C interfaces to GALAHAD RPD

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Mon Feb 21 2022

Chapter 1

GALAHAD C package rpd

1.1 Introduction

1.1.1 Purpose

Read and write data for the linear program (LP)

$$\text{minimize } \ g^T x + f \text{ subject to } c_l \leq Ax \leq c_u \text{ and } x_l \leq x \leq x_u,$$

the linear program with quadratic constraints (QCP)

$$\text{minimize } \ g^T x + f \text{ subject to } c_l \leq Ax + \frac{1}{2}\text{vec}(x.H_c.x) \leq c_u \text{ and } x_l \leq x \leq x_u,$$

the bound-constrained quadratic program (BQP)

$$\text{minimize } \ \frac{1}{2}x^T Hx + g^T x + f \text{ subject to } x_l \leq x \leq x_u,$$

the quadratic program (QP)

$$\text{minimize } \ \frac{1}{2}x^T Hx + g^T x + f \text{ subject to } c_l \leq Ax \leq c_u \text{ and } x_l \leq x \leq x_u,$$

or the quadratic program with quadratic constraints (QCQP)

$$\text{minimize } \ \frac{1}{2}x^T Hx + g^T x + f \text{ subject to } c_l \leq Ax + \frac{1}{2}\text{vec}(x.H_c.x) \leq c_u \text{ and } x_l \leq x \leq x_u,$$

where vec $(x.H_c.x)$ is the vector whose i-th component is $x^T(H_c)_i x$ for the i-th constraint, from and to a QPLIB-format data file. Variables may be continuous, binary or integer.

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

January 2006, C interface January 2022.

1.1.4 Reference

The QPBLIB format is defined in

F. Furini, E. Traversi, P. Belotti, A. Frangioni, A. Gleixner, N. Gould, L. Liberti, A. Lodi, R. Misener, H. Mittelmann, N. V. Sahinidis, S. Vigerske and A. Wiegele (2019). QPLIB: a library of quadratic programming instances, Mathematical Programming Computation **11** 237–265.

1.1.5 Call order

To decode a given QPLIB file, functions from the rpd package must be called in the following order:

- rpd_initialize - provide default control parameters and set up initial data structures
- rpd_get_stats - read a given QPLIB file into internal data structures, and report vital statistics
- (optionally, and in any order, where relevant)
  - rpd_get_g - get the objective gradient term g
  - rpd_get_f - get the objective constant term f
  - rpd_get_xlu - get the variable bounds x_l and x_u
  - rpd_get_xlu - get the constraint bounds c_l and c_u
  - rpd_get_h - get the objective Hessian term H
  - rpd_get_a - get the constrain Jacobian term A
  - rpd_get_h_c - get the constraint Hessian terms H_c
  - rpd_get_x_type - determine the type of each variable x
  - rpd_get_x - get initial value of the variable x
  - rpd_get_y - get initial value of Lagrange multipliers y
  - rpd_get_z - get initial value of the dual variables z
- rpd_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.6 Sparse unsymmetric co-ordinate storage format

The unsymmetric m by n constraint matrix A will be output in sparse co-ordinate format.

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.

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.

1.1.7 Sparse symmetric co-ordinate storage format

Likewise, the symmetric n by n objective Hessian matrix H will be returned in a sparse co-ordinate format. 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).

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 Joint sparse symmetric co-ordinate storage format

The symmetric n by n constraint Hessian matrices $(H_c)_i$ are stored as a whole in a joint symmetric co-ordinate storage format. In addition to the row and column indices and values of each lower triangular matrix, record is also kept of the particular constraint invlved.

