C interfaces to GALAHAD LPA

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory

Thu Jan 13 2022

Chapter 1

GALAHAD C package lpa

1.1 Introduction

1.1.1 Purpose

This package uses the **simplex method** to solve the **linear programming problem**

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

subject to the general linear constraints

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

and the simple bound constraints

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

where the vectors g, w, x^0, 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 A whose rows are the transposes of the vectors a_i.

N.B. The package is simply a sophisticated interface to the HSL package LA04, and requires that a user has obtained the latter. **LA04 is not included in GALAHAD** but is available without charge to recognised academics, see `http://www.hsl.rl.ac.uk/catalogue/la04.html`. If LA04 is unavailable, the GALAHAD interior-point linear programming package LPB is an alternative.

1.1.2 Authors

N. I. M. Gould and J. K. Reid, STFC-Rutherford Appleton Laboratory, England.

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

1.1.3 Originally released

October 2018, C interface September 2021.

1.1.4 Terminology

The required solution x necessarily satisfies the primal optimality conditions

$$\text{(1a)} \qquad Ax = c$$

and

$$\text{(1b)} \qquad c^l \leq c \leq c^u, \;\; x^l \leq x \leq x^u,$$

the dual optimality conditions

$$\text{(2a)} \quad g = A^T y + z$$

where

$$\text{(2b)} \qquad y = y^l + y^u, \;\; z = z^l + z^u, \;\; y^l \geq 0, \;\; y^u \leq 0, \;\; z^l \geq 0 \text{ and } z^u \leq 0,$$

and the complementary slackness conditions

$$\text{(3)} \qquad (Ax - c^l)^T y^l = 0, \;\; (Ax - c^u)^T y^u = 0, \;\; (x - x^l)^T z^l = 0 \text{ and } (x - x^u)^T z^u = 0,$$

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.

The so-called dual to this problem is another linear program

$$-\text{minimize } c^{lT} y^l + c^{uT} y^u + x^{lT} z^l + x^{uT} z^u + f \text{ subject to the constraints (2a) and (2b)}$$

that uses the same data. The solution to the two problems, it is exists, is the same, but if one is infeasible, the other is unbounded. It can be more efficient to solve the dual, particularly if m is much larger than n.

1.1.5 Method

The bulk of the work is peformed by the HSL package LA04. The main subbroutine from this package requires that the input problem be transformed into the `‘standard form"

$$\begin{array}{rl} \text{minimize} & g'^T x' \\ (4) \;\; \text{subject to} & A'x' = b \\ & l_i \leq x'_i \leq u_i, \;\; (i \leq k) \\ \text{and} & x'_l \geq 0, \;\; (i \geq l) \end{array}$$

by introducing slack an surpulus variables, reordering and removing fixed variables and free constraints. The resulting problem involves n' unknowns and m' general constraints. In order to deal with the possibility that the general constraints are inconsistent or not of full rank, LA04 introduces additional `‘artifical" variables v and replaces the constraints of (4) by

$$(5) \;\; A'x' + v = b$$

and gradually encourages v to zero as a first solution phase.

Once a selection of m' independent **non-basic** variables is made, the constraints (5) determine the remaining m' dependent **basic** variables. The **simplex method** is a scheme for systematically adjusting the choice of basic and non-basic variables until a set which defines an optimal solution of (4) is obtained. Each iteration of the simplex method requires the solution of a number of sets of linear equations whose coefficient matrix is the **basis** matrix B, made up of the columns of $[A' \; I]$ corresponding to the basic variables, or its transpose B^T. As the basis matrices for consecutive iterations are closely related, it is normally advantageous to update (rather than recompute) their factorizations as the computation proceeds. If an initial basis is not provided by the user, a set of basic variables which provide a (permuted) triangular basis matrix is found by the simple crash algorithm of Gould and Reid (1989), and initial steepest-edge weights are calculated.

Phases one (finding a feasible solution) and two (solving (4) of the simplex method are applied, as appropriate, with the choice of entering variable as described by Goldfarb and Reid (1977) and the choice of leaving variable as proposed by Harris (1973). Refactorizations of the basis matrix are performed whenever doing so will reduce the average iteration time or there is insufficient memory for its factors. The reduced cost for the entering variable is computed afresh. If it is found to be of a different sign from the recurred value or more than 10% different in magnitude, a fresh computation of all the reduced costs is performed. Details of the factorization and updating procedures are given by Reid (1982). Iterative refinement is encouraged for the basic solution and for the reduced costs after each factorization of the basis matrix and when they are recomputed at the end of phase 1.

