Working Paper

LP-DIT
Data Interchange Tool for Linear Programming Problems (version 1.20)

Marek Makowski

WP-94-36
June 1994

IIASA International Institute for Applied Systems Analysis □ A-2361 Laxenburg □ Austria
Telephone: +43 2236 71521 □ Telex: 079 137 iiasa a □ Telefax: +43 2236 71313

LP-DIT
Data Interchange Tool
for Linear Programming Problems
(version 1.20)

Marek Makowski

WP-94-36
June 1994

International Institute for Applied Systems Analysis □ A-2361 Laxenburg □ Austria
Telephone: +43 2236 71521 □ Telex: 079 137 iiasa a □ Telefax: +43 2236 71313

Foreword

Many model-based Decision Support Systems (DSS) require formulation, solution, analysis and modification of mathematical programming problems. Each of these activities use corresponding pieces of software and each of these software pieces uses internal (*private*) data structures that are different. Data structures are quite often different also for solvers that use a same method for solving a given type of a mathematical programming problem. Therefore, in order to couple those software, an efficient way of data interchange is needed that allows for efficient access to data without restricting actual implementation of the internal data structure. Hence, to find a commonly accepted way of data interchange is an important issue for practical applications of Operations Research tools developed by teams specialized in different fields.

In order to spread the scope of potential applications and to increase the ability to meet specific needs of users, in particular in various IIASA projects, there is a need to modularize the architecture of Decision Support Systems. A modular DSS consists of a collection of tools rather than one closed system, thus allowing the user to carry out various and problem-specific analyses. Modularity also eases software reusability, which is one of the key factors in any major software development project.

This Working Paper documents the LP-DIT, which is a prototype implementation of a tool for LP data interchange between modules (such as a problem generator, solver, software supporting interactive multicriteria analysis) that form a DSS. LP-DIT has already been applied to several applications at IIASA. LP-DIT is implemented in three LP solvers and one MIP solver which are available free of charge for non-commercial applications.

Abstract

The MPS format (and its various extensions) is the de facto standard for handling data of an LP problem. The MPS format is widely used despite of its well known disadvantages, simply because there is no other widely agreed way of handling LP problem data and solution. However, for any real-life application when a sequence of modified LP problems is solved, preparing data (and their modifications) using the MPS format is both inefficient and cumbersome. Therefore the need for an efficient alternative is widely recognized. It is hardly possible to propose an alternative that could be commonly accepted unless the alternative is efficient and easy to incorporate into existing software. Therefore instead of considering another format of data, one should rather agree on a set of data structures and functions that can be used in a way similar to usage of standard libraries. Restricting a specification to the data structures and functions makes it possible to hide internal data structures and implementations of functions.

LP-DIT is an attempt to contribute to the creation of such an alternative. LP-DIT serves two purposes: First, to propose a data structure and declarations of functions that can easily provide efficient data processing commonly needed for interchange of LP problem data between different software modules. Second, to provide a public domain tool (fairly easy to implement and efficient to use) which allows for LP problem data interchange. In other words, LP-DIT provides an alternative for using the MPS format to access and modification of LP problem data. Additionally, LP-DIT provides efficient and flexible functions for a full definition (which includes information contained both in the MPS format and in a specification files) of an LP problem, its modifications and solutions.

The current version of LP-DIT is the result of several applications made for different problems (i.e. using different problem generators and LP solvers). However, it is still a prototype and therefore, criticism and suggestions will be appreciated.

Contents

LP-DIT
Data Interchange Tool
for Linear Programming Problems
(version 1.20)

Marek Makowski

1 Introduction

Many model-based research and applications require formulation, solution, analysis and modification of mathematical programming problems. This is especially important for model-based Decision Support Systems (DSS), where a sequence of related problems is generated, solved and analyzed. One can distinguish the following groups of related modeling activities, underlying methodologies and software:

Problem generation: Generation of an initial (core) formulation of a mathematical programming problem. Usually generation of a real-life problem requires processing of large amounts of data and logical relations, and it results in a *core* formulation of an MP (Mathematical Programming) problem that serves as a base for the model analysis.

Problem solving: Solution of the resulting MP problem requires a robust and efficient solver. It is quite often desirable, especially for large scale problems, to have a possibility of trying different solvers. Also for testing solvers, different problems (or even instances of modified problems) are usually helpful.

Problem analysis and modification: Analysis of a solution and generation of another, usually closely related MP problem. Usually a series of modifications of the core model (e.g. by changing a goal function and/or selected constraints) serves for the model analysis. This software module is usually either well integrated with the problem generator, or it is a stand-alone tool for problem modification, or it is used for generation and interactive modification of a series of multi-criteria problems (cf e.g. [Mak94]).

In order to spread the scope of potential applications and to increase the ability to meet specific needs of users, there is a need to modularize the architecture of Decision Support Systems. A modular DSS consists of a collection of tools rather than one closed system, thus allowing both efficient problem-specific analyses and efficient development and maintenance of the needed software. Therefore, we can consider the modules of software that correspond to the above listed groups of research and modeling activities.