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne-1$, of H, its constraint index k, row index i, column index j and value $(h_k)_{ij}$, $0 \leq j \leq i \leq n-1$, are stored as the l-th components of the integer arrays H_c_ptr, H_c_row and H_c_col and real array H_c_val, respectively, while the number of nonzeros is recorded as H_c_ne = ne. Note as before that only the entries in the lower triangles should be stored.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 rpd.h File Reference

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

Data Structures

- struct rpd_control_type
- struct rpd_inform_type

Functions

- void rpd_initialize (void **data, struct rpd_control_type *control, int *status)
- void rpd_get_stats (char qplib_file[ ], int qplib_file_len, struct rpd_control_type *control, void **data, int *status, char p_type[4], int *n, int *m, int *h_ne, int *a_ne, int *h_c_ne)
- void rpd_get_g (void **data, int *status, int n, real_wp_ g[ ])
- void rpd_get_f (void **data, int *status, real_wp_ *f)
- void rpd_get_xlu (void **data, int *status, int n, real_wp_ x_l[ ], real_wp_ x_u[ ])
- void rpd_get_clu (void **data, int *status, int m, real_wp_ c_l[ ], real_wp_ c_u[ ])
- void rpd_get_h (void **data, int *status, int h_ne, int h_row[ ], int h_col[ ], real_wp_ h_val[ ])
- void rpd_get_a (void **data, int *status, int a_ne, int a_row[ ], int a_col[ ], real_wp_ a_val[ ])
- void rpd_get_h_c (void **data, int *status, int h_c_ne, int h_c_ptr[ ], int h_c_row[ ], int h_c_col[ ], real_wp_ h_c_val[ ])
- void rpd_get_x_type (void **data, int *status, int n, int x_type[ ])
- void rpd_get_x (void **data, int *status, int n, real_wp_ x[ ])
- void rpd_get_y (void **data, int *status, int m, real_wp_ y[ ])
- void rpd_get_z (void **data, int *status, int n, real_wp_ z[ ])
- void rpd_terminate (void **data, struct rpd_control_type *control, struct rpd_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct rpd_control_type

control derived type as a C struct

Examples

rpdt.c, and rpdtf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	qplib	QPLIB file input stream number.
int	error	error and warning diagnostics occur on stream error
int	out	general output occurs on stream out
int	print_level	the level of output required is specified by print_level • $\leq$ 0 gives no output, • $\geq$ 1 gives increasingly verbose (debugging) output
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

3.1.1.2 struct rpd_inform_type

inform derived type as a C struct

Examples

rpdt.c, and rpdtf.c.

Data Fields

int	status	return status. Possible values are: • 0 successful return • -1 allocation failure • -2 deallocation failure • -3 end of file reached prematurely • -4 other read error • -5 unrecognised type
int	alloc_status	the status of the last attempted allocation or deallocation
char	bad_alloc[81]	the name of the array for which an allocation or deallocation error ocurred
int	io_status	status from last read attempt
int	line	number of last line read from i/o file
char	p_type[4]	problem type

3.1.2 Function Documentation

3.1.2.1 rpd_initialize()

```
void rpd_initialize (
             void ** data,
             struct rpd_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 rpd_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

rpdt.c, and rpdtf.c.

3.1.2.2 rpd_get_stats()

