C interfaces to GALAHAD BQPB

Generated by Doxygen 1.8.17

Chapter 1

GALAHAD C package bqpb

1.1 Introduction

1.1.1 Purpose

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

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

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 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-semi-definite matrix H, the vectors g, w, x^0, x^l, x^u and the scalar f are given. Any of the constraint bounds x_j^l and x_j^u may be infinite. Full advantage is taken of any zero coefficients in the matrix H.

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

July 2021, C interface December 2021.

1.1.4 Method

The required solution x necessarily satisfies the primal optimality conditions

$$x^l \leq x \leq x^u, \tag{1}$$

the dual optimality conditions

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

where

$$z = z^l + z^u, \ z^l \geq 0 \ \text{ and } \ z^u \leq 0, \tag{2b}$$

and the complementary slackness conditions

$$(x - x^l)^T z^l = 0 \ \text{ and } \ (x - x^u)^T z^u = 0, \tag{3}$$

where the diagonal matrix W^2 has diagonal entries w_j^2, $j = 1, \ldots, n$, where the vector z is known as the dual variables for the bounds, respectively, and where the vector inequalities hold component-wise.

Primal-dual interior point methods iterate towards a point that satisfies these conditions by ultimately aiming to satisfy (2a) and (3), while ensuring that (1) and (2b) are satisfied as strict inequalities at each stage. Appropriate norms of the amounts by which (2a) and (3) fail to be satisfied are known as the primal and dual infeasibility, and the violation of complementary slackness, respectively. The fact that (1) and (2b) are satisfied as strict inequalities gives such methods their other title, namely interior-point methods.

The method aims at each stage to reduce the overall violation of (2a) and (3), rather than reducing each of the terms individually. Given an estimate $v = (x, c, z, z^l, z^u)$ of the primal-dual variables, a correction $\Delta v = \Delta(x, c, z, z^l, z^u)$ is obtained by solving a suitable linear system of Newton equations for the nonlinear systems (2a) and a parameterized ``residual trajectory'' perturbation of (3); residual trajectories proposed by Zhang (1994) and Zhao and Sun (1999) are possibilities. An improved estimate $v + \alpha\Delta v$ is then used, where the step-size α is chosen as close to 1.0 as possible while ensuring both that (1) and (2b) continue to hold and that the individual components which make up the complementary slackness (3) do not deviate too significantly from their average value. The parameter that controls the perturbation of (3) is ultimately driven to zero.

The Newton equations are solved by applying the GALAHAD matrix factorization package SBLS, but there are options to factorize the matrix as a whole (the so-called "augmented system" approach), to perform a block elimination first (the "Schur-complement" approach), or to let the method itself decide which of the two previous options is more appropriate.

The package is actually just a front-end to the more-sophisticated GALAHAD package CQP that saves users from setting unnecessary arguments.

1.1.5 Reference

The basic algorithm is a generalisation of those of

Y. Zhang (1994), On the convergence of a class of infeasible interior-point methods for the horizontal linear complementarity problem, SIAM J. Optimization 4(1) 208-227,

and

G. Zhao and J. Sun (1999). On the rate of local convergence of high-order infeasible path-following algorithms for the P_* linear complementarity problems, Computational Optimization and Applications 14(1) 293-307,

with many enhancements described by

N. I. M. Gould, D. Orban and D. P. Robinson (2013). Trajectory-following methods for large-scale degenerate convex quadratic programming, Mathematical Programming Computation 5(2) 113-142.

1.1.6 Call order

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

- bqpb_initialize - provide default control parameters and set up initial data structures
- bqpb_read_specfile (optional) - override control values by reading replacement values from a file
- bqpb_import - set up problem data structures and fixed values
- bqpb_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
  - bqpb_solve_qp - solve the bound-constrained quadratic program
  - bqpb_solve_sldqp - solve the bound-constrained shifted least-distance problem
- bqpb_information (optional) - recover information about the solution and solution process
- bqpb_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.7 Symmetric matrix storage formats

The symmetric n by n objective Hessian matrix H may be presented and stored in a variety of convenient 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).

Both C-style (0 based) and fortran-style (1-based) indexing is allowed. Choose `control.f_indexing` as `false` for C style and `true` for fortran style; the discussion below presumes C style, but add 1 to indices for the corresponding fortran version.