These modules are complex pieces of software (which also typically have modular structure) that are usually developed by different research teams. Each module processes large amounts of *"private"* data, but the amounts of data that have to be exchanged between modules are quite often also large. Internal (*private*) data structures are different,

quite often also for solvers that use a same method for solving a given type of a mathematical programming problem. In order to couple those modules, an efficient way of data interchange is needed for providing efficient access to data without restricting actual implementation of the internal data structure. Finding a commonly accepted way of data interchange is an important issue for practical applications of Operations Research tools developed by teams specialized in different fields.

In many situations, a reasonable approach to provide decision support is to use an integrated modeling environment (e.g. AMPL [FGK93], GAMS [BKM88]). Each integrated modeling environment uses a (usually proprietary or not documented) binary format for efficient handling of data. Therefore, the data interchange is not a problem for such environments as long as the tools provided by an environment are adequate for generation, solution, analysis and modification of the MP problem. However, requirements for data handling and user interface quite often make usage of existing general purpose modeling tools not practicable, especially for many complex, real-life applications when specific requirements (like access and processing of data, user interface) are difficult to be fulfilled by an integrated modeling environment.

For applications that do need a problem specific elements of software, the needed software modules are either customized or developed in order to support the above summarized functions. Many parts of these software modules could be developed and used more efficiently, if an efficient and commonly agreed way for data interchange would exist. Moreover, also a part of the functionality of integrated modeling environments could be used (e.g. for generating an initial formulation of the problem or for solving an MP problem), if these environments would provide a commonly accepted way of data interchange.

One of the attempts to stimulate activities in the direction of establishing a widely accepted way of data interchange was the proposal for a Data Interchange Tool for Mathematical Programming (cf [MaS93a]). Since applications of LP (Linear Programming) and MIP (Mixed Integer Linear Programming) problems constitute a substantial part of optimization problems, a pilot implementation of a data interchange tool LP-DIT has been made for LP problems.[1]

The MPS format (cf e.g. [Mur81]) is the de facto standard for handling data of an LP problem. The MPS format has been designed several decades ago to be used with batch oriented software for solving small[2] LP problems. The MPS format has later been modified by different producers of commercial software and extended for handling also MIP problems. The MPS format is still widely used despite of its well known disadvantages, simply because there is no other widely agreed way of handling LP problem data and solution. Nowadays, handling of data for LP problems in the MPS format may be considered rational for development and testing of LP solvers. However, for any real-life application when a sequence of modified LP problems is solved, preparing data (and their modifications) using the MPS format is both inefficient and cumbersome. Therefore the need for an efficient alternative is widely recognized.

It is hardly possible to propose an alternative that could be commonly accepted unless the alternative is efficient and easy to incorporate into existing software. Therefore, instead of considering another format of data, one should rather agree on a set of data structures and functions that can be used in a way similar to usage of standard libraries. Restricting a specification to the data structures and functions makes it possible to hide internal data structures and implementations of functions. Hence, once the data struc-

[1] For the sake of brevity, we will further on assume that the abbreviation LP also covers MIP problems.

[2] The size of the LP problems that were on the limits of computers available in 1950's is nowadays considered to be small.

tures and declarations of functions that handle the data are agreed upon, then different implementations of the LP-DIT can be made independently from applications that use a preferable implementation. Typically, a change in, or replacement of LP-DIT should not result in a need of modifications of the modules that uses LP-DIT; only recompilation should be required.

This Working Paper, and the corresponding software, serves two mutually related purposes:

- To propose data structures and declarations of functions that can easily provide efficient data processing commonly needed for interchange of LP problem data between different modules.
- To provide a pilot implementation of those data and function specifications in a form of a public domain tool (fairly easy to implement and efficient to use) which allows for LP problem data interchange between different software modules.

In other words, LP-DIT provides an alternative for using the MPS format to access to, and modification of an LP problem data. Additionally, LP-DIT provides efficient and flexible functions for a full definition (which includes information contained both in the MPS format [including commonly used extensions] and in a specification file) of an LP problem, its modifications and solutions.

The current version of LP-DIT is the result of several applications made for different problems (i.e. using different problem generators and LP solvers). However, it is still a prototype and therefore criticism and suggestions will be appreciated.

2 Assumptions

The assumptions adopted for LP-DIT result in covering functionality of the MPS de facto standard for input data and its commonly accepted extensions (like BV, INT, LI, UI, INITIAL). LP-DIT also provides handling of other data necessary for problem specification (usually placed in a specification file), problem modification and solution handling.

2.1 Functional assumption

LP-DIT has been designed and implemented for efficient interchange of data between:

- A problem generator that generates an LP problem using any way of problem specification and a data base.
- A preprocessor that uses a definition of an LP problem in order to generate a multicriteria problem.
- A preprocessor that allows for analysis of previously obtained solutions and generates yet another LP problem.
- A solver.
- Report writers and possibly other application programs that need access to a solution of the LP problem.