```
void rpd_get_stats (
             char qplib_file[],
             int qplib_file_len,
             struct rpd_control_type * control,
             void ** data,
             int * status,
             char p_type[4],
             int * n,
             int * m,
             int * h_ne,
             int * a_ne,
             int * h_c_ne )
```

Read the data from a specified QPLIB file into internal storage, and report the type of problem encoded, along with problem-specific dimensions.

Parameters

in	*qplib_file*	is a one-dimensional array of type char that specifies the name of the QPLIB file that is to be read.
in	*qplib_file_len*	is a scalar variable of type int, that gives the number of characters in the name encoded in qplib_file.
in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see rpd_control_type)
in,out	*data*	holds private internal data

Parameters

out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -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.

Parameters

out	*p_type*	is a one-dimensional array of size 4 and type char that specifies the type of quadratic programming problem encoded in the QPLIB file. The first character indicates the type of objective function used. It will be one of the following: • L a linear objective function. • D a convex quadratic objective function whose Hessian is a diagonal matrix. • C a convex quadratic objective function. • Q a quadratic objective function whose Hessian may be indefinite. The second character indicates the types of variables that are present. It will be one of the following: • C all the variables are continuous. • B all the variables are binary (0-1). • M the variables are a mix of continuous and binary. • I all the variables are integer. • G the variables are a mix of continuous, binary and integer. The third character indicates the type of the (most extreme) constraint function used; other constraints may be of a lesser type. It will be one of the following: • N there are no constraints. • B some of the variables lie between lower and upper bounds (box constraint). • L the constraint functions are linear. • D the constraint functions are convex quadratics with diagonal Hessians. • C the constraint functions are convex quadratics. • Q the constraint functions are quadratics whose Hessians may be indefinite. Thus for continuous problems, we would have • LCL a linear program. • LCC or LCQ a linear program with quadratic constraints. • CCB or QCB a bound-constrained quadratic program. • CCL or QCL a quadratic program. • CCC or CCQ or QCC or QCQ a quadratic program with quadratic constraints. For integer problems, the second character would be I rather than C, and for mixed integer problems, the second character would by M or G.
out	*n*	is a scalar variable of type int, that holds the number of variables.
out	*m*	is a scalar variable of type int, that holds the number of general constraints.

Parameters

out	*h_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H stored in the sparse symmetric co-ordinate storage scheme.
out	*a_ne*	is a scalar variable of type int, that holds the number of entries in A stored in the sparse co-ordinate storage scheme.
out	*h_c_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H_c stored in the joint sparse co-ordinate storage scheme.

Examples

rpdt.c, and rpdtf.c.

3.1.2.3 rpd_get_g()

```
void rpd_get_g (
             void ** data,
             int * status,
             int n,
             real_wp_ g[] )
```

Recover the linear term g from in objective function

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*n*	is a scalar variable of type int, that holds the number of variables.
out	*g*	is a one-dimensional array of size n and type double, that gives the linear term g of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.

Examples

rpdt.c, and rpdtf.c.

3.1.2.4 rpd_get_f()

```
void rpd_get_f (
             void ** data,
             int * status,
             real_wp_ * f )
```

Recover the constant term f in the objective function.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
out	*f*	is a scalar of type double, that gives the constant term f from the objective function.

Examples

rpdt.c, and rpdtf.c.

3.1.2.5 rpd_get_xlu()

```
void rpd_get_xlu (
             void ** data,
             int * status,
             int n,
             real_wp_ x_l[],
             real_wp_ x_u[] )
```

Recover the variable lower and upper bounds x_l and x_u.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*n*	is a scalar variable of type int, that holds the number of variables.
out	*x_l*	is a one-dimensional array of size n and type double, that gives the lower bounds x_l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains $(x_l)_j$.
out	*x_u*	is a one-dimensional array of size n and type double, that gives the upper bounds x_u on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains $(x_u)_j$.

Examples

rpdt.c, and rpdtf.c.

3.1.2.6 rpd_get_clu()

```
void rpd_get_clu (
             void ** data,
             int * status,
             int m,
             real_wp_ c_l[],
             real_wp_ c_u[] )
```

Recover the constraint lower and upper bounds c_l and c_u.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*m*	is a scalar variable of type int, that holds the number of general constraints.
out	*c_l*	is a one-dimensional array of size m and type double, that gives the lower bounds c_l on the constraints Ax. The i-th component of c_l, i = 0, ... , m-1, contains $(c_l)_i$.
out	*c_u*	is a one-dimensional array of size m and type double, that gives the upper bounds c_u on the constraints Ax. The i-th component of c_u, i = 0, ... , m-1, contains $(c_u)_i$.

Examples

rpdt.c, and rpdtf.c.

3.1.2.7 rpd_get_h()

