C interfaces to GALAHAD BQP

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Fri Mar 18 2022

Chapter 1

GALAHAD C package bqp

1.1 Introduction

1.1.1 Purpose

This package uses a preconditioned, projected-gradient method to solve the **convex bound-constrained quadratic programming problem**

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

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 postive semi-definite matrix H, the vectors g, 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; the matrix need not be provided as there are options to obtain matrix-vector products involving H by reverse communication.

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

November 2009, C interface February 2022.

1.1.4 Terminology

The required solution x necessarily satisfies the primal optimality conditions

$$x^l \leq x \leq x^u,$$

the dual optimality conditions

$$Hx + g = z$$

where

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

and the complementary slackness conditions

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

where the vector z is known as the dual variables for the bounds, respectively, and where the vector inequalities hold component-wise.

1.1.5 Method

The method is iterative. Each iteration proceeds in two stages. Firstly, the so-called generalized Cauchy point for the quadratic objective is found. (The purpose of this point is to ensure that the algorithm converges and that the set of bounds which are satisfied as equations at the solution is rapidly identified.) Thereafter an improvement to the objective is sought using either a direct-matrix or truncated conjugate-gradient algorithm.

1.1.6 Reference

This is a specialised version of the method presented in

A. R. Conn, N. I. M. Gould and Ph. L. Toint (1988). Global convergence of a class of trust region algorithms for optimization with simple bounds. SIAM Journal on Numerical Analysis **25** 433-460,

1.1.7 Call order

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

- bqp_initialize - provide default control parameters and set up initial data structures
- bqp_read_specfile (optional) - override control values by reading replacement values from a file
- set up problem data structures and fixed values by caling one of
  - bqp_import - in the case that H is explicitly available
  - bqp_import_without_h - in the case that only the effect of applying H to a vector is possible
- bqp_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
  - bqp_solve_given_h - solve the problem using values of H
  - bqp_solve_reverse_h_prod - solve the problem by returning to the caller for products of H with specified vectors
- bqp_information (optional) - recover information about the solution and solution process
- bqp_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.8 Symmetric matrix storage formats

If it is explicitly available, 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).

1.1.8.1 Dense storage format

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

1.1.8.2 Sparse co-ordinate storage format

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

1.1.8.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row i+1. For the i-th row of H the i-th component of the integer array H_ptr holds the position of the first entry in this row, while H_ptr(n) holds the total number of entries 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.

1.1.8.4 Diagonal storage format

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 bqp.h File Reference

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

Data Structures

- struct bqp_control_type
- struct bqp_time_type
- struct bqp_inform_type

Functions

- void bqp_initialize (void **data, struct bqp_control_type *control, int *status)
- void bqp_read_specfile (struct bqp_control_type *control, const char specfile[ ])
- void bqp_import (struct bqp_control_type *control, void **data, int *status, int n, const char H_type[ ], int ne, const int H_row[ ], const int H_col[ ], const int H_ptr[ ])
- void bqp_import_without_h (struct bqp_control_type *control, void **data, int *status, int n)
- void bqp_reset_control (struct bqp_control_type *control, void **data, int *status)
- void bqp_solve_given_h (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_ y[ ], real_wp_ z[ ], int x_stat[ ])
- void bqp_solve_reverse_h_prod (void **data, int *status, int n, const real_wp_ g[ ], const real_wp_ f, const real_wp_ x_l[ ], const real_wp_ x_u[ ], real_wp_ y[ ], real_wp_ z[ ], int x_stat[ ], real_wp_ v[ ], const real_wp_ prod[ ], int nz_v[ ], int *nz_v_start, int *nz_v_end, const int nz_prod[ ], int nz_prod_end)
- void bqp_information (void **data, struct bqp_inform_type *inform, int *status)
- void bqp_terminate (void **data, struct bqp_control_type *control, struct bqp_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct bqp_control_type