LP-DIT provides the following functionality:

1. Handling of input data needed for definition of an LP problem, including information specific to MIP problems, like binary and integer variables, different types of SOS (Special Ordered Sets). Existence of lower and upper bounds for both variables and constraints is provided explicitly.
2. Handling of data needed for specification of a task for a solver (usually in specs file). Handling of most commonly used data (like declaring minimimization or maximization,

size and status of the problem) is provided explicitly. Other specifications and options are handled as character strings.
3. Selective storing of and access to elements of a solution.
4. An easy way of a problem modification.

Two utility programs are also provided: dit2mps and mps2dit for conversion between the MPS format and the LP-DIT hidden data format (cf Section 4).

2.2 Implementation assumption

1. LP-DIT internal data structure and handling are hidden from the user. Instead, a specification of a set of functions and their parameters is provided. Parameters include data structures that are declared in a way suitable for handling formulation, modifications and solutions of an LP problem. Such approach allows for different implementations of LP-DIT without a necessity of modifications in applications that are using LP-DIT.
2. Supply of and access to data handled by LP-DIT is similar to that commonly used for handling the MPS input format, thus modifications of data handling in existing generators and solvers are easy. Only few simple functions shall be supplied by the user in order to convert data from structures used by an application into (or from) LP-DIT data structures. Those functions depend on a type of module (problem generator, solver, pre- or post-processor) but in a typical application, all functions for a module can be coded in about 100 lines.
3. LP-DIT is used by calling one or two (depending on a type of application) LP-DIT library functions.
4. LP-DIT can be used for different precisions applied to the data. Namely, indices of variables can be 2 or 4 bytes integers and floating point numbers can be 4, 8 or 10 bytes long. The default are 2 byte integers for indices and 4 byte floating point numbers. The default is a reasonable choice for a majority of practical problems, where the number of variables is not very large and the precision of data does not exceed 7 digits.
5. LP-DIT is currently coded in ANSI C, implemented under Unix and DOS (cf. Section 6 for details). Therefore it can be used with ANSI C, C++ or fortran code. Since enough positive experience has been gathered for linking C++ and fortran code, hence future versions of LP-DIT will be implemented in C++.
6. Memory management and user interface (messages) are implemented in such a way that it is easy to replace their functionality by customized functions used by an application which uses also LP-DIT.

Efficiency of LP-DIT can be illustrated by a comparison of the time needed for reading a problem formulation and of sizes of input data files, for MPS and LP-DIT format, respectively. The results of processing two problems from the Netlib test library (cf [Gay85]), namely, `wood1p` and `fit2d` are presented in Table 1, which contains reading (processing) times (in seconds) and sizes[3] of the respective input files (in bytes). Times of reading input data has been tested on a Sparc-2 Workstation using the Cplex code (cf [CPL93]) and the Simplex solver (cf [Swi94]). The two solvers were selected in order to compare the efficiency of LP-DIT also with a commercial implementation. The functions implemented for processing of MPS format input files by Simplex are not efficient, because this version of the code was aimed at testing the solution technique. Nevertheless, the results summarized in Table 1 clearly show, that even an ad-hoc implementation of

[3]Note that the sizes of LP-DIT files may differ depending on a structure alignment option selected for the used compiler.

	woodlp	fit2d
time of reading mps file by Cplex	7.2	14.2
time of reading mps file by Simplex	51.2	695.0
time of reading LP-DIT file	0.8	1.9
size of mps file	2182090	4698538
size of LP-DIT file	656329	1462293

Table 1: Reading times and sizes of MPS format and LP-DIT binary files.

LP-DIT can be, especially for large scale problems, far more efficient than a very good implementation of processing the MPS format file.

One should also note that the time of processing MPS input files is often comparable with the time required for solution of a problem. For example, solution times by the Cplex code for the above two examples were 8.7 and 70.9, respectively.

3 User's guide

3.1 General information

Out of several possible applications of LP-DIT (for problem generation, modification, solution or analysis), each have two groups of functions:

- LP-DIT functions that are just called by the user.
- Functions that should be provided by the user and linked with an application. These functions are called by LP-DIT. For these functions, dummy functions are provided in the LP-DIT library. However, a call to a dummy function will result in a fatal error (and information about which function should have been supplied by the user).

These two groups of functions will be discussed separately for each type of application.

3.2 Commonly used parameters and data structures

The following parameters are used by the LP-DIT library functions which are called by the user:

`int iinput` – should be equal to 1, if data are supplied to LP-DIT (e.g. generating LP problem, providing a solution), and should be set to 0, if data are to be provided by LP-DIT (e.g. getting LP formulation for a solver).

`int by_cols` – currently must be set to 1 (in future releases of LP-DIT also row-wise access to data will be provided).

`int deb_lev` – debug level, 0 for quiet operation, 1 provides very limited information about processing, 2 and 3 may result in vast amount of output for a large problem, but might be useful for debugging.

`int ind_off` – offset of indices, should be set to 0 for C and C++ codes and 1 for fortran code.

`char *fname` – name of a file used for storing data in LP-DIT format.

`void *user` – a data pointer which is used in all user supplied functions. Any data needed by user supplied functions (which are called by LP-DIT) should be placed in a single structure, and a pointer to the structure should be passed as `void *user`.