```
void rpd_get_h (
             void ** data,
             int * status,
             int h_ne,
             int h_row[],
             int h_col[],
             real_wp_ h_val[] )
```

Recover the Hessian term H in the objective function.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.

Parameters

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.
out	*h_row*	is a one-dimensional array of size h_ne and type int, that gives the row indices of the lower triangular part of H in the sparse co-ordinate storage scheme.
out	*h_col*	is a one-dimensional array of size h_ne and type int, that gives the column indices of the lower triangular part of H in the sparse co-ordinate storage scheme.
out	*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 the sparse co-ordinate storage scheme.

Examples

rpdt.c, and rpdtf.c.

3.1.2.8 rpd_get_a()

```
void rpd_get_a (
             void ** data,
             int * status,
             int a_ne,
             int a_row[],
             int a_col[],
             real_wp_ a_val[] )
```

Recover the Jacobian term A in the constraints.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*a_ne*	is a scalar variable of type int, that holds the number of entries in the constraint Jacobian matrix A.
out	*a_row*	is a one-dimensional array of size a_ne and type int, that gives the row indices of A in the sparse co-ordinate storage scheme.
out	*a_col*	is a one-dimensional array of size a_ne and type int, that gives the column indices of A in the sparse co-ordinate, storage scheme.
out	*a_val*	is a one-dimensional array of size a_ne and type double, that gives the values of the entries of the constraint Jacobian matrix A in the sparse co-ordinate scheme.

Examples

rpdt.c, and rpdtf.c.

3.1.2.9 rpd_get_h_c()

```
void rpd_get_h_c (
            void ** data,
            int * status,
            int h_c_ne,
            int h_c_ptr[],
            int h_c_row[],
            int h_c_col[],
            real_wp_ h_c_val[] )
```

Recover the Hessian terms H_c in the constraints.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*h_c_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix H.
out	*h_c_ptr*	is a one-dimensional array of size h_c_ne and type int, that gives the constraint indices of the lower triangular part of H_c in the joint sparse co-ordinate storage scheme.
out	*h_c_row*	is a one-dimensional array of size h_c_ne and type int, that gives the row indices of the lower triangular part of H_c in the joint sparse co-ordinate storage scheme.
out	*h_c_col*	is a one-dimensional array of size h_c_ne and type int, that gives the column indices of the lower triangular part of H_c in the sparse co-ordinate storage scheme.
out	*h_c_val*	is a one-dimensional array of size h_c_ne and type double, that holds the values of the entries of the lower triangular part of the Hessian matrix H_c in the sparse co-ordinate storage scheme.

Examples

rpdt.c, and rpdtf.c.

3.1.2.10 rpd_get_x_type()

```
void rpd_get_x_type (
            void ** data,
            int * status,
            int n,
            int x_type[] )
```

Recover the types of the variables x.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*n*	is a scalar variable of type int, that holds the number of variables.
out	*x_type*	is a one-dimensional array of size n and type int, that specifies the type of each variable x. Specifically, for j = 0, ... , n-1, x(j) = • 0 variable x_j is continuous, • 1 variable x_j is integer, and • 2 variable x_j is binary (0,1)

Examples

rpdt.c, and rpdtf.c.

3.1.2.11 rpd_get_x()

```
void rpd_get_x (
             void ** data,
             int * status,
             int n,
             real_wp_ x[] )
```

Recover the initial values of the variables x.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*n*	is a scalar variable of type int, that holds the number of variables.
out	*x*	is a one-dimensional array of size n and type double, that gives the initial values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.

Examples

rpdt.c, and rpdtf.c.

3.1.2.12 rpd_get_y()

```
void rpd_get_y (
             void ** data,
             int * status,
             int m,
             real_wp_ y[] )
```

Recover the initial values of the Lagrange multipliers y.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*m*	is a scalar variable of type int, that holds the number of general constraints.
out	*y*	is a one-dimensional array of size n and type double, that gives the initial values y of the Lagrange multipliers for the general constraints. The j-th component of y, j = 0, ... , n-1, contains y_j.

Examples

rpdt.c, and rpdtf.c.

3.1.2.13 rpd_get_z()

