C interfaces to GALAHAD QPB

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

Chapter 1

GALAHAD C package qpb

1.1 Introduction

1.1.1 Purpose

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

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

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 matrix H, the vectors g, 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.

If the matrix H is positive semi-definite, a global solution is found. However, if H is indefinite, the procedure may find a (weak second-order) critical point that is not the global solution to the given problem.

1.1.2 Authors

N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England, and Philippe L. Toint, University of Namur, Belgium.

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

1.1.3 Originally released

December 1999, C interface January 2022.

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) $$Hx + g = A^T y + z$$

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 vectors y and z are known as the Lagrange multipliers for2 the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold component-wise.

1.1.5 Method

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

The problem is solved in two phases. The goal of the first "initial feasible point" phase is to find a strictly interior point which is primal feasible, that is that {1a} is satisfied. The GALAHAD package LSQP is used for this purpose, and offers the options of either accepting the first strictly feasible point found, or preferably of aiming for the so-called "analytic center" of the feasible region. Having found such a suitable initial feasible point, the second "optimality" phase ensures that \req{4.1a} remains satisfied while iterating to satisfy dual feasibility (2a) and complementary slackness (3). The optimality phase proceeds by approximately minimizing a sequence of barrier functions

$$\frac{1}{2} x^T H x + g^T x + f - \mu \left[\sum_{i=1}^{m} \log(c_i - c_i^l) + \sum_{i=1}^{m} \log(c_i^u - c_i) + \sum_{j=1}^{n} \log(x_j - x_j^l) + \sum_{j=1}^{n} \log(x_j^u - x_j) \right],$$

for an approriate sequence of positive barrier parameters μ converging to zero while ensuring that (1a) remain satisfied and that x and c are strictly interior points for (1b). Note that terms in the above sumations corresponding to infinite bounds are ignored, and that equality constraints are treated specially.

Each of the barrier subproblems is solved using a trust-region method. Such a method generates a trial correction step $\Delta(x, c)$ to the current iterate (x, c) by replacing the nonlinear barrier function locally by a suitable quadratic model, and approximately minimizing this model in the intersection of \req{4.1a} and a trust region $\|\Delta(x, c)\| \leq \Delta$ for some appropriate strictly positive trust-region radius Δ and norm $\| \cdot \|$. The step is accepted/rejected and the radius adjusted on the basis of how accurately the model reproduces the value of barrier function at the trial step. If the step proves to be unacceptable, a linesearch is performed along the step to obtain an acceptable new iterate. In practice, the natural primal "Newton" model of the barrier function is frequently less successful than an alternative primal-dual model, and consequently the primal-dual model is usually to be preferred.

Once a barrier subproblem has been solved, extrapolation based on values and derivatives encountered on the central path is optionally used to determine a good starting point for the next subproblem. Traditional Taylor-series

extrapolation has been superceded by more accurate Puiseux-series methods as these are particularly suited to deal with degeneracy.

The trust-region subproblem is approximately solved using the combined conjugate-gradient/Lanczos method implemented in the GALAHAD package GLTR. Such a method requires a suitable preconditioner, and in our case, the only flexibility we have is in approximating the model of the Hessian. Although using a fixed form of preconditioning is sometimes effective, we have provided the option of an automatic choice, that aims to balance the cost of applying the preconditioner against the needs for an accurate solution of the trust-region subproblem. The preconditioner is applied using the GALAHAD matrix factorization package SBLS, but options at this stage are to factorize the preconditioner as a whole (the so-called "augmented system" approach), or to perform a block elimination first (the "Schur-complement" approach). The latter is usually to be prefered when a (non-singular) diagonal preconditioner is used, but may be inefficient if any of the columns of A is too dense.

In order to make the solution as efficient as possible, the variables and constraints are reordered internally by the GALAHAD package QPP prior to solution. In particular, fixed variables, and free (unbounded on both sides) constraints are temporarily removed.

1.1.6 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,

with a number of enhancements described by

A. R. Conn, N. I. M. Gould, D. Orban and Ph. L. Toint (1999). A primal-dual trust-region algorithm for minimizing a non-convex function subject to general inequality and linear equality constraints. Mathematical Programming **87** 215-249.

1.1.7 Call order

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

- qpb_initialize - provide default control parameters and set up initial data structures
- qpb_read_specfile (optional) - override control values by reading replacement values from a file
- qpb_import - set up problem data structures and fixed values
- qpb_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- qpb_solve_qp - solve the quadratic program
- qpb_information (optional) - recover information about the solution and solution process
- qpb_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.