The following data types and lengths of string are `typedef`'d in `lp_head.h`:

`LP_IND` – an integer used for row and column indices. It is typically defined as `unsigned short`, but for large problems it should be redefined to `unsigned int` or `unsigned long` This redefinition is hardware and compiler dependant. Some solvers and preprocessors use negative index value as a mark. Also, Fortran compilers do not provide `unsigned` integer types. For such applications, `LP_IND` should be redefined accordingly. The current implementation of LP-DIT uses `short int` as `LP_IND`, because it is also being used with a solver coded in Fortran.

`LP_FLOAT` – a floating point type used for storing all floating point data that define an LP problem. Usually 4 bytes `float` corresponds well to the precision of data available for most applications and therefore it is used as default by LP-DIT. For applications that need higher floating point data precision, `LP_FLOAT` can be redefined as `double` or `long double`.

`LP_NAME_LEN` – length of names used for columns and rows. Currently defined to be 8.

`LP_STR_LEN` – maximum length of a string used for specification lines, a problem name and comments attached to a solution. Currently defined to be 80.

The following structures (all are `typedef`'d in the file `lp_head.h`) are used by the user-provided functions. Consult the respective sections for more information:

`struct LP_HEAD` – Sec. 3.3.2
`struct LP_VAR` – Sec. 3.3.2
`struct LP_MAT` – Sec. 3.3.2
`struct LP_SOLUTION` – Sec. 3.4.2
`struct LP_MOD_VAR` – Sec. 3.5.2
`struct LP_MOD_ELEM` – Sec. 3.5.2

3.3 Problem definition

This group of functions provides input of a problem (generated by a problem generator) to LP-DIT and output of the problem definition for a solver. A definition of the LP problem contains information provided by the MPS format file (with extensions for MIP problems) and in a specification file.

All user supplied functions (or one of their `#include`'d files) should have a statement:

```
#include "lp_dit.h"
```

3.3.1 Library functions

The following two functions are to be called by an application (cf Sec. 3.2 for information about parameters).

`void lp_init(int iinput, int by_cols, int deb_lev, int ind_off, char *fname, void *user)` – initializes LP-DIT and calls a user supplied function either `lpi_par()` for `iinput = 1` or `lpo_par()` for `iinput = 0` (cf below for descriptions). Therefore, after return from `lp_init()` an application has enough information for dynamic allocation of memory for the problem being processed.

`void lp_def(int iinput, int deb_lev, void *user)` – allocates memory for a working area and actually handles LP problem definition. `lp_def()` calls other (than `lpi_par()` and `lpo_par()`) functions either specified in Sec. 3.3.3 for `iinput = 1` or

described in Sec. 3.3.4 for `iinput = 0`. Before returning `lp_def()` frees all memory allocated for working area. `lp_def()` does not return (calls `faterr()` function instead), if errors are detected during processing of the problem.

3.3.2 Data structures

The implemented data structures correspond to one of common ways of processing LP data available from an MPS format data file. Therefore there are three groups of data that define an LP problem:

- Problem specification that is usually given in a specification file. Solvers often allow dozens of optional parameters. Therefore LP-DIT handles only the most commonly used specifications in the `LP_HEAD` structure. Additionally, LP-DIT provides two functions, `lpi_specs()` and `lpo_specs()`, for handling any specifications which are stored in form of vector of strings (cf Sections 3.3.3 and 3.3.4, respectively, for details).
- Data for each row and column (except of matrix elements). LP-DIT treats rows and columns in a similar way. For each row and column name, number of non-zero elements, lower and upper bounds are provided. Additionally, information about a type of a row or a column is also provided. It includes not only traditional types provided by the MPS format, but also types used for MIP problems and additional solver specific attributes (that can be used for specific types of LP problems, like dynamic, stochastic, stair-case).
- Non-zero elements of the matrix are stored in a sequences of columns, each column containing number of non-zero elements and a vector of pairs composed of a row index and a corresponding value. Therefore the time consuming processing of column and row names[4] is avoided.

The following data structures are used for handling the above listed data:

`LP_HEAD` – contains basic information about the LP problem and has the following members that have to be set (in `lpi_par()`) by a user or that can be used by an application that gets problem definition from LP-DIT (by `lpo_par()`):
`char name[LP_STR_LEN + 1 ]` – problem name
`short min_max` – should be set to one of the following `#define`'d in `lp_head.h` values: `MINIMIZE`, `MAXIMIZE`.
`LP_IND m` – number of rows,
`LP_IND obj` – index of a goal function row,
`LP_IND n` – number of columns,
`LP_IND nint` – number of integer variables,
`long nz` – number of non-zero elements,
`short specs` – number of specifications lines.

Additionally, the following members can be used by an application:
`short status` – status of the problem (cf the predefined values below),
`double feas` – feasibility tolerance,
`double optim` – optimality tolerance,
`double infty` – a large number (used for "infinite" bounds), set by LP-DIT to `INFTY` (which is `#define`'d in `lp_head.h`).

`LP_VAR` – contains data for an LP variable (a column or a row) and has the following members:

[4] Names of rows and columns are in fact not needed for a solver. However, the names are available for solvers that use them (e.g. for diagnostic purposes).