```
void rpd_get_z (
             void ** data,
             int * status,
             int n,
             real_wp_ z[] )
```

Recover the initial values of the dual variables z.

Parameters

in,out	*data*	holds private internal data
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The statistics have been recovered succesfully. • -93. The QPLIB file did not contain the required data.
in	*n*	is a scalar variable of type int, that holds the number of variables.
out	*z*	is a one-dimensional array of size n and type double, that gives the initial values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.

Examples

rpdt.c, and rpdtf.c.

3.1.2.14 rpd_terminate()

```
void rpd_terminate (
            void ** data,
            struct rpd_control_type * control,
            struct rpd_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 rpd_control_type)
out	*inform*	is a struct containing output information (see rpd_inform_type)

Examples

rpdt.c, and rpdtf.c.

Chapter 4

Example Documentation

4.1 rpdt.c

This is an example of how to use the package to decode a QPLIB file.

Notice that C-style indexing is used, and that this is flaggeed by setting `control.f_indexing` to `false`.

```
/* rpdt.c */
/* Full test for the RPD C interface using C sparse matrix indexing */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "rpd.h"
#define BUFSIZE 1000
int main(void) {
    // Derived types
    void *data;
    struct rpd_control_type control;
    struct rpd_inform_type inform;
    char qplib_file[BUFSIZE];
    char *galahad = "GALAHAD";
    int qplib_file_len;
    // make sure the GALAHAD environment variable actually exists
    if(!getenv(galahad)){
      fprintf(stderr, " The environment variable %s was not found.\n", galahad);
      exit( 1 );
    }
    // make sure the buffer is large enough to hold the environment variable
    // value, and if so, copy it into qplib_file
    if( snprintf( qplib_file, BUFSIZE, "%s", getenv(galahad) ) >= BUFSIZE){
      fprintf (stderr, " BUFSIZE of %d was too small. Aborting\n", BUFSIZE );
      exit( 1 );
    }
    // extend the qplib_file string to include the actual position of the
    // provided ALLINIT.qplib example file provided as part GALAHAD
    char source[] = "/examples/ALLINIT.qplib";
    strcat( qplib_file, source );
    // compute the length of the string
    qplib_file_len = strlen( qplib_file );
    printf( " QPLIB file: %s\n", qplib_file );
    int status;
    int n;
    int m;
    int h_ne;
    int a_ne;
    int h_c_ne;
    char p_type[4];
    printf(" C sparse matrix indexing\n\n");
    printf(" basic tests of storage formats\n\n");
    // Initialize RPD */
    rpd_initialize( &data, &control, &status );
    // Set user-defined control options */
    control.f_indexing = false; // C sparse matrix indexing