1.1.9.7 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 qpb.h File Reference

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

Data Structures

- struct qpb_control_type
- struct qpb_time_type
- struct qpb_inform_type

Functions

- void qpb_initialize (void **data, struct qpb_control_type *control, int *status)
- void qpb_read_specfile (struct qpb_control_type *control, const char specfile[ ])
- void qpb_import (struct qpb_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 qpb_reset_control (struct qpb_control_type *control, void **data, int *status)
- void qpb_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 qpb_information (void **data, struct qpb_inform_type *inform, int *status)
- void qpb_terminate (void **data, struct qpb_control_type *control, struct qpb_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct qpb_control_type

control derived type as a C struct

Examples

qpbt.c, and qpbtf.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	maxit	at most maxit inner iterations are allowed
int	itref_max	the maximum number of iterative refinements allowed
int	cg_maxit	the maximum number of CG iterations allowed. If cg_maxit $<$ 0, this number will be reset to the dimension of the system + 1
int	indicator_type	specifies the type of indicator function used. Pssible values are • 1 primal indicator: constraint active $<=>$ distance to nearest bound $<=$ .indicator_p_tol • 2 primal-dual indicator: constraint active $<=>$ distance to nearest bound $<=$ .indicator_tol_pd $*$ size of corresponding multiplier • 3 primal-dual indicator: constraint active $<=>$ distance to nearest bound $<=$ .indicator_tol_tapia $*$ distance to same bound at previous iteration
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	extrapolate	should extrapolation be used to track the central path? Possible values • 0 never • 1 after the final major iteration • 2 at each major iteration
int	path_history	the maximum number of previous path points to use when fitting the data
int	factor	the factorization to be used. Possible values are • 0 automatic • 1 Schur-complement factorization • 2 augmented-system factorization

Data Fields

int	max_col	the maximum number of nonzeros in a column of A which is permitted with the Schur-complement factorization
int	indmin	an initial guess as to the integer workspace required by SBLS
int	valmin	an initial guess as to the real workspace required by SBLS
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	precon	the preconditioner to be used for the CG is defined by precon. Possible values are • 0 automatic • 1 no preconditioner, i.e, the identity within full factorization • 2 full factorization • 3 band within full factorization • 4 diagonal using the barrier terms within full factorization
int	nsemib	the semi-bandwidth of a band preconditioner, if appropriate
int	path_derivatives	the maximum order of path derivative to use
int	fit_order	the order of (Puiseux) series to fit to the path data: $<$=0 to fit all data
int	sif_file_device	specifies the unit number to write generated SIF file describing the current problem
real_wp_	infinity	any bound larger than infinity in modulus will be regarded as infinite
real_wp_	stop_p	the required accuracy for the primal infeasibility
real_wp_	stop_d	the required accuracy for the dual infeasibility
real_wp_	stop_c	the required accuracy for the complementarity
real_wp_	theta_d	tolerances used to terminate the inner iteration (for given mu): dual feasibility $<$= MAX(theta_d $*$ mu $**$ beta, 0.99 $*$ stop_d) complementarity $<$= MAX(theta_c $*$ mu $**$ beta, 0.99 $*$ stop_d)
real_wp_	theta_c	see theta_d
real_wp_	beta	see theta_d
real_wp_	prfeas	initial primal variables will not be closer than prfeas from their bound
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

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_	pivot_tol	the threshold pivot used by the matrix factorization. See the documentation for SBLS for details
real_wp_	pivot_tol_for_dependencies	the threshold pivot used by the matrix factorization when attempting to detect linearly dependent constraints. See the documentation for FDC for details
real_wp_	zero_pivot	any pivots smaller than zero_pivot in absolute value will be regarded to zero when attempting to detect linearly dependent constraints
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_	inner_stop_relative	the search direction is considered as an acceptable approximation to the minimizer of the model if the gradient of the model in the preconditioning(inverse) norm is less than max(inner_stop_relative $*$ initial preconditioning(inverse) gradient norm, inner_stop_absolute)
real_wp_	inner_stop_absolute	see inner_stop_relative
real_wp_	initial_radius	the initial trust-region radius
real_wp_	mu_min	start terminal extrapolation when mu reaches mu_min
real_wp_	inner_fraction_opt	a search direction which gives at least inner_fraction_opt times the optimal model decrease will be found
real_wp_	indicator_tol_p	if .indicator_type = 1, a constraint/bound will be deemed to be active $<=>$ distance to nearest bound $<=$ .indicator_p_tol
real_wp_	indicator_tol_pd	if .indicator_type = 2, a constraint/bound will be deemed to be active $<=>$ distance to nearest bound $<=$ .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 $<=>$ distance to nearest bound $<=$ .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