control derived type as a C struct

Examples

bqpt.c, and bqptf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	error	unit number for error and warning diagnostics
int	out	general output unit number
int	print_level	the level of output required
int	start_print	on which iteration to start printing
int	stop_print	on which iteration to stop printing
int	print_gap	how many iterations between printing
int	maxit	how many iterations to perform (-ve reverts to HUGE(1)-1)
int	cold_start	cold_start should be set to 0 if a warm start is required (with variable assigned according to B_stat, see below), and to any other value if the values given in prob.X suffice
int	ratio_cg_vs_sd	the ratio of how many iterations use CG rather steepest descent
int	change_max	the maximum number of per-iteration changes in the working set permitted when allowing CG rather than steepest descent
int	cg_maxit	how many CG iterations to perform per BQP iteration (-ve reverts to n+1)
int	sif_file_device	the unit number to write generated SIF file describing the current probl
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 complementary slackness
real_wp_	identical_bounds_tol	any pair of constraint bounds (x_l,x_u) that are closer than i dentical_bounds_tol will be reset to the average of their values
real_wp_	stop_cg_relative	the CG iteration will be stopped as soon as the current norm of the preconditioned gradient is smaller than max(stop_cg_relative $*$ initial preconditioned gradient, stop_cg_absolut
real_wp_	stop_cg_absolute	see stop_cg_relative
real_wp_	zero_curvature	threshold below which curvature is regarded as zero
real_wp_	cpu_time_limit	the maximum CPU time allowed (-ve = no limit)
bool	exact_arcsearch	exact_arcsearch is true if an exact arcsearch is required, and false if approximation suffices
bool	space_critical	if space_critical is true, every effort will be made to use as little space as possible. This may result in longer computation times
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, a SIF file describing the current problem will be generated
char	sif_file_name[31]	name (max 30 characters) of generated SIF file containing input problem
char	prefix[31]	all output lines will be prefixed by a string (max 30 characters) prefix(2:LEN(TRIM(.prefix))-1) where prefix contains the required string enclosed in quotes, e.g. "string" or 'string'

Data Fields

struct sbls_control_type	sbls_control	control parameters for SBLS

3.1.1.2 struct bqp_time_type

time derived type as a C struct

Data Fields

real_sp_	total	total time
real_sp_	analyse	time for the analysis phase
real_sp_	factorize	time for the factorization phase
real_sp_	solve	time for the linear solution phase

3.1.1.3 struct bqp_inform_type

inform derived type as a C struct

Examples

bqpt.c, and bqptf.c.

Data Fields

int	status	reported return status: • 0 success • -1 allocation error • -2 deallocation error • -3 matrix data faulty (.n $<$ 1, .ne $<$ 0) • -20 alegedly +ve definite matrix is not
int	alloc_status	Fortran STAT value after allocate failure.
int	factorization_status	status return from factorization
int	iter	number of iterations required
int	cg_iter	number of CG iterations required
real_wp_	obj	current value of the objective function
real_wp_	norm_pg	current value of the projected gradient
char	bad_alloc[81]	name of array which provoked an allocate failure
struct bqp_time_type	time	times for various stages
struct sbls_inform_type	sbls_inform	inform values from SBLS

3.1.2 Function Documentation

3.1.2.1 bqp_initialize()

```
void bqp_initialize (
            void ** data,
            struct bqp_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 bqp_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

bqpt.c, and bqptf.c.

3.1.2.2 bqp_read_specfile()

```
void bqp_read_specfile (
            struct bqp_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 bqp_control_type)
in	*specfile*	is a character string containing the name of the specification file