1.1.6 References

D. Goldfarb and J. K. Reid (1977). A practicable steepest-edge simplex algorithm. Mathematical Programming **12** 361-371.

N. I. M. Gould and J. K. Reid (1989) New crash procedures for large systems of linear constraints. Mathematical Programming **45** 475-501.

P. M. J. Harris (1973). Pivot selection methods of the Devex LP code. Mathematical Programming **5** 1-28.

J. K. Reid (1982) A sparsity-exploiting variant of the Bartels-Golub decomposition for linear-programming bases. Mathematical Programming **24** 55-69.

1.1.7 Call order

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

- lpa_initialize - provide default control parameters and set up initial data structures
- lpa_read_specfile (optional) - override control values by reading replacement values from a file
- lpa_import - set up problem data structures and fixed values
- lpa_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- lpa_solve_lp - solve the linear program
- lpa_information (optional) - recover information about the solution and solution process
- lpa_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), $\ldots$, 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.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 lpa.h File Reference

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

Data Structures

- struct lpa_control_type
- struct lpa_time_type
- struct lpa_inform_type

Functions

- void lpa_initialize (void ∗∗data, struct lpa_control_type ∗control, int ∗status)
- void lpa_read_specfile (struct lpa_control_type ∗control, const char specfile[ ])
- void lpa_import (struct lpa_control_type ∗control, void ∗∗data, int ∗status, int n, int m, const char A_type[ ], int A_ne, const int A_row[ ], const int A_col[ ], const int A_ptr[ ])
- void lpa_reset_control (struct lpa_control_type ∗control, void ∗∗data, int ∗status)
- void lpa_solve_lp (void ∗∗data, int ∗status, int n, int m, 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 lpa_information (void ∗∗data, struct lpa_inform_type ∗inform, int ∗status)
- void lpa_terminate (void ∗∗data, struct lpa_control_type ∗control, struct lpa_inform_type ∗inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct lpa_control_type

control derived type as a C struct

Examples

lpat.c, and lpatf.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 (>= 2 turns on LA)4 output)
int	start_print	any printing will start on this iteration
int	stop_print	any printing will stop on this iteration
int	maxit	at most maxit inner iterations are allowed
int	max_iterative_refinements	maximum number of iterative refinements allowed
int	min_real_factor_size	initial size for real array for the factors and other data
int	min_integer_factor_size	initial size for integer array for the factors and other data
int	random_number_seed	the initial seed used when generating random numbers
int	sif_file_device	specifies the unit number to write generated SIF file describing the current problem
int	qplib_file_device	specifies the unit number to write generated QPLIB file describing the current problem
real_wp_	infinity	any bound larger than infinity in modulus will be regarded as infinite
real_wp_	tol_data	the tolerable relative perturbation of the data (A,g,..) defining the problem
real_wp_	feas_tol	any constraint violated by less than feas_tol will be considered to be satisfied
real_wp_	relative_pivot_tolerance	pivot threshold used to control the selection of pivot elements in the matrix factorization. Any potential pivot which is less than the largest entry in its row times the threshold is excluded as a candidate
real_wp_	growth_limit	limit to control growth in the upated basis factors. A refactorization occurs if the growth exceeds this limit
real_wp_	zero_tolerance	any entry in the basis smaller than this is considered zero
real_wp_	change_tolerance	any solution component whose change is smaller than a tolerence times the largest change may be considered to be zero
real_wp_	identical_bounds_tol	any pair of constraint bounds (c_l,c_u) or (x_l,x_u) that are closer than identical_bounds_tol will be reset to the average of their values
real_wp_	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	scale	if .scale is true, the problem will be automatically scaled prior to solution. This may improve computation time and accuracy
bool	dual	should the dual problem be solved rather than the primal?
bool	warm_start	should a warm start using the data in C_stat and X_stat be attempted?
bool	steepest_edge	should steepest-edge weights be used to detetrmine the variable leaving the basis?
bool	space_critical	if .space_critical is true, every effort will be made to use as little space as possible. This may result in longer computation time
bool	deallocate_error_fatal	if .deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue
bool	generate_sif_file	if .generate_sif_file is .true. if a SIF file describing the current problem is to be generated
bool	generate_qplib_file	if .generate_qplib_file is .true. if a QPLIB file describing the current problem is to be generated
char	sif_file_name[31]	name of generated SIF file containing input problem
char	qplib_file_name[31]	name of generated QPLIB file containing input problem

Data Fields

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'

3.1.1.2 struct lpa_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_	clock_total	the total clock time spent in the package
real_wp_	clock_preprocess	the clock time spent preprocessing the problem

3.1.1.3 struct lpa_inform_type

inform derived type as a C struct

Examples

lpat.c, and lpatf.c.