`char name[LP_NAME_LEN + 1 ]` – a row/column name,
`LP_IND elems` – number of elements in a row/column,
`LP_FLOAT low_bnd` – lower bound value (`-INFTY`, if none),
`LP_FLOAT upp_bnd` – upper bound value (`INFTY`, if none),
`unsigned int is_eq:1` – is EQ row or FX col,
`unsigned int is_le:1` – is LE row or MI col,
`unsigned int is_ge:1` – is GE row or PL col,[5]
`unsigned int is_ne:1` – is N row or FR col,
`unsigned int is_low_bnd:1` – has row/column a finite lower bound,
`unsigned int is_upp_bnd:1` – has row/column a finite upper bound,
`unsigned int mip_type:2` – info for MIP problems,
`char attr` – place holder for a solver specific attributes.

Members which names start with `is_` are of `boolean` type and therefore should have values `0` or `1`. The `attr` member is a place-holder and can be used for any solver specific information about rows and columns (e.g. for MIP, dynamic, stochastic and stair-case types of problems). Most typical information for MIP solvers are stored in the `mip_type` member contains information specific for MIP problems. The following values are defined in `lp_head.h` and are used for MIP problems by the MOMIP solver (cf [OgZ94]) and by the `mps2dit` utility (cf Sec. 4) as the `mip_type` member:

```
/* Values used in MOMIP as mip_type (must be [0,3])*/
#define MIP_CV   ((char) 0)  /* continuous variable / non-SOS row */
#define MIP_SOS1 ((char) 1)  /* SOS1 row */
#define MIP_SOS2 ((char) 2)  /* SOS2 row */
#define MIP_INT  ((char) 1)  /* integer variable */
#define MIP_BV   ((char) 2)  /* binary variable */
```

`LP_MAT` – contains information about one vector of the LP matrix and has the following members:
`LP_IND index` – index of a column/row (for column/row wise handling of the matrix, respectively),
`LP_IND elems` – number of elements,
`LP_VECT *el` – vector of elements.

`LP_VECT` – is a vector composed of two element structures that have the following members:
`LP_IND index` – index of a column/row (for row/column wise handling of the matrix, respectively),
`LP_FLOAT value` – value of an element.

`LP_IND` and `LP_FLOAT` (defined in `lp_head.h`) correspond to a precision of variable indices and floating point numbers. In the current implementations of LP-DIT, these types are `short int` and `float`, respectively.

The following status codes used by LP-DIT are defined in `lp_head.h`:

```
/* Codes of the problem status */
#define LP_UNDEF  ( (short) 0)       /* Problem undefined */
```

[5] Following the commonly accepted practice, the `PL` identifier is used also for columns that have both lower and upper bounds.

```
#define LP_ERROR  ( (short) 1)      /* Error(s) in problem definition */
#define LP_INI    ( (short) 2)      /* Initial formulation of LP */
#define LP_MOD    ( (short) 3)      /* Modification of LP */
#define T_INI     ( (short) 4)      /* reserved for task */
#define T_MOD     ( (short) 5)      /* reserved for task */
#define S_UNB     ( (short) 6)      /* solution : problem unbounded */
#define S_LP_INF  ( (short) 7)      /* solution : LP infeasible */
#define S_IP_INF  ( (short) 8)      /* solver : IP_INFEASIBLE */
#define S_IP_QINF ( (short) 9)      /* solver : IP_?_INFEASIBLE */
#define S_OPT     ( (short) 10)     /* solver : optimal solution */
#define S_SOPT    ( (short) 11)     /* solver : sub-optimal solution */
#define S_QOPT    ( (short) 12)     /* solver : ?IP_OPTIMAL */
#define S_UOPT    ( (short) 13)     /* solver : ?IP_UNPROVEN */
#define S_UNF     ( (short) 14)     /* solver : problem not solved */
```

The LP-DIT function `char *statmsg(int status)` returns a string which corresponds to one of the above defined statuses.

3.3.3 User input functions

The following user supplied functions are to be provided (i.e. linked) for an application that generates an LP problem (these functions are called by LP-DIT if `lp_init()` is called with `iinput = 1`):

`void lpi_par(LP_HEAD *h, void *user)` – should supply `LP_HEAD` structure members listed in Sec. 3.3.2,

`void lpi_specs(LP_HEAD h, char **s, void *user)` – is called only, if `specs` member of `LP_HEAD` has a positive value. The function should copy into `s` strings corresponding to specification lines that are to be passed to a solver. The function should make sure that strings are not longer than a defined value `LP_STR_LEN` (80 characters in the current implementation).

`void lpi_rows(LP_HEAD h, LP_VAR *r, void *user)` – is called once for getting data for definition of all rows. The function should load all members of the `LP_VAR` structure for each row.

`void lpi_cols(LP_HEAD h, LP_VAR *c, void *user)` – is called once for getting data for definition of all columns. The function should load all members of the `LP_VAR` structure for each column.

`void lpi_vect(LP_HEAD h, LP_COL *v, void *user)` – is called for each column (for column-wise matrix handling) or for each row (for row-wise matrix handling).[6] It stores non-zero coefficients of a column or of a row.