3.1.2.3 bqp_import()

```
void bqp_import (
            struct bqp_control_type * control,
```

```
void ** data,
int * status,
int n,
const char H_type[],
int 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 bqp_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: • 1. The import was succesful, and the package is ready for the solve phase • -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 type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows' or 'diagonal' has been violated.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal' or 'absent', the latter if access to the Hessian is via matrix-vector products; lower or upper case variants are allowed.
in	*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 three schemes.
in	*H_row*	is a one-dimensional array of size 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 ne and type int, that holds the column indices of the lower triangular part of H in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL
in	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of H, as well as the total number of entries 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

bqpt.c, and bqptf.c.

3.1.2.4 bqp_import_without_h()

```
void bqp_import_without_h (
             struct bqp_control_type * control,
             void ** data,
             int * status,
             int n )
```

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 bqp_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: • 1. The import was succesful, and the package is ready for the solve phase • -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 has been violated.
in	*n*	is a scalar variable of type int, that holds the number of variables.

Examples

bqpt.c, and bqptf.c.

3.1.2.5 bqp_reset_control()

```
void bqp_reset_control (
             struct bqp_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 bqp_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: • 1. The import was succesful, and the package is ready for the solve phase

3.1.2.6 bqp_solve_given_h()

```
void bqp_solve_given_h (
             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_ y[],
             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 type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows' or 'diagonal' has been violated. • -4. 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. • -20. The Hessian matrix H appears to be indefinite. specified. • -23. An entry from the strict upper triangle of H has been
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.
in,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

bqpt.c, and bqptf.c.

3.1.2.7 bqp_solve_reverse_h_prod()

```
void bqp_solve_reverse_h_prod (
             void ** data,
             int * status,
             int n,
             const real_wp_ g[],
             const real_wp_ f,
             const real_wp_ x_l[],
             const real_wp_ x_u[],
             real_wp_ y[],
             real_wp_ z[],
             int x_stat[],
             real_wp_ v[],
             const real_wp_ prod[],
             int nz_v[],
             int * nz_v_start,
             int * nz_v_end,
             const int nz_prod[],
             int nz_prod_end )
```

Solve the bound-constrained quadratic program when the products of the Hessian H with specified vectors may be computed by the calling program.

Parameters

in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. Possible exit are: • 0. The run was succesful. • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction n $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows' or 'diagonal' has been violated. • -4. 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. • -20. The Hessian matrix H appears to be indefinite. specified. • -23. An entry from the strict upper triangle of H has been specified.

Parameters

	status	(continued) • 2. The product Hv of the Hessian H with a given output vector v is required from the user. The vector v will be stored in v and the product Hv must be returned in prod, and bqp_solve_reverse_h_prod re-entered with all other arguments unchanged. • 3. The product Hv of the Hessian H with a given output vector v is required from the user. Only components nz_v[nz_v_start-1:nz_v_end-1] of the vector v stored in v are nonzero. The resulting product Hv must be placed in prod, and bqp_solve_reverse_h_prod re-entered with all other arguments unchanged. • 4. The product Hv of the Hessian H with a given output vector v is required from the user. Only components nz_v[nz_v_start-1:nz_v_end-1] of the vector v stored in v are nonzero. The resulting **nonzeros** in the product Hv must be placed in their appropriate comnponents of prod, while a list of indices of the nonzeros placed in nz_prod[0 : nz_prod_end-1]. bqp_solve_reverse_h_prod should then be re-entered with all other arguments unchanged. Typically v will be very sparse (i.e., nz_p_end-nz_p_start will be small).
`in`	*n*	is a scalar variable of type int, that holds the number of variables
`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.
`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.
`in,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`	*v*	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=2-4 above for details)
`in`	*prod*	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=2-4 above for details)
`out`	*nz_v*	is a one-dimensional array of size n and type int, that is used for reverse communication (see status=3-4 above for details)
`out`	*nz_v_start*	is a scalar of type int, that is used for reverse communication (see status=3-4 above for details)
`out`	*nz_v_end*	is a scalar of type int, that is used for reverse communication (see status=3-4 above for details)
`in`	*nz_prod*	is a one-dimensional array of size n and type int, that is used for reverse communication (see status=4 above for details)
`in`	*nz_prod_end*	is a scalar of type int, that is used for reverse communication (see status=4 above for details)

Examples

bqpt.c, and bqptf.c.

3.1.2.8 bqp_information()

```
void bqp_information (
             void ** data,
             struct bqp_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 bqp_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

bqpt.c, and bqptf.c.

