GNU Linear Programming Kit

Reference Manual

Version 4.8

(Draft Edition, January 2005)

The GLPK package is part of the GNU Project released under the aegis of GNU.

Free Software Foundation, Inc., 59 Temple Place — Suite 330, Boston, MA 02111, USA.

Contents

Chapter 1

Introduction

GLPK (GNU Linear Programming Kit) is a set of routines written in the ANSI C programming language and organized in the form of a callable library. It is intended for solving linear programming (LP), mixed integer programming (MIP), and other related problems.

1.1 LP Problem

GLPK assumes the following formulation of *linear programming (LP)* problem:

minimize (or maximize)

$$Z = c_1x_{m+1} + c_2x_{m+2} + \ldots + c_nx_{m+n} + c_0 \tag{1.1}$$

subject to linear constraints

$$\begin{array}{r@{\:}c@{\:}l} x_1 & = & a_{11}x_{m+1} + a_{12}x_{m+2} + \ldots + a_{1n}x_{m+n} \\ x_2 & = & a_{21}x_{m+1} + a_{22}x_{m+2} + \ldots + a_{2n}x_{m+n} \\ & \multicolumn{2}{c}{\ldots\ldots\ldots\ldots\ldots\ldots} \\ x_m & = & a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} \end{array} \tag{1.2}$$

and bounds of variables

$$\begin{array}{r@{\:}c@{\:}c@{\:}c@{\:}l} l_1 & \leq & x_1 & \leq & u_1 \\ l_2 & \leq & x_2 & \leq & u_2 \\ & & \ldots\ldots\ldots & & \\ l_{m+n} & \leq & x_{m+n} & \leq & u_{m+n} \end{array} \tag{1.3}$$

where: $x_1, x_2, \ldots, x_m$ — auxiliary variables; $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ — structural variables; Z — objective function; $c_1, c_2, \ldots, c_n$ — objective coefficients; c_0 — constant term ("shift") of the objective function; $a_{11}, a_{12}, \ldots, a_{mn}$ — constraint coefficients; $l_1, l_2, \ldots, l_{m+n}$ — lower bounds of variables; $u_1, u_2, \ldots, u_{m+n}$ — upper bounds of variables.

Auxiliary variables are also called *rows*, because they correspond to rows of the constraint matrix (i.e. a matrix built of the constraint coefficients). Analogously, structural variables are also called *columns*, because they correspond to columns of the constraint matrix.

Bounds of variables can be finite as well as infinite. Besides, lower and upper bounds can be equal to each other. Thus, the following types of variables are possible:

Bounds of variable	Type of variable
$-\infty < x_k < +\infty$	Free (unbounded) variable
$l_k \leq x_k < +\infty$	Variable with lower bound
$-\infty < x_k \leq u_k$	Variable with upper bound
$l_k \leq x_k \leq u_k$	Double-bounded variable
$l_k = x_k = u_k$	Fixed variable

Note that the types of variables shown above are applicable to structural as well as to auxiliary variables.

To solve the LP problem (1.1)—(1.3) is to find such values of all structural and auxiliary variables, which:

a) satisfy to all the linear constraints (1.2), and

b) are within their bounds (1.3), and

c) provide a smallest (in the case of minimization) or a largest (in the case of maximization) value of the objective function (1.1).

For solving LP problems GLPK uses a well known numerical procedure called *the simplex method*. The simplex method performs iterations, where on each iteration it transforms the original system of equaility constraints (1.2) resolving them through different sets of variables to an equivalent system called *the simplex table* (or sometimes *the simplex tableau*), which has the following form:

$$
\begin{array}{rcl}
Z &=& d_1(x_N)_1 + \ \ d_2(x_N)_2 + \ldots + \ \ d_n(x_N)_n \\
(x_B)_1 &=& \alpha_{11}(x_N)_1 + \alpha_{12}(x_N)_2 + \ldots + \alpha_{1n}(x_N)_n \\
(x_B)_2 &=& \alpha_{21}(x_N)_1 + \alpha_{22}(x_N)_2 + \ldots + \alpha_{2n}(x_N)_n \\
&\ldots\ldots\ldots\ldots\ldots\ldots& \\
(x_B)_m &=& \alpha_{m1}(x_N)_1 + \alpha_{m2}(x_N)_2 + \ldots + \alpha_{mn}(x_N)_n
\end{array}
\tag{1.4}
$$

where: $(x_B)_1, (x_B)_2, \ldots, (x_B)_m$ — basic variables; $(x_N)_1, (x_N)_2, \ldots, (x_N)_n$ — non-basic variables; $d_1, d_2, \ldots, d_n$ — reduced costs; $\alpha_{11}, \alpha_{12}, \ldots, \alpha_{mn}$ — coefficients of the simplex table. (May note that the original LP problem (1.1)—(1.3) also has the form of a simplex table, where all equalities are resolved through auxiliary variables.)

