C interfaces to GALAHAD QPA

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Sat Mar 26 2022

1 GALAHAD C package qpa **1**
1.1 Introduction . . . 1
1.1.1 Purpose . . . 1
1.1.2 Authors . . . 2
1.1.3 Originally released . . . 2
1.1.4 Terminology . . . 2
1.1.5 Method . . . 2
1.1.6 Reference . . . 3
1.1.7 Call order . . . 4
1.1.8 Unsymmetric matrix storage formats . . . 4
1.1.8.1 Dense storage format . . . 4
1.1.8.2 Sparse co-ordinate storage format . . . 4
1.1.8.3 Sparse row-wise storage format . . . 5
1.1.9 Symmetric matrix storage formats . . . 5
1.1.9.1 Dense storage format . . . 5
1.1.9.2 Sparse co-ordinate storage format . . . 5
1.1.9.3 Sparse row-wise storage format . . . 5
1.1.9.4 Diagonal storage format . . . 5
1.1.9.5 Multiples of the identity storage format . . . 5
1.1.9.6 The identity matrix format . . . 6
1.1.9.7 The zero matrix format . . . 6

2 File Index **7**
2.1 File List . . . 7

3 File Documentation **9**
3.1 galahad_qpa.h File Reference . . . 9
3.1.1 Data Structure Documentation . . . 10
3.1.1.1 struct qpa_control_type . . . 10
3.1.1.2 struct qpa_time_type . . . 13
3.1.1.3 struct qpa_inform_type . . . 13
3.1.2 Function Documentation . . . 14
3.1.2.1 qpa_initialize() . . . 14
3.1.2.2 qpa_read_specfile() . . . 15
3.1.2.3 qpa_import() . . . 15
3.1.2.4 qpa_reset_control() . . . 17
3.1.2.5 qpa_solve_qp() . . . 17
3.1.2.6 qpa_solve_l1qp() . . . 19
3.1.2.7 qpa_solve_bcl1qp() . . . 22
3.1.2.8 qpa_information() . . . 25
3.1.2.9 qpa_terminate() . . . 25

4 Example Documentation **27**

4.1 qpat.c 27
4.2 qpatf.c 29

Index **33**

Chapter 1

GALAHAD C package qpa

1.1 Introduction

1.1.1 Purpose

This package uses a working-set method to solve the ℓ_1 **quadratic programming problem**

$$(1) \quad \underset{x \in \mathbf{R}^n}{\text{minimize}} \quad q(x) + \rho_g v_g(x) + \rho_b v_b(x)$$

involving the quadratic objective

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

and the infeasibilities

$$v_g(x) = \sum_{i=1}^{m} \max(c_i^l - a_i^T x, 0) + \sum_{i=1}^{m} \max(a_i^T x - c_i^u, 0)$$

and

$$v_b(x) = \sum_{j=1}^{n} \max(x_j^l - x_j, 0) + \sum_{j=1}^{n} \max(x_j - x_j^u, 0),$$

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.

The package may also be used to solve the **quadratic programming problem**

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

subject to the general linear constraints

$$(3) \quad c_i^l \leq a_i^T x \leq c_i^u, \quad i = 1, \ldots, m,$$

and the simple bound constraints

$$(4) \quad x_j^l \leq x_j \leq x_j^u, \quad j = 1, \ldots, n,$$

by automatically adjusting ρ_b in (1).

Similarly, the package is capable of solving the **bound-constrained** ℓ_1 **quadratic programming problem**

$$(5) \quad \underset{x \in \mathbf{R}^n}{\text{minimize}} \quad q(x) + \rho_g v_g(x),$$

subject to the simple bound constraints (4), by automatically adjusting ρ_b in (1).

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.

N.B. In many cases, the alternative GALAHAD quadratic programming package QPB is faster, and thus to be preferred.

1.1.2 Authors

N. I. M. Gould and D. P. Robinson, 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

October 2001, C interface January 2022.