3.1.2.9 bqp_terminate()

```
void bqp_terminate (
             void ** data,
             struct bqp_control_type * control,
             struct bqp_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 bqp_control_type)
out	*inform*	is a struct containing output information (see bqp_inform_type)

Examples

bqpt.c, and bqptf.c.

Chapter 4

Example Documentation

4.1 bqpt.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`.

```
/* bqpt.c */
/* Full test for the BQP C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "bqp.h"
int main(void) {
    // Derived types
    void *data;
    struct bqp_control_type control;
    struct bqp_inform_type inform;
    // Set problem data
    int n = 10; // dimension
    int H_ne = 2 * n - 1; // Hesssian elements, NB lower triangle
    int H_dense_ne = n * ( n + 1 ) / 2; // dense Hessian elements
    int H_row[H_ne]; // row indices,
    int H_col[H_ne]; // column indices
    int H_ptr[n+1];  // row pointers
    double H_val[H_ne]; // values
    double H_dense[H_dense_ne]; // dense values
    double H_diag[n];   // diagonal values
    double g[n];  // linear term in the objective
    double f = 1.0;  // constant term in the objective
    double x_l[n]; // variable lower bound
    double x_u[n]; // variable upper bound
    double x[n]; // variables
    double z[n]; // dual variables
    // Set output storage
    int x_stat[n]; // variable status
    char st;
    int i, l, status;
    printf("H_dense_ne = %i\n", H_dense_ne);
    g[0] = 2.0;
    for( int i = 1; i < n; i++) g[i] = 0.0;
    x_l[0] = -1.0;
    for( int i = 1; i < n; i++) x_l[i] = - INFINITY;
    x_u[0] = 1.0;
    x_u[1] = INFINITY;
    for( int i = 2; i < n; i++) x_u[i] = 2.0;
    // H = tridiag(2,1), H_dense = diag(2)
    l = 0 ;
    H_ptr[0] = l;
    H_row[l] = 0; H_col[l] = 0; H_val[l] = 2.0;
    for( int i = 1; i < n; i++)
    {
      l = l + 1;
      H_ptr[i] = l;
      H_row[l] = i; H_col[l] = i - 1; H_val[l] = 1.0;
```

```
  l = l + 1;
  H_row[l] = i; H_col[l] = i; H_val[l] = 2.0;
}
H_ptr[n] = l + 1;
l = - 1 ;
for( int i = 0; i < n; i++)
{
  H_diag[i] = 2.0;
  for( int j = 0; j <= i; j++)
  {
    l = l + 1;
    if ( j < i - 1 ) {
      H_dense[l] = 0.0;
    }
    else if ( j == i - 1 ) {
      H_dense[l] = 1.0;
    }
    else {
      H_dense[l] = 2.0;
    }
  }
}
printf(" C sparse matrix indexing\n\n");
printf(" basic tests of bqp storage formats\n\n");
for( int d=1; d <= 4; d++){
    // Initialize BQP
    bqp_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    // Start from 0
    for( int i = 0; i < n; i++) x[i] = 0.0;
    for( int i = 0; i < n; i++) z[i] = 0.0;
    switch(d){
        case 1: // sparse co-ordinate storage
            st = 'C';
            bqp_import( &control, &data, &status, n,
                        "coordinate", H_ne, H_row, H_col, NULL );
            bqp_solve_given_h( &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';
            bqp_import( &control, &data, &status, n,
                         "sparse_by_rows", H_ne, NULL, H_col, H_ptr );
            bqp_solve_given_h( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
            break;
        case 3: // dense
            st = 'D';
            bqp_import( &control, &data, &status, n,
                         "dense", H_dense_ne, NULL, NULL, NULL );
            bqp_solve_given_h( &data, &status, n, H_dense_ne, H_dense,
                               g, f, x_l, x_u, x, z, x_stat );
            break;
        case 4: // diagonal
            st = 'L';
            bqp_import( &control, &data, &status, n,
                         "diagonal", H_ne, NULL, NULL, NULL );
            bqp_solve_given_h( &data, &status, n, n, H_diag, g, f,
                               x_l, x_u, x, z, x_stat );
            break;
        }
    bqp_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: BQP_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
    bqp_terminate( &data, &control, &inform );
}
printf("\n tests reverse-communication options\n\n");
// reverse-communication input/output
int nz_v_start, nz_v_end, nz_prod_end;
int nz_v[n], nz_prod[n], mask[n];
double v[n], prod[n];
nz_prod_end = 0;
// Initialize BQP
bqp_initialize( &data, &control, &status );
```