From the linear programming theory it is well known that if an optimal solution of the LP problem (1.1)—(1.3) exists, it can always be written in the form (1.4), where non-basic variables are set on their bounds while values of the objective function and basic variables are determined by the corresponding equalities of the simplex table.

A set of values of all basic and non-basic variables determined by the simplex table is called *basic solution*. If all basic variables are within their bounds, the basic solution is called *(primal) feasible*, otherwise it is called *(primal) infeasible*. A feasible basic solution, which provides a smallest (in case of minimization) or a largest (in case of maximization) value of the objective function is called *optimal*. Therefore, for solving LP problem the simplex method tries to find its optimal basic solution.

Primal feasibility of some basic solution may be stated by simple checking if all basic variables are within their bounds. Basic solution is optimal if additionally the following optimality conditions are satisfied for all non-basic variables:

Status of $(x_N)_j$	Minimization	Maximization
$(x_N)_j$ is free	$d_j = 0$	$d_j = 0$
$(x_N)_j$ is on its lower bound	$d_j \geq 0$	$d_j \leq 0$
$(x_N)_j$ is on its upper bound	$d_j \leq 0$	$d_j \geq 0$

In other words, basic solution is optimal if there is no non-basic variable, which changing in the feasible direction (i.e. increasing if it is free or on its lower bound, or decreasing if it is free or on its upper bound) can improve (i.e. decrease in case of minimization or increase in case of maximization) the objective function.

If all non-basic variables satisfy to the optimality conditions shown above (independently on whether basic variables are within their bounds or not), the basic solution is called *dual feasible*, otherwise it is called *dual infeasible*.

It may happen that some LP problem has no primal feasible solution due to incorrect formulation — this means that its constraints conflict with each other. It also may happen that some LP problem has unbounded solution again due to incorrect formulation — this means that some non-basic variable can improve the objective function, i.e. the optimality conditions are violated, and at the same time this variable can infinitely change in the feasible direction meeting no resistance from basic variables. (May note that in the latter case the LP problem has no dual feasible solution.)

1.2 MIP Problem

Mixed integer linear programming (MIP) problem is LP problem in which some variables are additionally required to be integer.

GLPK assumes that MIP problem has the same formulation as ordinary (pure) LP problem (1.1)—(1.3), i.e. includes auxiliary and structural variables, which may have lower and/or upper bounds. However, in case of MIP problem some variables may be required to be integer. This additional constraint means that a value of each *integer variable* must be only integer number. (Should note that GLPK allows only structural variables to be of integer kind.)

1.3 Brief Example

In order to understand what GLPK is from the user's standpoint, consider the following simple LP problem:

maximize

$$Z = 10x_1 + 6x_2 + 4x_3$$

subject to

$$\begin{array}{r} x_1 + x_2 + x_3 \leq 100 \\ 10x_1 + 4x_2 + 5x_3 \leq 600 \\ 2x_1 + 2x_2 + 6x_3 \leq 300 \end{array}$$

where all variables are non-negative

$$x_1 \geq 0, \ x_2 \geq 0, \ x_3 \geq 0$$

At first this LP problem should be transformed to the standard form (1.1)—(1.3). This can be easily done by introducing auxiliary variables, by one for each original inequality constraint. Thus, the problem can be reformulated as follows:

maximize

$$Z = 10x_1 + 6x_2 + 4x_3$$

subject to

$$\begin{aligned} p &= \ \ x_1 + \ x_2 + \ x_3 \\ q &= 10x_1 + 4x_2 + 5x_3 \\ r &= \ 2x_1 + 2x_2 + 6x_3 \end{aligned}$$

and bounds of variables

$$\begin{array}{ll} -\infty < p \leq 100 & 0 \leq x_1 < +\infty \\ -\infty < q \leq 600 & 0 \leq x_2 < +\infty \\ -\infty < r \leq 300 & 0 \leq x_3 < +\infty \end{array}$$

where p, q, r are auxiliary variables (rows), and x_1, x_2, x_3 are structural variables (columns).