Wrappers will automatically convert between 0-based (C) and 1-based (fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing.

1.1.7.1 Dense storage format

The matrix H is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Since H is symmetric, only the lower triangular part (that is the part h_{ij} for $0 \leq j \leq i \leq n-1$) need be held. In this case the lower triangle should be stored by rows, that is component $i * i/2 + j$ of the storage array H_val will hold the value h_{ij} (and, by symmetry, h_{ji}) for $0 \leq j \leq i \leq n-1$.

1.1.7.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne-1$, of H, its row index i, column index j and value h_{ij}, $0 \leq j \leq i \leq n-1$, are stored as the l-th components of the integer arrays H_row and H_col and real array H_val, respectively, while the number of nonzeros is recorded as H_ne = ne. Note that only the entries in the lower triangle should be stored.

1.1.7.3 Sparse row-wise storage format

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

1.1.7.4 Diagonal storage format

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

1.1.7.5 Multiples of the identity storage format

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

1.1.7.6 The identity matrix format

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

1.1.7.7 The zero matrix format

The same is true if H is the zero matrix.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 bqpb.h File Reference

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

Data Structures

- struct bqpb_control_type
- struct bqpb_time_type
- struct bqpb_inform_type

Functions

- void bqpb_initialize (void **data, struct bqpb_control_type *control, int *status)
- void bqpb_read_specfile (struct bqpb_control_type *control, const char specfile[ ])
- void bqpb_import (struct bqpb_control_type *control, void **data, int *status, int n, const char H_type[ ], int H_ne, const int H_row[ ], const int H_col[ ], const int H_ptr[ ])
- void bqpb_reset_control (struct bqpb_control_type *control, void **data, int *status)
- void bqpb_solve_qp (void **data, int *status, int n, int h_ne, const real_wp_ H_val[ ], const real_wp_ g[ ], const real_wp_ f, const real_wp_ x_l[ ], const real_wp_ x_u[ ], real_wp_ x[ ], real_wp_ z[ ], int x_stat[ ])
- void bqpb_solve_sldqp (void **data, int *status, int n, const real_wp_ w[ ], const real_wp_ x0[ ], const real_wp_ g[ ], const real_wp_ f, const real_wp_ x_l[ ], const real_wp_ x_u[ ], real_wp_ x[ ], real_wp_ z[ ], int x_stat[ ])
- void bqpb_information (void **data, struct bqpb_inform_type *inform, int *status)
- void bqpb_terminate (void **data, struct bqpb_control_type *control, struct bqpb_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct bqpb_control_type

control derived type as a C struct

Examples

bqpbt.c, and bqpbtf.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 • $\leq$ 0 gives no output, • = 1 gives a one-line summary for every iteration, • = 2 gives a summary of the inner iteration for each iteration, • $\geq$ 3 gives increasingly verbose (debugging) output
int	start_print	any printing will start on this iteration
int	stop_print	any printing will stop on this iteration
int	maxit	at most maxit inner iterations are allowed
int	infeas_max	the number of iterations for which the overall infeasibility of the problem is not reduced by at least a factor .reduce_infeas before the problem is flagged as infeasible (see reduce_infeas)
int	muzero_fixed	the initial value of the barrier parameter will not be changed for the first muzero_fixed iterations
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	indicator_type	specifies the type of indicator function used. Pssible values are • 1 primal indicator: a constraint is active if and only if the distance to its nearest bound $\leq$ .indicator_p_tol • 2 primal-dual indicator: a constraint is active if and only if the distance to its nearest bound $\leq$ .indicator_tol_pd $*$ size of corresponding multiplier • 3 primal-dual indicator: a constraint is active if and only if the distance to its nearest bound $\leq$ .indicator_tol_tapia $*$ distance to same bound at previous iteration