1.1.4 Terminology

The required solution x to (2)-(4) necessarily satisfies the primal optimality conditions

(1a) $$Ax = c$$

and

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

the dual optimality conditions

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

where

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

and the complementary slackness conditions

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

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

1.1.5 Method

At the k-th iteration of the method, an improvement to the value of the merit function $m(x, \rho_g, \rho_b) = q(x) + \rho_g v_g(x) + \rho_b v_b(x)$ at $x = x^{(k)}$ is sought. This is achieved by first computing a search direction $s^{(k)}$, and then setting $x^{(k+1)} = x^{(k)} + \alpha^{(k)} s^{(k)}$, where the stepsize $\alpha^{(k)}$ is chosen as the first local minimizer of $\phi(\alpha) = m(x^{(k)} + \alpha s^{(k)}, \rho_g, \rho_b)$ as α incesases from zero. The stepsize calculation is straightforward, and exploits the fact that $\phi(\alpha)$ is a piecewise quadratic function of α.

The search direction is defined by a subset of the "active" terms in $v(x)$, i.e., those for which $a_i^T x = c_i^l$ or c_i^u (for $i = 1, \ldots, m$) or $x_j = x_j^l$ or x_j^u (for {j=1,\ldots ,n}). The "working" set $W^{(k)}$ is chosen from the active terms, and is such that its members have linearly independent gradients. The search direction $s^{(k)}$ is chosen as an approximate solution of the equality-constrained quadratic program

$$(6) \quad \underset{s \in \mathbf{R}^n}{\text{minimize}} \;\; q(x^{(k)} + s) + \rho_g l_g^{(k)}(s) + \rho_b l_b^{(k)}(s),$$

subject to

$$(7) \;\; a_i^T s = 0, \;\; i \in \{1, \ldots, m\} \cap W^{(k)}, \; \text{and} \; x_j = 0, \;\; i \in \{1, \ldots, n\} \cap W^{(k)},$$

where

$$l_g^{(k)}(s) = - \sum_{\substack{i=1 \\ a_i^T x < c_i^l}}^{m} a_i^T s \; + \sum_{\substack{i=1 \\ a_i^T x > c_i^u}}^{m} a_i^T s$$

and

$$l_b^{(k)}(s) = - \sum_{\substack{j=1 \\ x_j < x_j^l}}^{n} s_j \; + \sum_{\substack{j=1 \\ x_j > x_j^u}}^{n} s_j .$$

The equality-constrained quadratic program (6)-(7) is solved by a projected preconditioned conjugate gradient method. The method terminates either after a prespecified number of iterations, or if the solution is found, or if a direction of infinite descent, along which $q(x^{(k)} + s) + \rho_g l_g^{(k)}(s) + \rho_b l_b^{(k)}(s)$ decreases without bound within the feasible region (7), is located. Succesively more accurate approximations are required as suspected solutions of (1) are approached.

Preconditioning of the conjugate gradient iteration requires the solution of one or more linear systems of the form

$$(8) \quad \begin{pmatrix} M^{(k)} & A^{(k)T} \\ A^{(k)} & 0 \end{pmatrix} \begin{pmatrix} p \\ u \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix},$$

where $M^{(k)}$ is a "suitable" approximation to H and the rows of $A^{(k)}$ comprise the gradients of the terms in the current working set. Rather than recomputing a factorization of the preconditioner at every iteration, a Schur complement method is used, recognising the fact that gradual changes occur to successive working sets. The main iteration is divided into a sequence of "major" iterations. At the start of each major iteration (say, the overall iteration l), a factorization of the current "reference" matrix, that is the matrix

$$(9) \quad \begin{pmatrix} M^{(l)} & A^{(l)T} \\ A^{(l)} & 0 \end{pmatrix}$$

is obtained using the GALAHAD matrix factorization package SLS. This reference matrix may be factorized as a whole (the so-called "augmented system" approach), or by performing a block elimination first (the "Schur-complement" approach). The latter is usually to be preferred when a (non-singular) diagonal preconditioner is used, but may be inefficient if any of the columns of $A^{(l)}$ is too dense. Subsequent iterations within the current major iteration obtain solutions to (8) via the factors of (9) and an appropriate (dense) Schur complement, obtained from the GALAHAD package SCU. The major iteration terminates once the space required to hold the factors of the (growing) Schur complement exceeds a given threshold.