```
   // control.print_level = 1;
   // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    // Start from 0
    for( int i = 0; i < n; i++) x[i] = 0.0;
    for( int i = 0; i < n; i++) z[i] = 0.0;
    st = 'I';
    for( int i = 0; i < n; i++) mask[i] = 0;
    bqp_import_without_h( &control, &data, &status, n ) ;
    while(true){ // reverse-communication loop
        bqp_solve_reverse_h_prod( &data, &status, n, g, f, x_l, x_u,
                                  x, z, x_stat, v, prod,
                                  nz_v, &nz_v_start, &nz_v_end,
                                  nz_prod, nz_prod_end );
        if(status == 0){ // successful termination
            break;
        }else if(status < 0){ // error exit
            break;
        }else if(status == 2){ // evaluate Hv
          prod[0] = 2.0 * v[0] + v[1];
          for( int i = 1; i < n-1; i++) prod[i] = 2.0 * v[i] + v[i-1] + v[i+1];
          prod[n-1] = 2.0 * v[n-1] + v[n-2];
        }else if(status == 3){ // evaluate Hv for sparse v
          for( int i = 0; i < n; i++) prod[i] = 0.0;
          for( int l = nz_v_start - 1; l < nz_v_end; l++){
             i = nz_v[l];
             if (i > 0) prod[i-1] = prod[i-1] + v[i];
             prod[i] = prod[i] + 2.0 * v[i];
             if (i < n-1) prod[i+1] = prod[i+1] + v[i];
           }
        }else if(status == 4){ // evaluate sarse Hv for sparse v
          nz_prod_end = 0;
          for( int l = nz_v_start - 1; l < nz_v_end; l++){
             i = nz_v[l];
             if (i > 0){
               if (mask[i-1] == 0){
                 mask[i-1] = 1;
                 nz_prod[nz_prod_end] = i - 1;
                 nz_prod_end = nz_prod_end + 1;
                 prod[i-1] = v[i];
               }else{
                 prod[i-1] = prod[i-1] + v[i];
               }
             }
             if (mask[i] == 0){
               mask[i] = 1;
               nz_prod[nz_prod_end] = i;
               nz_prod_end = nz_prod_end + 1;
               prod[i] = 2.0 * v[i];
             }else{
               prod[i] = prod[i] + 2.0 * v[i];
             }
             if (i < n-1){
               if (mask[i+1] == 0){
                 mask[i+1] = 1;
                 nz_prod[nz_prod_end] = i + 1;
                 nz_prod_end = nz_prod_end + 1;
                 prod[i+1] = prod[i+1] + v[i];
               }else{
                 prod[i+1] = prod[i+1] + v[i];
               }
             }
          }
          for( int l = 0; l < nz_prod_end; l++) mask[nz_prod[l]] = 0;
        }else{
            printf(" the value %1i of status should not occur\n", status);
            break;
        }
    }
    // Record solution information
    bqp_information( &data, &inform, &status );
    // Print solution details
    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: BQP_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
    bqp_terminate( &data, &control, &inform );
}
```

4.2 bqptf.c

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