The following defaults values are set for members of `LP_VAR` before calling `lpi_rows()` and `lpi_cols()`:
`name[0]` – is set to `'\0'`
`is_eq` – is set to 0
`is_le` – is set to 0

[6] Row-wise matrix handling is not implemented yet.

`is_ge` – is set to 0
`is_ne` – is set to 0
`is_low_bnd` – is set to 0 for rows and to 1 for columns
`is_upp_bnd` – is set to 0
`mip_type` – is set to MIP_CV
`attr` – is set to 0
`low_bnd` – is set to `-INFTY` for rows and to 0. for columns
`upp_bnd` – is set to `INFTY`
An application that generates an LP problem has to set appropriate types of rows and columns (by setting `1` to an appropriate `is_` member of `LP_VAR`) and the corresponding values of lower and upper bounds. Inconsistency of such settings results in a fatal error generated by LP-DIT. Such an approach prevents sending a not fully defined LP problem to a solver.

3.3.4 User output functions

The following user supplied functions are to be provided (i.e. linked) for a solver (these functions are called by LP-DIT if `lp_init()` is called with `iinput = 0`):

`void lpo_par(LP_HEAD h, void *user)` – provides values of the `LP_HEAD` structure members (cf Sec. 3.3.2),

`void lpo_specs(LP_HEAD h, char **s, void *user)` – is called only, if `specs` member of `LP_HEAD` has a positive value. The function provides in `s` strings corresponding to specification lines.

`void lpo_cols(LP_HEAD h, LP_VAR *c, void *user)` – is called once for providing data for all rows (contained in the `LP_VAR` structure for each row).

`void lpo_rows(LP_HEAD h, LP_VAR *r, void *user)` – is called once for providing data for all columns (contained in the `LP_VAR` structure for each column).

`void lpo_vect(LP_HEAD h, LP_COL *v, void *user)` – is called for each column (for column-wise matrix handling) or for each row (for row-wise matrix handling). It provides non-zero coefficients of a column or of a row.

3.3.5 Handling other data

Quite often a problem specific data that are not handled by the MPS standard can be used by an LP solver, pre- or postprocessor. Examples of such data include:
- Starting point.
- Initial or optimal basis.
- Data defining piece-wise linear function.
- A reference point used by the regularization technique.
- Data defining linear-quadratic problems.

Data structures for such data are usually specific for an implementation and a list of different types of data might be quite long. Therefore LP-DIT contains just one `lp_usr()` function, which is general enough for handling such data.

The function has the following declaration:
`int lp_usr(const char *fname, int iinput, const char *id, void *data, long size)` and the meaning of the parameters is as follows:

`fname` – name of a file used for storing data. One can use the same file as used for LP-DIT other data, if `lp_usr()` is called for storing the data after `lp_def()`.

`iinput` – should be set to `1`, if data is to be stored, and to `0` for data retrieval.

`id` – a string (max of 10 char length) identifying the data. The id should be unique (for all data items stored in one file). In order to avoid conflicts with id's used by LP-DIT, the id used by an application should not start with `lp_`, if same file used for other LP-DIT functions.

`data` – pointer to the data. For the data retrieval an appropriate amount of memory must be allocated by the calling application. Note, that LP-DIT has no way to check, if enough space is allocated for the `data`. Allocating not enough memory will result in a bug, usually a difficult one to be traced.

`size` – length of data (in bytes).

A non-zero return value indicates successful reading or writing of data.

3.4 Solution

This group of functions provides selective storing of, and access to a solution.

All user supplied functions should have a statement:

```
#include "lp_dit.h"
```

3.4.1 Library function

Only one function is to be called by an application (cf Sec. 3.2 for information about parameters):

`void lp_res(iinput, deb_lev, fname, user)` – which initializes LP-DIT and calls a user supplied function: either `lpi_res()` for `iinput = 1` or `lpo_res()` for `iinput = 0` (cf below for descriptions).

3.4.2 Data structures

`LP_SOLUTION` – contains a solution and has the following members:

- `short status` – status of the problem (cf Sec. 3.3.2 for details about the predefined values of the `status` member),
- `char comment[LP_STR_LEN + 1 ]` – any comment supplied by a solver (initialized by LP-DIT to `(none)` before calling `lpi_res()`,
- `char date[30 ]` – date (initialized by LP-DIT to a current date before calling `lpi_res()`),
- `LP_IND m` – number of rows,
- `LP_IND n` – number of columns,
- `double objv` – objective value,
- `LP_FLOAT time1` – execution time1,
- `LP_FLOAT time2` – execution time2,
- `LP_SOL cols` – values of variables (columns),
- `LP_SOL cols_d` – values of dual variables for bounds,
- `LP_SOL rows` – values of rows,
- `LP_SOL rows_d` – values of dual variables (shadow prices) for rows.

`LP_SOL` – contains a part of a solution and has the following members:
`LP_IND elems` – number of elements (columns or rows) for which the part a solution has been supplied by a solver,
`LP_VECT *el` – elements.