The working set changes by (a) adding an active term encountered during the determination of the stepsize, or (b) the removal of a term if $s = 0$ solves (6)-(7). The decision on which to remove in the latter case is based upon the expected decrease upon the removal of an individual term, and this information is available from the magnitude and sign of the components of the auxiliary vector u computed in (8). At optimality, the components of u for a_i terms will all lie between 0 and ρ_g—and those for the other terms between 0 and ρ_b—and any violation of this rule indicates further progress is possible. The components of u corresonding to the terms involving $a_i^T x$ are sometimes known as Lagrange multipliers (or generalized gradients) and denoted by y, while those for the remaining x_j terms are dual variables and denoted by z.

To solve (2)-(4), a sequence of problems of the form (1) are solved, each with a larger value of ρ_g and/or ρ_b than its predecessor. The required solution has been found once the infeasibilities $v_g(x)$ and $v_b(x)$ have been reduced to zero at the solution of (1) for the given ρ_g and ρ_b.

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 method is described in detail in

N. I. M. Gould and Ph. L. Toint (2001). ``An iterative working-set method for large-scale non-convex quadratic programming''. *Applied* Numerical Mathematics **43 (1-2)** (2002) 109–128.

1.1.7 Call order

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

- qpa_initialize - provide default control parameters and set up initial data structures
- qpa_read_specfile (optional) - override control values by reading replacement values from a file
- qpa_import - set up problem data structures and fixed values
- qpa_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
  - qpa_solve_qp - solve the quadratic program (2)-(4)
  - qpa_solve_l1qp - solve the l1 quadratic program (1)
  - qpa_solve_bcl1qp - solve the bound constrained l1 quadratic program (4)-(5)
- qpa_information (optional) - recover information about the solution and solution process
- qpa_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:

galahad_qpa.h 9

Chapter 3

File Documentation

3.1 galahad_qpa.h File Reference

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

Data Structures

- struct qpa_control_type
- struct qpa_time_type
- struct qpa_inform_type

Functions

- void qpa_initialize (void **data, struct qpa_control_type *control, int *status)
- void qpa_read_specfile (struct qpa_control_type *control, const char specfile[ ])
- void qpa_import (struct qpa_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 qpa_reset_control (struct qpa_control_type *control, void **data, int *status)
- void qpa_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 qpa_solve_l1qp (void **data, int *status, int n, int m, int h_ne, const real_wp_ H_val[ ], const real_wp_ g[ ], const real_wp_ f, const real_wp_ rho_g, const real_wp_ rho_b, 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 qpa_solve_bcl1qp (void **data, int *status, int n, int m, int h_ne, const real_wp_ H_val[ ], const real_wp_ g[ ], const real_wp_ f, const real_wp_ rho_g, 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 qpa_information (void **data, struct qpa_inform_type *inform, int *status)
- void qpa_terminate (void **data, struct qpa_control_type *control, struct qpa_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct qpa_control_type

control derived type as a C struct

Examples

qpat.c, and qpatf.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	factor	the factorization to be used. Possible values are 0 automatic 1 Schur-complement factorization 2 augmented-system factorization
int	max_col	the maximum number of nonzeros in a column of A which is permitted with the Schur-complement factorization
int	max_sc	the maximum permitted size of the Schur complement before a refactorization is performed
int	indmin	an initial guess as to the integer workspace required by SLS (OBSOL
int	valmin	an initial guess as to the real workspace required by SLS (OBSOL
int	itref_max	the maximum number of iterative refinements allowed (OBSOL
int	infeas_check_interval	the infeasibility will be checked for improvement every infeas_check_interval iterations (see infeas_g_improved_by_factor and infeas_b_improved_by_factor below)
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	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