```
/* bqptf.c */
/* Full test for the BQP C interface using fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "bqp.h"
int main(void) {
    // Derived types
    void *data;
    struct bqp_control_type control;
    struct bqp_inform_type inform;
    // Set problem data
    int n = 10; // dimension
    int H_ne = 2 * n - 1; // Hesssian elements, NB lower triangle
    int H_dense_ne = n * ( n + 1 ) / 2; // dense Hessian elements
    int H_row[H_ne]; // row indices,
    int H_col[H_ne]; // column indices
    int H_ptr[n+1];  // row pointers
    double H_val[H_ne]; // values
    double H_dense[H_dense_ne]; // dense values
    double H_diag[n];    // diagonal values
    double g[n];  // linear term in the objective
    double f = 1.0;  // constant term in the objective
    double x_l[n]; // variable lower bound
    double x_u[n]; // variable upper bound
    double x[n]; // variables
    double z[n]; // dual variables
    // Set output storage
    int x_stat[n]; // variable status
    char st;
    int i, l, status;
    printf("H_dense_ne = %i\n", H_dense_ne);
    g[0] = 2.0;
    for( int i = 1; i < n; i++) g[i] = 0.0;
    x_l[0] = -1.0;
    for( int i = 1; i < n; i++) x_l[i] = - INFINITY;
    x_u[0] = 1.0;
    x_u[1] = INFINITY;
    for( int i = 2; i < n; i++) x_u[i] = 2.0;
    // H = tridiag(2,1), H_dense = diag(2)
    l = 0 ;
    H_ptr[0] = l + 1;
    H_row[l] = 1; H_col[l] = 1; H_val[l] = 2.0;
    for( int i = 1; i < n; i++)
    {
      l = l + 1;
      H_ptr[i] = l + 1;
      H_row[l] = i + 1; H_col[l] = i; H_val[l] = 1.0;
      l = l + 1;
      H_row[l] = i + 1; H_col[l] = i + 1; H_val[l] = 2.0;
    }
    H_ptr[n] = l + 2;
    l = - 1;
    for( int i = 0; i < n; i++)
    {
      H_diag[i] = 2.0;
      for( int j = 0; j <= i; j++)
      {
        l = l + 1;
        if ( j < i - 1 ) {
          H_dense[l] = 0.0;
        }
        else if ( j == i - 1 ) {
          H_dense[l] = 1.0;
        }
        else {
          H_dense[l] = 2.0;
        }
      }
    }
    printf(" fortran sparse matrix indexing\n\n");
    printf(" basic tests of bqp storage formats\n\n");
    for( int d=1; d <= 4; d++){
        // Initialize BQP
        bqp_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = true; // fortran sparse matrix indexing
        // Start from 0
        for( int i = 0; i < n; i++) x[i] = 0.0;
        for( int i = 0; i < n; i++) z[i] = 0.0;
        switch(d){
            case 1: // sparse co-ordinate storage
```