Data Fields

int	arc	which residual trajectory should be used to aim from the current iterate to the solution. Possible values are • 1 the Zhang linear residual trajectory • 2 the Zhao-Sun quadratic residual trajectory • 3 the Zhang arc ultimately switching to the Zhao-Sun residual trajectory • 4 the mixed linear-quadratic residual trajectory • 5 the Zhang arc ultimately switching to the mixed linear-quadratic residual trajectory
int	series_order	the order of (Taylor/Puiseux) series to fit to the path data
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_	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 infeasibility
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_	perturb_h	.perturb_h will be added to the Hessian
real_wp_	prfeas	initial primal variables will not be closer than .prfeas from their bounds
real_wp_	dufeas	initial dual variables will not be closer than .dufeas from their bounds
real_wp_	muzero	the initial value of the barrier parameter. If muzero is not positive, it will be reset to an appropriate value
real_wp_	tau	the weight attached to primal-dual infeasibility compared to complementa when assessing step acceptance
real_wp_	gamma_c	individual complementarities will not be allowed to be smaller than gamma_c times the average value
real_wp_	gamma_f	the average complementarity will not be allowed to be smaller than gamma_f times the primal/dual infeasibility

Data Fields

real_wp_	reduce_infeas	if the overall infeasibility of the problem is not reduced by at least a factor .reduce_infeas over .infeas_max iterations, the problem is flagged as infeasible (see infeas_max)
real_wp_	obj_unbounded	if the objective function value is smaller than obj_unbounded, it will be flagged as unbounded from below.
real_wp_	potential_unbounded	if W=0 and the potential function value is smaller than .potential_unbounded $*$ number of one-sided bounds, the analytic center will be flagged as unbounded
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_	mu_lunge	start terminal extrapolation when mu reaches mu_lunge
real_wp_	indicator_tol_p	if .indicator_type = 1, a constraint/bound will be deemed to be active if and only if the distance to its nearest bound $\leq$ .indicator_p_tol
real_wp_	indicator_tol_pd	if .indicator_type = 2, a constraint/bound will be deemed to be active if and only if the distance to its nearest bound $\leq$ .indicator_tol_pd $*$ size of corresponding multiplier
real_wp_	indicator_tol_tapia	if .indicator_type = 3, a constraint/bound will be deemed to be active if and only if the distance to its nearest bound $\leq$ .indicator_tol_tapia $*$ distance to same bound at previous iteration
real_wp_	cpu_time_limit	the maximum CPU time allowed (-ve means infinite)
real_wp_	clock_time_limit	the maximum elapsed clock time allowed (-ve means infinite)
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	treat_separable_as_general	if .just_feasible is true, the algorithm will stop as soon as a feasible point is found. Otherwise, the optimal solution to the problem will be found
bool	just_feasible	if .treat_separable_as_general, is true, any separability in the problem structure will be ignored
bool	getdua	if .getdua, is true, advanced initial values are obtained for the dual variables
bool	puiseux	decide between Puiseux and Taylor series approximations to the arc
bool	every_order	try every order of series up to series_order?
bool	feasol	if .feasol is true, the final solution obtained will be perturbed so that variables close to their bounds are moved onto these bounds
bool	balance_initial_complentarity	if .balance_initial_complentarity is true, the initial complemetarity is required to be balanced

Data Fields

bool	crossover	if .crossover is true, cross over the solution to one defined by linearly-independent constraints if possible
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	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'
struct sbls_control_type	sbls_control	control parameters for FDC struct fdc_control_type fdc_control; control parameters for SBLS

3.1.1.2 struct bqpb_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_	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

3.1.1.3 struct bqpb_inform_type

inform derived type as a C struct

Examples

bqpbt.c, and bqpbtf.c.

Data Fields

int	status	return status. See BQPB_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	factorization_status	the return status from the factorization
long int	factorization_integer	the total integer workspace required for the factorization
long int	factorization_real	the total real workspace required for the factorization
int	nfacts	the total number of factorizations performed
int	nbacts	the total number of "wasted" function evaluations during the linesearch
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 BQPB_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_	init_primal_infeasibility	these values at the initial point (needed by GALAHAD_CCQP)
real_wp_	init_dual_infeasibility	see init_primal_infeasibility
real_wp_	init_complementary_slackness	see init_primal_infeasibility
real_wp_	potential	the value of the logarithmic potential function sum -log(distance to constraint boundary)
real_wp_	non_negligible_pivot	the smallest pivot which was not judged to be zero when detecting linear 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 bqpb_time_type	time	timings (see above)
struct sbls_inform_type	sbls_inform	inform parameters for FDC struct fdc_inform_type fdc_inform; inform parameters for SBLS

3.1.2 Function Documentation

3.1.2.1 bqpb_initialize()

```
void bqpb_initialize (
            void ** data,
```

```
            struct bqpb_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 bqpb_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

bqpbt.c, and bqpbtf.c.

3.1.2.2 bqpb_read_specfile()

```
void bqpb_read_specfile (
            struct bqpb_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 bqpb_control_type)
in	*specfile*	is a character string containing the name of the specification file

3.1.2.3 bqpb_import()

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

Import problem data into internal storage prior to solution.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see bqpb_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 restriction n $>$ 0 or requirement that a H_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.