Data Fields

int	full_max_fill	if the ratio of the number of nonzeros in the factors of the reference matrix to the number of nonzeros in the matrix itself exceeds full_max_fill, and the preconditioner is being selected automatically (precon = 0), a banded approximation will be used instead
int	deletion_strategy	the constraint deletion strategy to be used. Possible values are: 0 most violated of all 1 LIFO (last in, first out) k LIFO(k) most violated of the last k in LIFO
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	monitor_residuals	the frequency at which residuals will be monitored
int	cold_start	indicates whether a cold or warm start should be made. Possible values a 0 warm start - the values set in C_stat and B_stat indicate which constraints will be included in the initial working set. 1 cold start from the value set in X; constraints active at X will determine the initial working set. 2 cold start with no active constraints 3 cold start with only equality constraints active 4 cold start with as many active constraints as possible
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_	feas_tol	any constraint violated by less than feas_tol will be considered to be satisfied
real_wp_	obj_unbounded	if the objective function value is smaller than obj_unbounded, it will b flagged as unbounded from below.
real_wp_	increase_rho_g_factor	if the problem is currently infeasible and solve_qp (see below) is .TRUE the current penalty parameter for the general constraints will be increased by increase_rho_g_factor when needed
real_wp_	infeas_g_improved_by_factor	if the infeasibility of the general constraints has not dropped by a fac of infeas_g_improved_by_factor over the previous infeas_check_interval iterations, the current corresponding penalty parameter will be increase
real_wp_	increase_rho_b_factor	if the problem is currently infeasible and solve_qp or solve_within_boun (see below) is .TRUE., the current penalty parameter for the simple boun constraints will be increased by increase_rho_b_factor when needed

Data Fields

real_wp_	infeas_b_improved_by_factor	if the infeasibility of the simple bounds has not dropped by a factor of infeas_b_improved_by_factor over the previous infeas_check_interval iterations, the current corresponding penalty parameter will be increase
real_wp_	pivot_tol	the threshold pivot used by the matrix factorization. See the documentation for SLS for details (OBSOLE
real_wp_	pivot_tol_for_dependencies	the threshold pivot used by the matrix factorization when attempting to detect linearly dependent constraints.
real_wp_	zero_pivot	any pivots smaller than zero_pivot in absolute value will be regarded to zero when attempting to detect linearly dependent constraints (OBSOLE
real_wp_	inner_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_	multiplier_tol	any dual variable or Lagrange multiplier which is less than multiplier_t outside its optimal interval will be regarded as being acceptable when checking for optimality
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	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	solve_qp	if solve_qp is .TRUE., the value of prob.rho_g and prob.rho_b will be increased as many times as are needed to ensure that the output solution is feasible, and thus aims to solve the qoadraic program (2)-(4)
bool	solve_within_bounds	if solve_within_bounds is .TRUE., the value of prob.rho_b will be increased as many times as are needed to ensure that the output solution is feasible with respect to the simple bounds, and thus aims to solve the bound-constrained qoadraic program (4)-(5)
bool	randomize	if randomize is .TRUE., the constraint bounds will be perturbed by small random quantities during the first stage of the solution process. Any randomization will ultimately be removed. Randomization helps when solving degenerate problems

Data Fields

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 (OBSOLE
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	symmetric_linear_solver[31]	indefinite linear equation solver
char	sif_file_name[31]	definite linear equation solver CHARACTER (LEN = 30) :: definite_linear_solver = & "sils" // REPEAT(' ', 26) 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'
bool	each_interval	component specifically for parametric problems (not used at present)
struct sls_control_type	sls_control	control parameters for SLS

3.1.1.2 struct qpa_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_	analyse	the CPU time spent analysing the required matrices prior to factorizatio
real_wp_	factorize	the CPU time spent factorizing the required matrices
real_wp_	solve	the CPU time spent computing the search direction
real_wp_	clock_total	the total clock time spent in the package
real_sp_	clock_preprocess	the clock time spent preprocessing the problem
real_sp_	clock_analyse	the clock time spent analysing the required matrices prior to factorizat
real_sp_	clock_factorize	the clock time spent factorizing the required matrices
real_sp_	clock_solve	the clock time spent computing the search direction

3.1.1.3 struct qpa_inform_type

inform derived type as a C struct

Examples

qpat.c, and qpatf.c.

Data Fields

int	status	return status. See QPA_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	major_iter	the total number of major iterations required
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	nmods	the total number of factorizations which were modified to ensure that th matrix was an appropriate preconditioner
int	num_g_infeas	the number of infeasible general constraints
int	num_b_infeas	the number of infeasible simple-bound constraints
real_wp_	obj	the value of the objective function at the best estimate of the solution determined by QPA_solve
real_wp_	infeas_g	the 1-norm of the infeasibility of the general constraints
real_wp_	infeas_b	the 1-norm of the infeasibility of the simple-bound constraints
real_wp_	merit	the merit function value = obj + rho_g $*$ infeas_g + rho_b $*$ infeas_b
struct qpa_time_type	time	timings (see above)
struct sls_inform_type	sls_inform	inform parameters for SLS

