (
            struct dqp_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 dqp_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 dqp_solve_qp()

```
void dqp_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[],
            const real_wp_ c_l[],
            const real_wp_ c_u[],
            const real_wp_ x_l[],
            const real_wp_ x_u[],
            real_wp_ x[],
            real_wp_ c[],
            real_wp_ y[],
            real_wp_ z[],
            int x_stat[],
            int c_stat[] )
```

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. 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 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. • -5. The simple-bound constraints are inconsistent. • -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.

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 in any of the available storage schemes.
in	*c_l*	is a one-dimensional array of size m and type double, that holds the lower bounds c^l on the constraints Ax. The i-th component of c_l, i = 0, ... , m-1, contains c_i^l.
in	*c_u*	is a one-dimensional array of size m and type double, that holds the upper bounds c^l on the constraints Ax. The i-th component of c_u, i = 0, ... , m-1, contains c_i^u.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x_j^l.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x_j^l.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
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 general linear constraints. The j-th component of y, j = 0, ... , n-1, contains y_j.
in,out	*z*	is a one-dimensional array of size n and type double, that holds the values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
out	*x_stat*	is a one-dimensional array of size n and type int, that gives the optimal status of the problem variables. If x_stat(j) is negative, the variable x_j most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.
out	*c_stat*	is a one-dimensional array of size m and type int, that gives the optimal status of the general linear constraints. If c_stat(i) is negative, the constraint value $a_i^T x$ most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.

Examples

dqpt.c, and dqptf.c.

3.1.2.6 dqp_solve_sldqp()

```
void dqp_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[],
            const real_wp_ c_l[],
            const real_wp_ c_u[],
```

```
const real_wp_ x_l[],
const real_wp_ x_u[],
real_wp_ x[],
real_wp_ c[],
real_wp_ y[],
real_wp_ z[],
int x_stat[],
int c_stat[] )
```

Solve the shifted least-distance quadratic program

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. 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 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. • -5. The simple-bound constraints are inconsistent. • -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.

Parameters

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_l*	is a one-dimensional array of size m and type double, that holds the lower bounds c^l on the constraints Ax. The i-th component of c_l, i = 0, ... , m-1, contains c_i^l.
in	*c_u*	is a one-dimensional array of size m and type double, that holds the upper bounds c^l on the constraints Ax. The i-th component of c_u, i = 0, ... , m-1, contains c_i^u.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x_j^l.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x_j^l.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
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 general linear constraints. The j-th component of y, j = 0, ... , n-1, contains y_j.
in,out	*z*	is a one-dimensional array of size n and type double, that holds the values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
out	*x_stat*	is a one-dimensional array of size n and type int, that gives the optimal status of the problem variables. If x_stat(j) is negative, the variable x_j most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.
out	*c_stat*	is a one-dimensional array of size m and type int, that gives the optimal status of the general linear constraints. If c_stat(i) is negative, the constraint value $a_i^T x$ most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.

Examples

dqpt.c, and dqptf.c.

3.1.2.7 dqp_information()

```
void dqp_information (
            void ** data,
            struct dqp_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 dqp_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

dqpt.c, and dqptf.c.

3.1.2.8 dqp_terminate()

```
void dqp_terminate (
            void ** data,
            struct dqp_control_type * control,
            struct dqp_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 dqp_control_type)
out	*inform*	is a struct containing output information (see dqp_inform_type)

Examples

dqpt.c, and dqptf.c.