The example C program shown below uses GLPK API routines in order to solve this LP problem.

```
/* sample.c */

#include <stdio.h>
#include <stdlib.h>
#include "glpk.h"

int main(void)
{     LPX *lp;
      int ia[1+1000], ja[1+1000];
      double ar[1+1000], Z, x1, x2, x3;
s1:   lp = lpx_create_prob();
s2:   lpx_set_prob_name(lp, "sample");
s3:   lpx_set_obj_dir(lp, LPX_MAX);
s4:   lpx_add_rows(lp, 3);
s5:   lpx_set_row_name(lp, 1, "p");
s6:   lpx_set_row_bnds(lp, 1, LPX_UP, 0.0, 100.0);
s7:   lpx_set_row_name(lp, 2, "q");
s8:   lpx_set_row_bnds(lp, 2, LPX_UP, 0.0, 600.0);
s9:   lpx_set_row_name(lp, 3, "r");
s10:  lpx_set_row_bnds(lp, 3, LPX_UP, 0.0, 300.0);
s11:  lpx_add_cols(lp, 3);
s12:  lpx_set_col_name(lp, 1, "x1");
s13:  lpx_set_col_bnds(lp, 1, LPX_LO, 0.0, 0.0);
s14:  lpx_set_obj_coef(lp, 1, 10.0);
s15:  lpx_set_col_name(lp, 2, "x2");
s16:  lpx_set_col_bnds(lp, 2, LPX_LO, 0.0, 0.0);
s17:  lpx_set_obj_coef(lp, 2, 6.0);
s18:  lpx_set_col_name(lp, 3, "x3");
s19:  lpx_set_col_bnds(lp, 3, LPX_LO, 0.0, 0.0);
s20:  lpx_set_obj_coef(lp, 3, 4.0);
s21:  ia[1] = 1, ja[1] = 1, ar[1] =  1.0; /* a[1,1] =  1 */
s22:  ia[2] = 1, ja[2] = 2, ar[2] =  1.0; /* a[1,2] =  1 */
```

```
s23:  ia[3] = 1, ja[3] = 3, ar[3] =  1.0; /* a[1,3] =  1 */
s24:  ia[4] = 2, ja[4] = 1, ar[4] = 10.0; /* a[2,1] = 10 */
s25:  ia[5] = 3, ja[5] = 1, ar[5] =  2.0; /* a[3,1] =  2 */
s26:  ia[6] = 2, ja[6] = 2, ar[6] =  4.0; /* a[2,2] =  4 */
s27:  ia[7] = 3, ja[7] = 2, ar[7] =  2.0; /* a[3,2] =  2 */
s28:  ia[8] = 2, ja[8] = 3, ar[8] =  5.0; /* a[2,3] =  5 */
s29:  ia[9] = 3, ja[9] = 3, ar[9] =  6.0; /* a[3,3] =  6 */
s30:  lpx_load_matrix(lp, 9, ia, ja, ar);
s31:  lpx_simplex(lp);
s32:  Z = lpx_get_obj_val(lp);
s33:  x1 = lpx_get_col_prim(lp, 1);
s34:  x2 = lpx_get_col_prim(lp, 2);
s35:  x3 = lpx_get_col_prim(lp, 3);
s36:  printf("\nZ = %g; x1 = %g; x2 = %g; x3 = %g\n", Z, x1, x2, x3);
s37:  lpx_delete_prob(lp);
      return 0;
}

/* eof */
```

The statement `s1` creates a problem object using the routine `lpx_create_prob`. Being created the object is initially empty. The statement `s2` assigns a symbolic name to the problem object.

The statement `s3` calls the routine `lpx_set_obj_dir` in order to set the optimization direction flag, where `LPX_MAX` means maximization.

The statement `s4` adds three rows to the problem object.

The statement `s5` assigns the symbolic name '`p`' to the first row, and the statement `s6` sets the type and bounds of the first row, where `LPX_UP` means that the row has an upper bound. The statements `s7`, `s8`, `s9`, `s10` are used in the same way in order to assign the symbolic names '`q`' and '`r`' to the second and third rows and set their types and bounds.

The statement `s11` adds three columns to the problem object.

The statement `s12` assigns the symbolic name '`x1`' to the first column, the statement `s13` sets the type and bounds of the first column, where `LPX_LO` means that the column has an lower bound, and the statement `s14` sets the objective coefficient for the first column. The statements `s15`—`s20` are used in the same way in order to assign the symbolic names '`x2`' and '`x3`' to the second and third columns and set their types, bounds, and objective coefficients.