Examples

bqpbt.c, and bqpbtf.c.

3.1.2.4 bqpb_reset_control()

```
void bqpb_reset_control (
            struct bqpb_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 bqpb_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 bqpb_solve_qp()

```
void bqpb_solve_qp (
            void ** data,
            int * status,
            int n,
            int h_ne,
            const real_wp_ H_val[],
            const real_wp_ g[],
            const real_wp_ f,
            const real_wp_ x_l[],
            const real_wp_ x_u[],
            real_wp_ x[],
            real_wp_ z[],
            int x_stat[] )
```

Solve the bound-constrained quadratic program when the Hessian H is available.

Parameters

in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful. • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction n $>$ 0 or requirement that a H_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. • -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	*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	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x^l_j.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x^l_j.

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	*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.

Examples

bqpbt.c, and bqpbtf.c.

3.1.2.6 bqpb_solve_sldqp()

```
void bqpb_solve_sldqp (
            void ** data,
            int * status,
            int n,
            const real_wp_ w[],
            const real_wp_ x0[],
            const real_wp_ g[],
            const real_wp_ f,
            const real_wp_ x_l[],
            const real_wp_ x_u[],
            real_wp_ x[],
            real_wp_ z[],
            int x_stat[] )
```

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. 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 restriction n $>$ 0 or requirement that h_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. • -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	*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	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x^l_j.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x^l_j.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.

Parameters

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.

Examples

bqpbt.c, and bqpbtf.c.

3.1.2.7 bqpb_information()

```
void bqpb_information (
            void ** data,
            struct bqpb_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 bqpb_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

bqpbt.c, and bqpbtf.c.

3.1.2.8 bqpb_terminate()

```
void bqpb_terminate (
            void ** data,
            struct bqpb_control_type * control,
            struct bqpb_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 bqpb_control_type)
out	*inform*	is a struct containing output information (see bqpb_inform_type)

Examples

bqpbt.c, and bqpbtf.c.

Chapter 4

Example Documentation

4.1 bqpbt.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`.

```
/* bqpbt.c */
/* Full test for the BQPB C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "bqpb.h"
int main(void) {
    // Derived types
    void *data;
    struct bqpb_control_type control;
    struct bqpb_inform_type inform;
    // Set problem data
    int n = 3; // dimension
    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[] = {2.0, 0.0, 0.0};   // linear term in the objective
    double f = 1.0;  // constant term in the objective
    double x_l[] = {-1.0, - INFINITY, - INFINITY}; // variable lower bound
    double x_u[] = {1.0, INFINITY, 2.0}; // variable upper bound
    // Set output storage
    int x_stat[n]; // variable 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 BQPB
        bqpb_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 z[] = {0.0,0.0,0.0};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                bqpb_import( &control, &data, &status, n,
                             "coordinate", H_ne, H_row, H_col, NULL );
                bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
            printf(" case %1i break\n",d);
            case 2: // sparse by rows
                st = 'R';
                bqpb_import( &control, &data, &status, n,
```