    // Recover vital statistics from the QPLIB file
    rpd_get_stats( qplib_file, qplib_file_len, &control, &data, &status,
```

```
                  p_type, &n, &m, &h_ne, &a_ne, &h_c_ne );
    printf( " QPLIB file is of type %s\n", p_type );
    printf( " n = %i, m = %i, h_ne = %i, a_ne = %i, h_c_ne = %i\n",
            n, m, h_ne, a_ne, h_c_ne );
    // Recover g
    double g[n];
    rpd_get_g( &data, &status, n, g );
    printf( " g = %.1f %.1f %.1f %.1f %.1f\n",g[0], g[1], g[2], g[3], g[4]);
    // Recover f
    double f;
    rpd_get_f( &data, &status, &f );
    printf( " f = %.1f\n", f );
    // Recover xlu
    double x_l[n];
    double x_u[n];
    rpd_get_xlu( &data, &status, n, x_l, x_u );
    printf( " x_l = %.1f %.1f %.1f %.1f %.1f\n", x_l[0], x_l[1], x_l[2],
             x_l[3], x_l[4]);
    printf( " x_u = %.1f %.1f %.1f %.1f %.1f\n", x_u[0], x_u[1], x_u[2],
             x_u[3], x_u[4]);
    // Recover clu
    double c_l[m];
    double c_u[m];
    rpd_get_clu( &data, &status, m, c_l, c_u );
    printf( " c_l = %.1f %.1f\n", c_l[0], c_l[1] );
    printf( " c_u = %.1f %.1f\n", c_u[0], c_u[1] );
    // Recover H
    int h_row[h_ne];
    int h_col[h_ne];
    double h_val[h_ne];
    rpd_get_h( &data, &status, h_ne, h_row, h_col, h_val );
    printf( " h_row, h_col, h_val =\n");
    for( int i = 0; i < h_ne; i++) printf("   %i %i %.1f\n",
         h_row[i], h_col[i], h_val[i]);
    // Recover A
    int a_row[a_ne];
    int a_col[a_ne];
    double a_val[a_ne];
    rpd_get_a( &data, &status, a_ne, a_row, a_col, a_val );
    printf( " a_row, a_col, a_val =\n");
    for( int i = 0; i < a_ne; i++) printf("   %i %i %.1f\n",
         a_row[i], a_col[i], a_val[i]);
    // Recover H_c
    int h_c_ptr[h_c_ne];
    int h_c_row[h_c_ne];
    int h_c_col[h_c_ne];
    double h_c_val[h_c_ne];
    rpd_get_h_c( &data, &status, h_c_ne, h_c_ptr, h_c_row, h_c_col, h_c_val );
    printf( " h_c_row, h_c_col, h_c_val =\n");
    for( int i = 0; i < h_c_ne; i++) printf("   %i %i %i %.1f\n",
         h_c_ptr[i], h_c_row[i], h_c_col[i], h_c_val[i]);
    // Recover x_type
    int x_type[n];
    rpd_get_x_type( &data, &status, n, x_type );
    printf( " x_type = %i %i %i %i %i\n", x_type[0], x_type[1], x_type[2],
            x_type[3], x_type[4] );
    // Recover x
    double x[n];
    rpd_get_x( &data, &status, n, x );
    printf( " x = %.1f %.1f %.1f %.1f %.1f\n",x[0], x[1], x[2], x[3], x[4]);
    // Recover y
    double y[m];
    rpd_get_y( &data, &status, m, y );
    printf( " y = %.1f %.1f\n",y[0], y[1]);
    // Recover z
    double z[n];
    rpd_get_z( &data, &status, n, z );
    printf( " z = %.1f %.1f %.1f %.1f %.1f\n",z[0], z[1], z[2], z[3], z[4]);
    // Delete internal workspace
    rpd_terminate( &data, &control, &inform );
}
```