3.1.2 Function Documentation

3.1.2.1 qpa_initialize()

```
void qpa_initialize (
            void ** data,
            struct qpa_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 qpa_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

qpat.c, and qpatf.c.

3.1.2.2 qpa_read_specfile()

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

Parameters

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

3.1.2.3 qpa_import()

```
void qpa_import (
             struct qpa_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 qpa_control_type)
in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 or m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none' has been violated. • -23. An entry from the strict upper triangle of H has been specified.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints.
in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian, H. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none', the latter pair if $H = 0$; lower or upper case variants are allowed.
in	*H_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*H_row*	is a one-dimensional array of size H_ne and type int, that holds the row indices of the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes, and in this case can be NULL.
in	*H_col*	is a one-dimensional array of size H_ne and type int, that holds the column indices of the lower triangular part of H in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense, diagonal or (scaled) identity storage schemes are used, and in this case can be NULL.
in	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of H, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.
in	*A_type*	is a one-dimensional array of type char that specifies the unsymmetric storage scheme used for the constraint Jacobian, A. It should be one of 'coordinate', 'sparse_by_rows' or 'dense; lower or upper case variants are allowed.
in	*A_ne*	is a scalar variable of type int, that holds the number of entries in A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*A_row*	is a one-dimensional array of size A_ne and type int, that holds the row indices of A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
in	*A_col*	is a one-dimensional array of size A_ne and type int, that holds the column indices of A in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL.

Parameters

in	*A_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of A, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.

Examples

qpat.c, and qpatf.c.

3.1.2.4 qpa_reset_control()

```
void qpa_reset_control (
            struct qpa_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 qpa_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 qpa_solve_qp()

```
void qpa_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 (2)-(4).

Parameters

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

Parameters

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.
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^l_i.
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^u_i.
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.
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.
in,out	*x_stat*	is a one-dimensional array of size n and type int, that gives the current 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. On entry, if control.cold_start = 0, x_stat should be set as above to provide a guide to the initial working set.
in,out	*c_stat*	is a one-dimensional array of size m and type int, that gives the current 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. On entry, if control.cold_start = 0, c_stat should be set as above to provide a guide to the initial working set.

Examples

qpat.c, and qpatf.c.

3.1.2.6 qpa_solve_l1qp()

```
void qpa_solve_l1qp (
             void ** data,
```

```
int * status,
int n,
int m,
int h_ne,
const real_wp_ H_val[],
const real_wp_ g[],
const real_wp_ f,
const real_wp_ rho_g,
const real_wp_ rho_b,
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 l_1 quadratic program (1).

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. • -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	*rho_g*	is a scalar of type double, that holds the parameter ρ_g associated with the linear constraints.
in	*rho_b*	is a scalar of type double, that holds the parameter ρ_b associated with the simple bound constraints.

Parameters

in	*a_ne*	is a scalar variable of type int, that holds the number of entries in the constraint Jacobian matrix A.
in	*A_val*	is a one-dimensional array of size a_ne and type double, that holds the values of the entries of the constraint Jacobian matrix A in any of the available storage schemes.
in	*c_l*	is a one-dimensional array of size m and type double, that holds the lower bounds c^l on the constraints Ax. The i-th component of c_l, i = 0, ... , m-1, contains c_i^l.
in	*c_u*	is a one-dimensional array of size m and type double, that holds the upper bounds c^l on the constraints Ax. The i-th component of c_u, i = 0, ... , m-1, contains c_i^u.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x_j^l.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x_j^l.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
in,out	*y*	is a one-dimensional array of size n and type double, that holds the values y of the Lagrange multipliers for the general linear constraints. The j-th component of y, j = 0, ... , n-1, contains y_j.
in,out	*z*	is a one-dimensional array of size n and type double, that holds the values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
in,out	*x_stat*	is a one-dimensional array of size n and type int, that gives the current 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. On entry, if control.cold_start = 0, x_stat should be set as above to provide a guide to the initial working set.
in,out	*c_stat*	is a one-dimensional array of size m and type int, that gives the current 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. On entry, if control.cold_start = 0, c_stat should be set as above to provide a guide to the initial working set.

Examples

qpat.c, and qpatf.c.

3.1.2.7 qpa_solve_bcl1qp()