Data Fields

int	status	return status. See LPA_solve for details
int	alloc_status	the status of the last attempted allocation/deallocation
char	bad_alloc[81]	the name of the array for which an allocation/deallocation error ocurred
int	iter	the total number of iterations required
int	la04_job	the final value of la04's job argument
int	la04_job_info	any extra information from an unsuccesfull call to LA04 (LA04's RINFO(35)
real_wp_	obj	the value of the objective function at the best estimate of the solution determined by LPA_solve
real_wp_	primal_infeasibility	the value of the primal infeasibility
bool	feasible	is the returned "solution" feasible?
real_wp_	RINFO[40]	the information array from LA04
struct lpa_time_type	time	timings (see above)

3.1.2 Function Documentation

3.1.2.1 lpa_initialize()

```
void lpa_initialize (
            void ** data,
            struct lpa_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 lpa_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

lpat.c, and lpatf.c.

3.1.2.2 lpa_read_specfile()

```
void lpa_read_specfile (
            struct lpa_control_type * control,
            const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters

Parameters

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

3.1.2.3 lpa_import()

```
void lpa_import (
            struct lpa_control_type * control,
            void ** data,
            int * status,
            int n,
            int m,
            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 lpa_control_type)
`in,out`	*data*	holds private internal data
`in,out`	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 or m $>$ 0 or requirement that A_type contains its relevant string 'dense', 'coordinate' or 'sparse_by_rows' has been violated.
`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`	*A_type*	is a one-dimensional array of type char that specifies the unsymmetric storage scheme used for the constraint Jacobian, A. It should be one of 'coordinate', 'sparse_by_rows' or 'dense; lower or upper case variants are allowed.
`in`	*A_ne*	is a scalar variable of type int, that holds the number of entries in A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
`in`	*A_row*	is a one-dimensional array of size A_ne and type int, that holds the row indices of A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
`in`	*A_col*	is a one-dimensional array of size A_ne and type int, that holds the column indices of A in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL.
`in`	*A_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of A, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.

Examples

lpat.c, and lpatf.c.

3.1.2.4 lpa_reset_control()

```
void lpa_reset_control (
            struct lpa_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 lpa_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 lpa_solve_lp()

```
void lpa_solve_lp (
            void ** data,
            int * status,
            int n,
            int m,
            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 linear program.

Parameters

in,out	*data*	holds private internal data

Parameters

`in,out`	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. Possible exit are: • 0. The run was succesful. • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 and m $>$ 0 or requirement that A_type contains its relevant string 'dense', 'coordinate' or 'sparse_by_rows' has been violated. • -5. The simple-bound constraints are inconsistent. • -7. The constraints appear to have no feasible point. • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -17. The step is too small to make further impact. • -18. Too many iterations have been performed. This may happen if control.maxit is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem.
`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`	*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.

Parameters

`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, i = 0, ... , m-1, contains $c_i(x)$.
`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, i = 0, ... , m-1, contains y_i.
`in,out`	*z*	is a one-dimensional array of size n and type double, that holds the values z of the dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
`out`	*x_stat*	is a one-dimensional array of size n and type int, that gives the optimal status of the problem variables. If x_stat(j) is negative, the variable x_j most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.
`out`	*c_stat*	is a one-dimensional array of size m and type int, that gives the optimal status of the general linear constraints. If c_stat(i) is negative, the constraint value $a_i^T x$ most likely lies on its lower bound, if it is positive, it lies on its upper bound, and if it is zero, it lies between its bounds.

Examples

lpat.c, and lpatf.c.

3.1.2.6 lpa_information()

```
void lpa_information (
            void ** data,
            struct lpa_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 lpa_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

lpat.c, and lpatf.c.

3.1.2.7 lpa_terminate()

```
void lpa_terminate (
            void ** data,
            struct lpa_control_type * control,
            struct lpa_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 lpa_control_type)
`out`	*inform*	is a struct containing output information (see lpa_inform_type)

Examples

lpat.c, and lpatf.c.

Chapter 4

Example Documentation

4.1 lpat.c

This is an example of how to use the package to solve a linear program. A variety of supported 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`.