Data Fields

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	center	if .center is true, the algorithm will use the analytic center of the feasible set as its initial feasible point. Otherwise, a feasible point as close as possible to the initial point will be used. We recommend using the analytic center
bool	primal	if .primal, is true, a primal barrier method will be used in place of t primal-dual method
bool	puiseux	If extrapolation is to be used, decide between Puiseux and Taylor series.
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	array_syntax_worse_than_do_loop	if .array_syntax_worse_than_do_loop is true, f77-style do loops will be used rather than f90-style array syntax for vector operations
bool	space_critical	if .space_critical true, every effort will be made to use as little space as possible. This may result in longer computation time
bool	deallocate_error_fatal	if .deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue
bool	generate_sif_file	if .generate_sif_file is .true. if a SIF file describing the current problem is to be generated
char	sif_file_name[31]	name of generated SIF file containing input problem
char	prefix[31]	all output lines will be prefixed by .prefix(2:LEN(TRIM(.prefix))-1) where .prefix contains the required string enclosed in quotes, e.g. "string" or 'string'
struct lsqp_control_type	lsqp_control	control parameters for LSQP
struct fdc_control_type	fdc_control	control parameters for FDC
struct sbls_control_type	sbls_control	control parameters for SBLS
struct gltr_control_type	gltr_control	control parameters for GLTR
struct fit_control_type	fit_control	control parameters for FIT

3.1.1.2 struct qpb_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 factorizatio

Data Fields

real_wp_	factorize	the CPU time spent factorizing the required matrices
real_wp_	solve	the CPU time spent computing the search direction
real_wp_	phase1_total	the total CPU time spent in the initial-point phase of the package
real_wp_	phase1_analyse	the CPU time spent analysing the required matrices prior to factorizatio in the inital-point phase
real_wp_	phase1_factorize	the CPU time spent factorizing the required matrices in the inital-point phase
real_wp_	phase1_solve	the CPU time spent computing the search direction in the inital-point ph
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 factorizat
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_phase1_total	the total clock time spent in the initial-point phase of the package
real_wp_	clock_phase1_analyse	the clock time spent analysing the required matrices prior to factorizat in the inital-point phase
real_wp_	clock_phase1_factorize	the clock time spent factorizing the required matrices in the inital-poi phase
real_wp_	clock_phase1_solve	the clock time spent computing the search direction in the inital-point

3.1.1.3 struct qpb_inform_type

inform derived type as a C struct

Examples

qpbt.c, and qpbtf.c.

Data Fields

int	status	return status. See QPB_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 conjugate gradient iterations required
int	factorization_status	the return status from the factorization
int	factorization_integer	the total integer workspace required for the factorization
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	nmods	the total number of factorizations which were modified to ensure that th matrix was an appropriate preconditioner
real_wp_	obj	the value of the objective function at the best estimate of the solution determined by QPB_solve

Data Fields

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?
struct qpb_time_type	time	timings (see above)
struct lsqp_inform_type	lsqp_inform	inform parameters for LSQP
struct fdc_inform_type	fdc_inform	inform parameters for FDC
struct sbls_inform_type	sbls_inform	inform parameters for SBLS
struct gltr_inform_type	gltr_inform	return information from GLTR
struct fit_inform_type	fit_inform	return information from FIT

3.1.2 Function Documentation

3.1.2.1 qpb_initialize()

```
void qpb_initialize (
            void ** data,
            struct qpb_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 qpb_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

qpbt.c, and qpbtf.c.

3.1.2.2 qpb_read_specfile()

```
void qpb_read_specfile (
            struct qpb_control_type * control,
            const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters. By default, the spcification file will be named RUNQPB.SPC and lie in the current directory. Refer to Table 2.1 in the fortran documentation provided in $GALAHAD/doc/qpb.pdf for a list of keywords that may be set.