Consult Sec. 3.3.2 for declarations of `LP_IND`, `LP_FLOAT`, `LP_VECT`.

Splitting solution into parts is done due to the two observations. First, typically only a small fraction of solution information is used for analysis. Second, some solvers do not provide reliable values for all components of every part of a solution.

3.4.3 User input function

The following user supplied function has to be provided (i.e. linked) for a solver. This function is called by LP-DIT if `lp_res()` is called with `iinput = 1`:

`void lpi_res(LP_SOLUTION *h, void *user)` – should set all (possibly with exception of `comment` and `date`) members of the structure `LP_SOLUTION`.

3.4.4 User output functions

The following user supplied function has to be provided (i.e. linked) for an application that needs access to a solution. This function is called by LP-DIT if `lp_res()` is called with `iinput = 0`:

`void lpo_res(LP_SOLUTION *h, void *user)` – should load to an application data structures the solution provided in the structure `LP_SOLUTION`.

3.5 Problem modification

This type of application is aimed at modification of a problem previously stored in LP-DIT format. The following modifications functionality is provided:
- Adding and removing rows and columns.
- Changing a status of a row or a column. This means modifications of lower/upper bounds and/or of a type (integer, binary, continuous, SOS type) of a row or a column.
- Adding, removing, modifying matrix elements.

After the modification is completed the LP-DIT packs the problem, i.e. it removes empty rows and columns.

All user supplied functions should have a statement:
`#include "lp_mod.h"`
(which contains `#include "lp_dit.h"`).

3.5.1 Library functions

Consult the Section 3.2 for information about parameters and Sec. 3.5.2 about data structures used by these functions.

The following functions are to be called by an application:

`void lpm_load(int iinput, int by_cols, int deb_lev, int ind_off, char *fname)` – initializes LP-DIT and allocates memory necessary for a problem modification. This function must be called as a first LP-DIT function for a problem modification.

`void lpm_var(int num, LP_MOD_VAR *v)` – handles modifications of variables (rows and colums); `num` is a number of items to be modified, `v` contains modification data (cf Sec. 3.5.2 for the description of `LP_MOD_VAR`).

`void lpm_mat(int num, LP_MOD_ELEM *e)` – handles modifications of matrix elements; `num` is a number of items to be modified, `v` contains modification data (cf Sec. 3.5.2 for the description of `LP_COL`).

`void lpm_save(int deb_lev, char *fname)` – packs and stores the problem and frees all working memory. This functions should be called as the last one during modification.

The following functions are useful for getting information about elements of the modified problem:

`LP_IND lpm_indx(int var, char *name)` – return an index of a variable with a given `name`. The `var` parameter should have one of the values: `LP_ROW, LP_COL`.

`LP_VAR *lpm_gv(int var, LP_IND index)` – return current data of a variable (row or column) with a given `index`. The `var` parameter should have one of the values: `LP_ROW, LP_COL`. The returned pointer points to a static `LP_VAR` structure whose content will be overwritten by a subsequent call to `lpm_gv()`.

`void lpm_bnd(LP_VAR *var, LP_FLOAT low, LP_FLOAT upp)` – sets the members of the `LP_VAR` structure related to bounds (`low_bnd, upp_bnd, is_eq, is_le, is_ge, is_ne, is_low_bnd, is_upp_bnd`). The `define`'d value `INFTY` should be used, if a corresponding bound(s) does/do not exist (`-INFTY` and `INFTY` for `low` and `upp`, respectively).

3.5.2 Data structures

`LP_MOD_VAR` – contains information about a variable (row or column) that is modified, added or removed. The structure has the following members:

- `int var` – variable type, should be either `LP_ROW` or `LP_COL`, which corresponds to a modification of a row or a column variable, respectively.
- `int mod` – modification type, should be either `LP_ADD`, `LP_MOD` or `LP_RM`,
- `LP_VAR v` – updated data for a row or a column (cf Sec. 3.3.2 for declaration of the `LP_VAR` structure).

`LP_MOD_ELEM` – contains information about modified matrix elements and has the following members:

- `LP_IND irow` – index of a row,
- `LP_IND icol` – index of a column,
- `LP_FLOAT value` – value of a matrix coefficient.

The same data structure is being used for adding, modifying and removing a matrix elements (for the latter case the `value` should be set to 0.).

Consult Sec. 3.5.1 for description of functions that are handy for programming a problem modifications.

3.5.3 User supplied functions

None.

3.6 User customizable functions

The following LP-DIT functions can be replaced by an application specific functions in order to make two groups functions (handling messages and memory management) used by LP-DIT consistent with the respective functions used by an application:

`void *xalloc(void *addr, size_t size, const char *id)` – allocates (if `addr == 0`) or reallocates memory pointed to by `addr`. The amount of the newly allocated memory is `size` bytes. The function must not return, if enough memory can not be provided. In such a case, a fatal error should be generated (`id` is a identificator that gives information about the memory request that can not be fulfilled). The function returns the address of the allocated memory.