```
            st = 'C';
            bqp_import( &control, &data, &status, n,
                        "coordinate", H_ne, H_row, H_col, NULL );
            bqp_solve_given_h( &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';
            bqp_import( &control, &data, &status, n,
                        "sparse_by_rows", H_ne, NULL, H_col, H_ptr );
            bqp_solve_given_h( &data, &status, n, H_ne, H_val, g, f,
                               x_l, x_u, x, z, x_stat );
            break;
        case 3: // dense
            st = 'D';
            bqp_import( &control, &data, &status, n,
                        "dense", H_dense_ne, NULL, NULL, NULL );
            bqp_solve_given_h( &data, &status, n, H_dense_ne, H_dense,
                               g, f, x_l, x_u, x, z, x_stat );
            break;
        case 4: // diagonal
            st = 'L';
            bqp_import( &control, &data, &status, n,
                        "diagonal", H_ne, NULL, NULL, NULL );
            bqp_solve_given_h( &data, &status, n, n, H_diag, g, f,
                               x_l, x_u, x, z, x_stat );
            break;
        }
    bqp_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: BQP_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
    bqp_terminate( &data, &control, &inform );
}
printf("\n tests reverse-communication options\n\n");
// reverse-communication input/output
int nz_v_start, nz_v_end, nz_prod_end;
int nz_v[n], nz_prod[n], mask[n];
double v[n], prod[n];
nz_prod_end = 0;
// Initialize BQP
bqp_initialize( &data, &control, &status );
// control.print_level = 1;
// Set user-defined control options
control.f_indexing = true; // fortran sparse matrix indexing
// Start from 0
for( int i = 0; i < n; i++) x[i] = 0.0;
for( int i = 0; i < n; i++) z[i] = 0.0;
st = 'I';
for( int i = 0; i < n; i++) mask[i] = 0;
bqp_import_without_h( &control, &data, &status, n ) ;
while(true){ // reverse-communication loop
    bqp_solve_reverse_h_prod( &data, &status, n, g, f, x_l, x_u,
                              x, z, x_stat, v, prod,
                              nz_v, &nz_v_start, &nz_v_end,
                              nz_prod, nz_prod_end );
    if(status == 0){ // successful termination
        break;
    }else if(status < 0){ // error exit
        break;
    }else if(status == 2){ // evaluate Hv
      prod[0] = 2.0 * v[0] + v[1];
      for( int i = 1; i < n-1; i++) prod[i] = 2.0 * v[i] + v[i-1] + v[i+1];
      prod[n-1] = 2.0 * v[n-1] + v[n-2];
    }else if(status == 3){ // evaluate Hv for sparse v
      for( int i = 0; i < n; i++) prod[i] = 0.0;
      for( int l = nz_v_start - 1; l < nz_v_end; l++){
         i = nz_v[l]-1;
         if (i > 0) prod[i-1] = prod[i-1] + v[i];
         prod[i] = prod[i] + 2.0 * v[i];
         if (i < n-1) prod[i+1] = prod[i+1] + v[i];
       }
    }else if(status == 4){ // evaluate sarse Hv for sparse v
      nz_prod_end = 0;
      for( int l = nz_v_start - 1; l < nz_v_end; l++){
```

```
            i = nz_v[l]-1;
            if (i > 0){
              if (mask[i-1] == 0){
                mask[i-1] = 1;
                nz_prod[nz_prod_end] = i - 1;
                nz_prod_end = nz_prod_end + 1;
                prod[i-1] = v[i];
              }else{
                prod[i-1] = prod[i-1] + v[i];
              }
            }
            if (mask[i] == 0){
              mask[i] = 1;
              nz_prod[nz_prod_end] = i;
              nz_prod_end = nz_prod_end + 1;
              prod[i] = 2.0 * v[i];
            }else{
              prod[i] = prod[i] + 2.0 * v[i];
            }
            if (i < n-1){
              if (mask[i+1] == 0){
                mask[i+1] = 1;
                nz_prod[nz_prod_end] = i + 1;
                nz_prod_end = nz_prod_end + 1;
                prod[i+1] = prod[i+1] + v[i];
              }else{
                prod[i+1] = prod[i+1] + v[i];
              }
            }
          }
          for( int l = 0; l < nz_prod_end; l++) mask[nz_prod[l]] = 0;
        }else{
           printf(" the value %1i of status should not occur\n", status);
           break;
        }
    }
    // Record solution information
    bqp_information( &data, &inform, &status );
    // Print solution details
    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: BQP_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
    bqp_terminate( &data, &control, &inform );
}
```

Index