4.2 rpdtf.c

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

```
/* rpdtf.c */
/* Full test for the RPD C interface using Fortran sparse matrix indexing */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include <math.h>
#include "rpd.h"
#define BUFSIZE 1000
int main(void) {
    // Derived types
    void *data;
    struct rpd_control_type control;
    struct rpd_inform_type inform;
    char qplib_file[BUFSIZE];
    char *galahad = "GALAHAD";
    int qplib_file_len;
    // make sure the GALAHAD environment variable actually exists
    if(!getenv(galahad)){
      fprintf(stderr, " The environment variable %s was not found.\n", galahad);
      exit( 1 );
    }
    // make sure the buffer is large enough to hold the environment variable
    // value, and if so, copy it into qplib_file
    if( snprintf( qplib_file, BUFSIZE, "%s", getenv(galahad) ) >= BUFSIZE){
      fprintf (stderr, " BUFSIZE of %d was too small. Aborting\n", BUFSIZE );
      exit( 1 );
    }
    // extend the qplib_file string to include the actual position of the
    // provided ALLINIT.qplib example file provided as part GALAHAD
    char source[] = "/examples/ALLINIT.qplib";
    strcat( qplib_file, source );
    // compute the length of the string
    qplib_file_len = strlen( qplib_file );
    printf( " QPLIB file: %s\n", qplib_file );
    int status;
    int n;
    int m;
    int h_ne;
    int a_ne;
    int h_c_ne;
    char p_type[4];
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" basic tests of storage formats\n\n");
    // Initialize RPD */
    rpd_initialize( &data, &control, &status );
    // Set user-defined control options */
    control.f_indexing = true; // fortran sparse matrix indexing
    // Recover vital statistics from the QPLIB file
    rpd_get_stats( qplib_file, qplib_file_len, &control, &data, &status,
                   p_type, &n, &m, &h_ne, &a_ne, &h_c_ne );
    printf( " QPLIB file is of type %s\n", p_type );
    printf( " n = %i, m = %i, h_ne = %i, a_ne = %i, h_c_ne = %i\n",
            n, m, h_ne, a_ne, h_c_ne );
    // Recover g
    double g[n];
    rpd_get_g( &data, &status, n, g );
    printf( " g = %.1f %.1f %.1f %.1f %.1f\n",g[0], g[1], g[2], g[3], g[4]);
    // Recover f
    double f;
    rpd_get_f( &data, &status, &f );
    printf( " f = %.1f\n", f );
    // Recover xlu
    double x_l[n];
    double x_u[n];
    rpd_get_xlu( &data, &status, n, x_l, x_u );
    printf( " x_l = %.1f %.1f %.1f %.1f %.1f\n", x_l[0], x_l[1], x_l[2],
             x_l[3], x_l[4]);
    printf( " x_u = %.1f %.1f %.1f %.1f %.1f\n", x_u[0], x_u[1], x_u[2],
             x_u[3], x_u[4]);
    // Recover clu
    double c_l[m];
    double c_u[m];
    rpd_get_clu( &data, &status, m, c_l, c_u );
    printf( " c_l = %.1f %.1f\n", c_l[0], c_l[1] );
    printf( " c_u = %.1f %.1f\n", c_u[0], c_u[1] );
    // Recover H
    int h_row[h_ne];
    int h_col[h_ne];
    double h_val[h_ne];
    rpd_get_h( &data, &status, h_ne, h_row, h_col, h_val );
    printf( " h_row, h_col, h_val =\n");
    for( int i = 0; i < h_ne; i++) printf("   %i %i %.1f\n",
         h_row[i], h_col[i], h_val[i]);
    // Recover A
    int a_row[a_ne];
    int a_col[a_ne];
    double a_val[a_ne];
    rpd_get_a( &data, &status, a_ne, a_row, a_col, a_val );
    printf( " a_row, a_col, a_val =\n");
    for( int i = 0; i < a_ne; i++) printf("   %i %i %.1f\n",
         a_row[i], a_col[i], a_val[i]);
    // Recover H_c
```

```
    int h_c_ptr[h_c_ne];
    int h_c_row[h_c_ne];
    int h_c_col[h_c_ne];
    double h_c_val[h_c_ne];
    rpd_get_h_c( &data, &status, h_c_ne, h_c_ptr, h_c_row, h_c_col, h_c_val );
    printf( " h_c_row, h_c_col, h_c_val =\n");
    for( int i = 0; i < h_c_ne; i++) printf("   %i %i %i %.1f\n",
         h_c_ptr[i], h_c_row[i], h_c_col[i], h_c_val[i]);
    // Recover x_type
    int x_type[n];
    rpd_get_x_type( &data, &status, n, x_type );
    printf( " x_type = %i %i %i %i %i\n", x_type[0], x_type[1], x_type[2],
            x_type[3], x_type[4] );
    // Recover x
    double x[n];
    rpd_get_x( &data, &status, n, x );
    printf( " x = %.1f %.1f %.1f %.1f %.1f\n",x[0], x[1], x[2], x[3], x[4]);
    // Recover y
    double y[m];
    rpd_get_y( &data, &status, m, y );
    printf( " y = %.1f %.1f\n",y[0], y[1]);
    // Recover z
    double z[n];
    rpd_get_z( &data, &status, n, z );
    printf( " z = %.1f %.1f %.1f %.1f %.1f\n",z[0], z[1], z[2], z[3], z[4]);
    // Delete internal workspace
    rpd_terminate( &data, &control, &inform );
}
```

Index