```
void qpa_solve_bcl1qp (
             void ** data,
             int * status,
             int n,
             int m,
             int h_ne,
             const real_wp_ H_val[],
             const real_wp_ g[],
             const real_wp_ f,
```

```
const real_wp_ rho_g,
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 bound-constrained l_1 quadratic program (4)-(5)

Parameters

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

Parameters

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	*rho_g*	is a scalar of type double, that holds the parameter ρ_g associated with the linear constraints.
in	*a_ne*	is a scalar variable of type int, that holds the number of entries in the constraint Jacobian matrix A.
in	*A_val*	is a one-dimensional array of size a_ne and type double, that holds the values of the entries of the constraint Jacobian matrix A in any of the available storage schemes.
in	*c_l*	is a one-dimensional array of size m and type double, that holds the lower bounds c^l on the constraints Ax. The i-th component of c_l, i = 0, ... , m-1, contains c_i^l.
in	*c_u*	is a one-dimensional array of size m and type double, that holds the upper bounds c^l on the constraints Ax. The i-th component of c_u, i = 0, ... , m-1, contains c_i^u.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x_j^l.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x_j^l.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
in,out	*y*	is a one-dimensional array of size n and type double, that holds the values y of the Lagrange multipliers for the general linear constraints. The j-th component of y, j = 0, ... , n-1, contains y_j.
in,out	*z*	is a one-dimensional array of size n and type double, that holds the values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
in,out	*x_stat*	is a one-dimensional array of size n and type int, that gives the current 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. On entry, if control.cold_start = 0, x_stat should be set as above to provide a guide to the initial working set.
in,out	*c_stat*	is a one-dimensional array of size m and type int, that gives the current 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. On entry, if control.cold_start = 0, c_stat should be set as above to provide a guide to the initial working set.

Examples

qpat.c, and qpatf.c.

3.1.2.8 qpa_information()

```
void qpa_information (
             void ** data,
             struct qpa_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 qpa_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

qpat.c, and qpatf.c.

3.1.2.9 qpa_terminate()

```
void qpa_terminate (
             void ** data,
             struct qpa_control_type * control,
             struct qpa_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 qpa_control_type)
out	*inform*	is a struct containing output information (see qpa_inform_type)

Examples

qpat.c, and qpatf.c.

Chapter 4

Example Documentation

4.1 qpat.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`.

```
/* qpat.c */
/* Full test for the QPA C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_qpa.h"
int main(void) {
    // Derived types
    void *data;
    struct qpa_control_type control;
    struct qpa_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
    double rho_g = 0.1;  // penalty paramter for general constraints
    double rho_b = 0.1;  // penalty paramter for simple bound constraints
    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 QPA
        qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "coordinate", H_ne, H_row, H_col, NULL,
                        "coordinate", A_ne, A_row, A_col, NULL );
            qpa_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';
            qpa_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 );
            qpa_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};
            qpa_import( &control, &data, &status, n, m,
                        "dense", H_ne, NULL, NULL, NULL,
                        "dense", A_ne, NULL, NULL, NULL );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "diagonal", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "scaled_identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "zero", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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;
        }
    qpa_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: QPA_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
    qpa_terminate( &data, &control, &inform );
}
```

```
    printf("\n basic tests of l_1 qp storage formats\n\n");
    qpa_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};
    // solve the l_1qp problem
    qpa_import( &control, &data, &status, n, m,
                "coordinate", H_ne, H_row, H_col, NULL,
                "coordinate", A_ne, A_row, A_col, NULL );
    qpa_solve_l1qp( &data, &status, n, m, H_ne, H_val, g, f, rho_g, rho_b,
                    A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                    x_stat, c_stat );
    qpa_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%s %6i iterations. Optimal objective value = %5.2f status = %1i\n",
                "l1qp  ", inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: QPA_solve exit status = %1i\n", st, inform.status);
    }
    // Start from 0
    for( int i=0; i <= n-1; i++) x[i] = 0.0;
    for( int i=0; i <= m-1; i++) y[i] = 0.0;
    for( int i=0; i <= n-1; i++) z[i] = 0.0;
    // solve the bound constrained l_1qp problem
    qpa_import( &control, &data, &status, n, m,
                "coordinate", H_ne, H_row, H_col, NULL,
                "coordinate", A_ne, A_row, A_col, NULL );
    qpa_solve_bcl1qp( &data, &status, n, m, H_ne, H_val, g, f, rho_g,
                      A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                      x_stat, c_stat );
    qpa_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%s %6i iterations. Optimal objective value = %5.2f status = %1i\n",
                "bcl1qp", inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: QPA_solve exit status = %1i\n", st, inform.status);
    }
    // Delete internal workspace
    qpa_terminate( &data, &control, &inform );
}
```