`void *xfree(void *addr, const char *file, int line)` – frees the memory pointed to by `addr` that was allocated by `xalloc`. The function should check if the address is non-zero. `file` and `line` arguments should be used in a fatal error message should the `addr` be equal to `0`. The function should return `0` (which is assigned to a respective address thus preventing double-freeing of memory).

`void outmsg(int code, const char *txt)` – output message pointed to by `txt`. The `code` parameter is used for different types of messages and can be disregarded by a customized function.

4 Utilities

Two programs are distributed together with the LP-DIT library:
`dit2mps` – - for conversion of LP-DIT data into MPS format
`mps2dit` – - for conversion of data in MPS format into LP-DIT format.
Each program called with `-h` argument provides a short description of its usage.

5 Availability of software

A beta version of the LP-DIT is available upon request by anonymous ftp. Currently SunOS and Solaris versions (compiled with Gnu C ver. 2.5.8 and linkable with both Gnu C++ and SunPro++ ver. 4.0 compilers) are available. A version for MS-DOS is available for Borland C++ ver. 4.0 compiler.

The distributed versions of the software will include also a Postscript file with the updated version of this Working Paper, which will continue to serve as a documentation of the software. Users are kindly requested to print this file and to make sure, that the version of the documentation corresponds to the version of the software.

LP-DIT is available free of charge for non-commercial research and educational purposes. Please contact the author (by e-mail: `marek@iiasa.ac.at`) for more information.

6 Conclusion

So far LP-DIT has been implemented within the following applications:

- Hybrid solver (cf [MaS93b]) applied to the RAINS model (cf [ASH90]). This application is a result of cooperation of TAP and MDA Projects.
- Problem generator and HOPDM LP solver applied for the problem of the land resources assessment (cf [GoM94]). This application is a result of cooperation of FAP and MDA Projects.
- Problem generator (cf [BMW93]) and MOMIP solver (cf [OgZ94]) applied for the problem of the regional water quality management problem. This application is a result of cooperation of WAT and MDA Projects.

Other applications are planned in the near future. This also includes implementation of the LP-DIT to a presolver which performs analysis and reduction of an LP problem.

Acknowledgments

The author would like to thank Martin W.P. Savelsbergh for his collaboration in the initial stage of the reported activity which has resulted in the proposal for a Data Interchange Tool for Mathematical Programming formulated in [MaS93a]. Thanks are also extended to Włodek Ogryczak for many fruitful discussions on the design and implementation of LP-DIT.

However, the author assumes full responsibility for any errors and faulty assumptions that might remain in the design and implementation of LP-DIT.

References

[ASH90] J. Alcamo, R. Shaw and L. Hordijk, eds., *The RAINS Model of Acidification*, Kluver Academic Publishers, Dordrecht, Boston, London, 1990.

[BKM88] A. Brooke, D. Kendrick and A. Meeraus, *GAMS, A User's Guide*, The Scientific Press, Redwood City, 1988.

[BMW93] R. Berkemer, M. Makowski and D. Watkins, *A prototype of a decision support system for river basin water quality management in Central and Eastern Europe*, Working Paper WP-93-49, International Institute for Applied Systems Analysis, Laxenburg, Austria, 1993.

[CPL93] CPLEX Optimization, Incline Village, *Using the CPLEX Callable Library and CPLEX Mixed Integer Library*, 1993.

[FGK93] R. Fourer, D. Gay and B. Kernighan, *AMPL, A Modeling Language for Mathematical Programming*, The Scientific Press, San Francisco, 1993.

[Gay85] D. Gay, *Electronic mail distribution of linear programming test problems*, Mathematical Programming Society COAL Newsletter (1985).

[GoM94] J. Gondzio and M. Makowski, *Solving a class of LP problems with primal-dual logarithmic barrier method*, European Journal of Operational Research (1994). (accepted for publication in 1993).

[Mak94] M. Makowski, *LP-MULTI, Modular tool for multiple criteria problems*, Working Paper WP-94-xx, International Institute for Applied Systems Analysis, Laxenburg, Austria, 1994. (To be published).

[MaS93a] M. Makowski and M. Savelsbergh, *MP-DIT Mathematical Programming Data Interchange Tool*, Mathematical Programming Society COAL Bulletin no. 22 (1993) 7–18.

[MaS93b] M. Makowski and J. Sosnowski, HYBRID*: Multicriteria linear programming system for computers under DOS and Unix*, in User-Oriented Methodology and Techniques of Decision Analysis and Support, J. Wessels and A. Wierzbicki, eds., Lecture Notes in Economics and Mathematical Systems, vol. 397, Springer Verlag, Berlin, New York, 1993, pp. 223–233.

[Mur81] B. Murtagh, *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York, 1981.

[OgZ94] W. Ogryczak and K. Zorychta, *Modular optimizer for mixed integer programming,* MOMIP *version 2.1*, Working Paper WP-94-35, International Institute for Applied Systems Analysis, Laxenburg, Austria, 1994.

[Swi94] A. Swietanowski, *SIMPLEX ver. 2.17: an implementation of the simplex algorithm for large scale linear problems – user's guide*, Working Paper WP-94-37, International Institute for Applied Systems Analysis, Laxenburg, Austria, 1994.