```
/* lpat.c */
/* Full test for the LPA C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "lpa.h"
int main(void) {
    // Derived types
    void *data;
    struct lpa_control_type control;
    struct lpa_inform_type inform;
    // Set problem data
    int n = 3; // dimension
    int m = 2; // number of general constraints
    double g[] = {0.0, 2.0, 0.0};   // linear term in the objective
    double f = 1.0;  // constant term in the objective
    int A_ne = 4; // Jacobian elements
    int A_row[] = {0, 0, 1, 1}; // row indices
    int A_col[] = {0, 1, 1, 2}; // column indices
    int A_ptr[] = {0, 2, 4}; // row pointers
    double A_val[] = {2.0, 1.0, 1.0, 1.0 }; // values
    double c_l[] = {1.0, 2.0};   // constraint lower bound
    double c_u[] = {2.0, 2.0};   // constraint upper bound
    double x_l[] = {-1.0, - INFINITY, - INFINITY}; // variable lower bound
    double x_u[] = {1.0, INFINITY, 2.0}; // variable upper bound
    // Set output storage
    double c[m]; // constraint values
    int x_stat[n]; // variable status
    int c_stat[m]; // constraint status
    char st;
    int status;
    printf(" C sparse matrix indexing\n\n");
    printf(" basic tests of lp storage formats\n\n");
    for( int d=1; d <= 3; d++){
        // Initialize LPA
        lpa_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';
                lpa_import( &control, &data, &status, n, m,
                           "coordinate", A_ne, A_row, A_col, NULL );
                lpa_solve_lp( &data, &status, n, m, 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';
                lpa_import( &control, &data, &status, n, m,
                            "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                lpa_solve_lp( &data, &status, n, m, 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 A_dense_ne = 6; // number of elements of A
                double A_dense[] = {2.0, 1.0, 0.0, 0.0, 1.0, 1.0};
                lpa_import( &control, &data, &status, n, m,
                            "dense", A_ne, NULL, NULL, NULL );
                lpa_solve_lp( &data, &status, n, m, g, f,
                              A_dense_ne, A_dense, c_l, c_u, x_l, x_u,
                              x, c, y, z, x_stat, c_stat );
                break;
            }
        lpa_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: LPA_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
        lpa_terminate( &data, &control, &inform );
    }
}
```

4.2 lpatf.c

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

```
/* lpatf.c */
/* Full test for the LPA C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "lpa.h"
int main(void) {
    // Derived types
    void *data;
    struct lpa_control_type control;
    struct lpa_inform_type inform;
    // Set problem data
    int n = 3; // dimension
    int m = 2; // number of general constraints
    double g[] = {0.0, 2.0, 0.0};   // linear term in the objective
    double f = 1.0;  // constant term in the objective
    int A_ne = 4; // Jacobian elements
    int A_row[] = {1, 1, 2, 2}; // row indices
    int A_col[] = {1, 2, 2, 3}; // column indices
    int A_ptr[] = {1, 3, 5}; // row pointers
    double A_val[] = {2.0, 1.0, 1.0, 1.0 }; // values
    double c_l[] = {1.0, 2.0};   // constraint lower bound
    double c_u[] = {2.0, 2.0};   // constraint upper bound
    double x_l[] = {-1.0, - INFINITY, - INFINITY}; // variable lower bound
    double x_u[] = {1.0, INFINITY, 2.0}; // variable upper bound
    // Set output storage
    double c[m]; // constraint values
    int x_stat[n]; // variable status
    int c_stat[m]; // constraint status
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" basic tests of lp storage formats\n\n");
    for( int d=1; d <= 3; d++){
        // Initialize LPA
        lpa_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';
                lpa_import( &control, &data, &status, n, m,
                           "coordinate", A_ne, A_row, A_col, NULL );
                lpa_solve_lp( &data, &status, n, m, 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';
                lpa_import( &control, &data, &status, n, m,
                            "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                lpa_solve_lp( &data, &status, n, m, 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 A_dense_ne = 6; // number of elements of A
                double A_dense[] = {2.0, 1.0, 0.0, 0.0, 1.0, 1.0};
                lpa_import( &control, &data, &status, n, m,
                            "dense", A_ne, NULL, NULL, NULL );
                lpa_solve_lp( &data, &status, n, m, g, f,
                              A_dense_ne, A_dense, c_l, c_u, x_l, x_u,
                              x, c, y, z, x_stat, c_stat );
                break;
            }
        lpa_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: LPA_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
        lpa_terminate( &data, &control, &inform );
    }
}
```

Index