```
                        "sparse_by_rows", H_ne, NULL, H_col, H_ptr );
            bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                           x_l, x_u, x, z, x_stat );
            break;
        case 3: // dense
            st = 'D';
            int H_dense_ne = 6; // number of elements of H
            double H_dense[] = {1.0, 0.0, 1.0, 0.0, 0.0, 1.0};
            bqpb_import( &control, &data, &status, n,
                         "dense", H_ne, NULL, NULL, NULL );
            bqpb_solve_qp( &data, &status, n, H_dense_ne, H_dense, g, f,
                           x_l, x_u, x, z, x_stat );
            break;
        case 4: // diagonal
            st = 'L';
            bqpb_import( &control, &data, &status, n,
                         "diagonal", H_ne, NULL, NULL, NULL );
            bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                           x_l, x_u, x, z, x_stat );
            break;
        case 5: // scaled identity
            st = 'S';
            bqpb_import( &control, &data, &status, n,
                         "scaled_identity", H_ne, NULL, NULL, NULL );
            bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                           x_l, x_u, x, z, x_stat );
            break;
        case 6: // identity
            st = 'I';
            bqpb_import( &control, &data, &status, n,
                         "identity", H_ne, NULL, NULL, NULL );
            bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                           x_l, x_u, x, z, x_stat );
            break;
        case 7: // zero
            st = 'Z';
            bqpb_import( &control, &data, &status, n,
                         "zero", H_ne, NULL, NULL, NULL );
            bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                           x_l, x_u, x, z, x_stat );
            break;
        }
    bqpb_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: BQPB_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
    bqpb_terminate( &data, &control, &inform );
}
// test shifted least-distance interface
for( int d=1; d <= 1; d++){
    // Initialize BQPB
    bqpb_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 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';
            bqpb_import( &control, &data, &status, n,
                       "shifted_least_distance", H_ne, NULL, NULL, NULL );
            bqpb_solve_sldqp( &data, &status, n, w, x_0, g, f,
                              x_l, x_u, x, z, x_stat );
            break;
        }
    bqpb_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: BQPB_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
        bqpb_terminate( &data, &control, &inform );
    }
}
```

4.2 bqpbtf.c

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

```
/* bqpbtf.c */
/* Full test for the BQPB C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "bqpb.h"
int main(void) {
    // Derived types
    void *data;
    struct bqpb_control_type control;
    struct bqpb_inform_type inform;
    // Set problem data
    int n = 3; // dimension
    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[] = {2.0, 0.0, 0.0};   // linear term in the objective
    double f = 1.0;  // constant term in the objective
    double x_l[] = {-1.0, - INFINITY, - INFINITY}; // variable lower bound
    double x_u[] = {1.0, INFINITY, 2.0}; // variable upper bound
    // Set output storage
    int x_stat[n]; // variable 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 BQPB
        bqpb_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 z[] = {0.0,0.0,0.0};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                bqpb_import( &control, &data, &status, n,
                             "coordinate", H_ne, H_row, H_col, NULL );
                bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
            printf(" case %1i break\n",d);
            case 2: // sparse by rows
                st = 'R';
                bqpb_import( &control, &data, &status, n,
                             "sparse_by_rows", H_ne, NULL, H_col, H_ptr );
                bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
            case 3: // dense
                st = 'D';
                int H_dense_ne = 6; // number of elements of H
                double H_dense[] = {1.0, 0.0, 1.0, 0.0, 0.0, 1.0};
                bqpb_import( &control, &data, &status, n,
                             "dense", H_ne, NULL, NULL, NULL );
                bqpb_solve_qp( &data, &status, n, H_dense_ne, H_dense, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
            case 4: // diagonal
                st = 'L';
                bqpb_import( &control, &data, &status, n,
                             "diagonal", H_ne, NULL, NULL, NULL );
                bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
```

```
            case 5: // scaled identity
                st = 'S';
                bqpb_import( &control, &data, &status, n,
                             "scaled_identity", H_ne, NULL, NULL, NULL );
                bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
            case 6: // identity
                st = 'I';
                bqpb_import( &control, &data, &status, n,
                             "identity", H_ne, NULL, NULL, NULL );
                bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
            case 7: // zero
                st = 'Z';
                bqpb_import( &control, &data, &status, n,
                             "zero", H_ne, NULL, NULL, NULL );
                bqpb_solve_qp( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
                break;
            }
        bqpb_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: BQPB_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
        bqpb_terminate( &data, &control, &inform );
    }
    // test shifted least-distance interface
    for( int d=1; d <= 1; d++){
        // Initialize BQPB
        bqpb_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 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';
                bqpb_import( &control, &data, &status, n,
                           "shifted_least_distance", H_ne, NULL, NULL, NULL );
                bqpb_solve_sldqp( &data, &status, n, w, x_0, g, f,
                                  x_l, x_u, x, z, x_stat );
                break;
            }
        bqpb_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: BQPB_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
        bqpb_terminate( &data, &control, &inform );
    }
}
```

Index