4.2 qpatf.c

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

```
/* qpatf.c */
/* Full test for the QPA C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_qpa.h"
int main(void) {
    // Derived types
    void *data;
    struct qpa_control_type control;
    struct qpa_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
    double rho_g = 0.1;  // penalty paramter for general constraints
    double rho_b = 0.1;  // penalty paramter for simple bound constraints
    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 QPA
    qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "coordinate", H_ne, H_row, H_col, NULL,
                        "coordinate", A_ne, A_row, A_col, NULL );
            qpa_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';
            qpa_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 );
            qpa_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};
            qpa_import( &control, &data, &status, n, m,
                        "dense", H_ne, NULL, NULL, NULL,
                        "dense", A_ne, NULL, NULL, NULL );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "diagonal", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "scaled_identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "identity", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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';
            qpa_import( &control, &data, &status, n, m,
                        "zero", H_ne, NULL, NULL, NULL,
                        "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
            qpa_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;
        }
    qpa_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: QPA_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
        qpa_terminate( &data, &control, &inform );
    }
    printf("\n basic tests of l_1 qp storage formats\n\n");
    qpa_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};
    // solve the l_1qp problem
    qpa_import( &control, &data, &status, n, m,
                "coordinate", H_ne, H_row, H_col, NULL,
                "coordinate", A_ne, A_row, A_col, NULL );
    qpa_solve_l1qp( &data, &status, n, m, H_ne, H_val, g, f, rho_g, rho_b,
                    A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                    x_stat, c_stat );
    qpa_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%s %6i iterations. Optimal objective value = %5.2f status = %1i\n",
               "l1qp  ", inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: QPA_solve exit status = %1i\n", st, inform.status);
    }
    // Start from 0
    for( int i=0; i <= n-1; i++) x[i] = 0.0;
    for( int i=0; i <= m-1; i++) y[i] = 0.0;
    for( int i=0; i <= n-1; i++) z[i] = 0.0;
    // solve the bound constrained l_1qp problem
    qpa_import( &control, &data, &status, n, m,
                "coordinate", H_ne, H_row, H_col, NULL,
                "coordinate", A_ne, A_row, A_col, NULL );
    qpa_solve_bcl1qp( &data, &status, n, m, H_ne, H_val, g, f, rho_g,
                      A_ne, A_val, c_l, c_u, x_l, x_u, x, c, y, z,
                      x_stat, c_stat );
    qpa_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%s %6i iterations. Optimal objective value = %5.2f status = %1i\n",
               "bcl1qp", inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: QPA_solve exit status = %1i\n", st, inform.status);
    }
    // Delete internal workspace
    qpa_terminate( &data, &control, &inform );
}
```

Index

galahad_qpa.h, 9
    qpa_import, 15
    qpa_information, 24
    qpa_initialize, 14
    qpa_read_specfile, 15
    qpa_reset_control, 17
    qpa_solve_bcl1qp, 22
    qpa_solve_l1qp, 19
    qpa_solve_qp, 17
    qpa_terminate, 25

qpa_control_type, 10
qpa_import
    galahad_qpa.h, 15
qpa_inform_type, 13
qpa_information
    galahad_qpa.h, 24
qpa_initialize
    galahad_qpa.h, 14
qpa_read_specfile
    galahad_qpa.h, 15
qpa_reset_control
    galahad_qpa.h, 17
qpa_solve_bcl1qp
    galahad_qpa.h, 22
qpa_solve_l1qp
    galahad_qpa.h, 19
qpa_solve_qp
    galahad_qpa.h, 17
qpa_terminate
    galahad_qpa.h, 25
qpa_time_type, 13