Chapter 4

Example Documentation

4.1 dqpt.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`.

```
/* dqpt.c */
/* Full test for the DQP C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "dqp.h"
int main(void) {
    // Derived types
    void *data;
    struct dqp_control_type control;
    struct dqp_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
    double c_l[] = {1.0, 2.0};   // constraint lower bound
    double c_u[] = {2.0, 2.0};   // constraint upper bound
    double x_l[] = {-1.0, - INFINITY, - INFINITY}; // variable lower bound
    double x_u[] = {1.0, INFINITY, 2.0}; // variable upper bound
    // 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 <= 6; d++){
        // Initialize DQP
        dqp_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';
            dqp_import( &control, &data, &status, n, m,
                        "coordinate", H_ne, H_row, H_col, NULL,
                        "coordinate", A_ne, A_row, A_col, NULL );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        printf(" case %1i break\n",d);
        case 2: // sparse by rows
            st = 'R';
            dqp_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 );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            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};
            dqp_import( &control, &data, &status, n, m,
                        "dense", H_ne, NULL, NULL, NULL,
                        "dense", A_ne, NULL, NULL, NULL );
            dqp_solve_qp( &data, &status, n, m, H_dense_ne, H_dense, g, f,
                          A_dense_ne, A_dense, c_l, c_u, x_l, x_u,
                          x, c, y, z, x_stat, c_stat );
            break;
        case 4: // diagonal
            st = 'L';
            dqp_import( &control, &data, &status, n, m,
                        "diagonal", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        case 5: // scaled identity
            st = 'S';
            dqp_import( &control, &data, &status, n, m,
                        "scaled_identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        case 6: // identity
            st = 'I';
            dqp_import( &control, &data, &status, n, m,
                        "identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        }
    dqp_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: DQP_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
    dqp_terminate( &data, &control, &inform );
}
// test shifted least-distance interface
for( int d=1; d <= 1; d++){
    // Initialize DQP
    dqp_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';
                dqp_import( &control, &data, &status, n, m,
                            "shifted_least_distance", H_ne, NULL, NULL, NULL,
                            "coordinate", A_ne, A_row, A_col, NULL );
                dqp_solve_sldqp( &data, &status, n, m, w, x_0, g, f,
                                 A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                                 x_stat, c_stat );
                break;
            }
        dqp_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
                   st, inform.iter, inform.obj, inform.status);
        }else{
            printf("%c: DQP_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
        dqp_terminate( &data, &control, &inform );
    }
}
```

4.2 dqptf.c

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

```
/* dqptf.c */
/* Full test for the DQP C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "dqp.h"
int main(void) {
    // Derived types
    void *data;
    struct dqp_control_type control;
    struct dqp_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
    double c_l[] = {1.0, 2.0};   // constraint lower bound
    double c_u[] = {2.0, 2.0};   // constraint upper bound
    double x_l[] = {-1.0, - INFINITY, - INFINITY}; // variable lower bound
    double x_u[] = {1.0, INFINITY, 2.0}; // variable upper bound
    // 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 <= 6; d++){
        // Initialize DQP
        dqp_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';
            dqp_import( &control, &data, &status, n, m,
                        "coordinate", H_ne, H_row, H_col, NULL,
                        "coordinate", A_ne, A_row, A_col, NULL );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        printf(" case %1i break\n",d);
        case 2: // sparse by rows
            st = 'R';
            dqp_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 );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            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};
            dqp_import( &control, &data, &status, n, m,
                        "dense", H_ne, NULL, NULL, NULL,
                        "dense", A_ne, NULL, NULL, NULL );
            dqp_solve_qp( &data, &status, n, m, H_dense_ne, H_dense, g, f,
                          A_dense_ne, A_dense, c_l, c_u, x_l, x_u,
                          x, c, y, z, x_stat, c_stat );
            break;
        case 4: // diagonal
            st = 'L';
            dqp_import( &control, &data, &status, n, m,
                        "diagonal", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        case 5: // scaled identity
            st = 'S';
            dqp_import( &control, &data, &status, n, m,
                        "scaled_identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        case 6: // identity
            st = 'I';
            dqp_import( &control, &data, &status, n, m,
                        "identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            dqp_solve_qp( &data, &status, n, m, H_ne, H_val, g, f,
                          A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                          x_stat, c_stat );
            break;
        }
    dqp_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: DQP_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
    dqp_terminate( &data, &control, &inform );
}
// test shifted least-distance interface
for( int d=1; d <= 1; d++){
    // Initialize DQP
    dqp_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';
                dqp_import( &control, &data, &status, n, m,
                            "shifted_least_distance", H_ne, NULL, NULL, NULL,
                            "coordinate", A_ne, A_row, A_col, NULL );
                dqp_solve_sldqp( &data, &status, n, m, w, x_0, g, f,
                                 A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                                 x_stat, c_stat );
                break;
            }
        dqp_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i iterations. Optimal objective value = %5.2f status = %1i\n",
                   st, inform.iter, inform.obj, inform.status);
        }else{
            printf("%c: DQP_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
        dqp_terminate( &data, &control, &inform );
    }
}
```

Index