Parameters

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

3.1.2.3 qpb_import()

```
void qpb_import (
             struct qpb_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 qpb_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 or m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none' has been violated. • -23. An entry from the strict upper triangle of H has been specified.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints.

Parameters

in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian, H. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none', the latter pair if $H = 0$; lower or upper case variants are allowed.
in	*H_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*H_row*	is a one-dimensional array of size H_ne and type int, that holds the row indices of the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes, and in this case can be NULL.
in	*H_col*	is a one-dimensional array of size H_ne and type int, that holds the column indices of the lower triangular part of H in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense, diagonal or (scaled) identity storage schemes are used, and in this case can be NULL.
in	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of H, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.
in	*A_type*	is a one-dimensional array of type char that specifies the unsymmetric storage scheme used for the constraint Jacobian, A. It should be one of 'coordinate', 'sparse_by_rows' or 'dense; lower or upper case variants are allowed.
in	*A_ne*	is a scalar variable of type int, that holds the number of entries in A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*A_row*	is a one-dimensional array of size A_ne and type int, that holds the row indices of A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
in	*A_col*	is a one-dimensional array of size A_ne and type int, that holds the column indices of A in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL.
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

qpbt.c, and qpbtf.c.

3.1.2.4 qpb_reset_control()

```
void qpb_reset_control (
             struct qpb_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 qpb_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 qpb_solve_qp()

```
void qpb_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

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. Possible exit are: • 0. The run was succesful. • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 and m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none' has been violated. • -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.
in	*A_val*	is a one-dimensional array of size a_ne and type double, that holds the values of the entries of the constraint Jacobian matrix A in any of the available storage schemes.

Parameters

in	*c_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

qpbt.c, and qpbtf.c.

3.1.2.6 qpb_information()

```
void qpb_information (
             void ** data,
             struct qpb_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 qpb_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

qpbt.c, and qpbtf.c.

3.1.2.7 qpb_terminate()

```
void qpb_terminate (
            void ** data,
            struct qpb_control_type * control,
            struct qpb_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 qpb_control_type)
out	*inform*	is a struct containing output information (see qpb_inform_type)

Examples

qpbt.c, and qpbtf.c.

Chapter 4

Example Documentation

4.1 qpbt.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`.

```
/* qpbt.c */
/* Full test for the QPB C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "qpb.h"
int main(void) {
    // Derived types
    void *data;
    struct qpb_control_type control;
    struct qpb_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 <= 7; d++){
        // Initialize QPB
        qpb_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';
                qpb_import( &control, &data, &status, n, m,
                            "coordinate", H_ne, H_row, H_col, NULL,
                            "coordinate", A_ne, A_row, A_col, NULL );
                qpb_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';
                qpb_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 );
                qpb_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};
                qpb_import( &control, &data, &status, n, m,
                            "dense", H_ne, NULL, NULL, NULL,
                            "dense", A_ne, NULL, NULL, NULL );
                qpb_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';
                qpb_import( &control, &data, &status, n, m,
                            "diagonal", H_ne, NULL, NULL, NULL,
                            "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                qpb_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';
                qpb_import( &control, &data, &status, n, m,
                            "scaled_identity", H_ne, NULL, NULL, NULL,
                            "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                qpb_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';
                qpb_import( &control, &data, &status, n, m,
                            "identity", H_ne, NULL, NULL, NULL,
                            "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                qpb_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 7: // zero
                st = 'Z';
                qpb_import( &control, &data, &status, n, m,
                            "zero", H_ne, NULL, NULL, NULL,
                            "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                qpb_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;
            }
        qpb_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: QPB_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
        qpb_terminate( &data, &control, &inform );
    }
}
```

4.2 qpbtf.c

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

```
/* qpbtf.c */
/* Full test for the QPB C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "qpb.h"
int main(void) {
    // Derived types
    void *data;
    struct qpb_control_type control;
    struct qpb_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 <= 7; d++){
        // Initialize QPB
        qpb_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';
                qpb_import( &control, &data, &status, n, m,
                            "coordinate", H_ne, H_row, H_col, NULL,
                            "coordinate", A_ne, A_row, A_col, NULL );
                qpb_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';
                qpb_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 );
                qpb_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};
                qpb_import( &control, &data, &status, n, m,
                            "dense", H_ne, NULL, NULL, NULL,
                            "dense", A_ne, NULL, NULL, NULL );
                qpb_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';
```

```
            qpb_import( &control, &data, &status, n, m,
                        "diagonal", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpb_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';
            qpb_import( &control, &data, &status, n, m,
                        "scaled_identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpb_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';
            qpb_import( &control, &data, &status, n, m,
                        "identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpb_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 7: // zero
            st = 'Z';
            qpb_import( &control, &data, &status, n, m,
                        "zero", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpb_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;
        }
      qpb_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: QPB_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
      qpb_terminate( &data, &control, &inform );
  }
}
```

Index