The statements `s21`—`s29` prepare non-zero elements of the constraint matrix (i.e. constraint coefficients). Row indices of each element are stored in the array `ia`, column indices are stored in the array `ja`, and numerical values of corresponding elements are stored in the array `ar`. Then the statement `s30` calls the routine `lpx_load_matrix`, which loads information from these three arrays into the problem object.

Now all data have been entered into the problem object, and therefore the statement `s31` calls the routine `lpx_simplex`, which is a driver to the simplex method, in order to solve the LP problem. This routine finds an optimal solution and stores all relevant information back into the problem object.

The statement `s32` obtains a computed value of the objective function, and the statements `s33`—`s35` obtain computed values of structural variables (columns), which correspond to the optimal basic solution found by the solver.

The statement `s36` prints the optimal solution to the standard output. The printout may look like follows:

```
Z = 733.333; x1 = 33.3333; x2 = 66.6667; x3 = 0
```

Finally, the statement `s37` calls the routine `lpx_delete_prob`, which frees all the memory allocated to the problem object.

Chapter 2

API Routines

This chapter describes GLPK API routines intended for using in application programs.

Error handling If some GLPK API routine detects erroneous or incorrect data passed by the application program, it sends appropriate diagnostic messages to the standard output and then abnormally terminates the application program. In most practical cases this allows to simplify programming avoiding numerous checks of return codes. Thus, in order to prevent crashing the application program should check all data, which are suspected to be incorrect, before calling GLPK API routines.

Should note that this kind of error handling is used only in cases of incorrect data passed by the application program. If, for example, the application program calls some GLPK API routine to read data from an input file and these data are incorrect, the GLPK API routine reports about error in the usual way by means of return code.

Thread safety Currently GLPK API routines are non-reentrant and therefore cannot be used in multi-thread programs.

Array indexing Normally all GLPK routines start array indexing from 1, not from 0 (except the specially stipulated cases). This means, for example, if some vector x of the length n is passed as an array to some GLPK routine, the latter expects vector components to be placed in locations `x[1]`, `x[2]`, ..., `x[n]`, and the location `x[0]` normally is not used.

In order to avoid indexing errors it is most convenient and most reliable to declare the array `x` as follows:

```
double x[1+n];
```

or to allocate it as follows:

```
double *x;
. . .
x = calloc(1+n, sizeof(double));
```

In both cases one extra location `x[0]` is reserved that allows passing this array to GLPK routines in a usual way.

Using GLPK routines in C++ programs If you need to use GLPK routines in C++ programs, use the following construction:

```
extern "C" {
#include "glpk.h"
}
```

2.1 Problem object

GLPK API routines deal with so called *problem objects*, which are program objects of type `LPX` intended to represent particular LP and MIP problem instances.

The type `LPX` is a data structure declared in the header file `glpk.h` as follows:

```
typedef struct { ... } LPX;
```

Problem objects (i.e. program objects of the `LPX` type) are allocated and managed internally by the GLPK API routines. The application program should never use any members of the LPX structure directly and should deal only with pointers to these objects (that is, `LPX *` values).

Each problem object consists of four logical segments, which are:

- problem segment,
- basis segment,
- interior point segment,
- MIP segment, and
- control parameters and statistics segment.

Problem segment The *problem segment* contains original LP/MIP data, which corresponds to the problem formulation (1.1)—(1.3) (see Section 1.1, page 7):

- rows (auxiliary variables),
- columns (structural variables),
- objective function, and
- constraint matrix.

Rows and columns have the same set of the following attributes:

- ordinal number,
- symbolic name (1 up to 255 arbitrary graphic characters),
- type (free, lower bound, upper bound, double bound, fixed),
- numerical values of lower and upper bounds,
- scale factor.

Ordinal numbers are intended for referencing rows and columns. Row ordinal numbers are integers $1, 2, \ldots, m$, and column ordinal numbers are integers $1, 2, \ldots, n$, where m and n are, respectively, the current number of rows and columns in the problem object.

Symbolic names are intended only for informational purposes. They cannot be used for referencing rows and columns.

Types and bounds of rows (auxiliary variables) and columns (structural variables) are explained above (see Section 1.1, page 7).

Scale factors are used internally for scaling corresponding rows and columns of the constraint matrix.

Information about the *objective function* includes numerical values of objective coefficients and a flag, which defines the optimization direction (i.e. minimization or maximization).

The *constraint matrix* is a $m \times n$ rectangular matrix built of constraint coefficients a_{ij}, which defines the system of linear constraints (1.2) (see Section 1.1, page 7). This matrix is stored in the problem object in both row-wise and column-wise sparse formats.

Once the problem object has been created, the application program can access and modify any components of the problem segment in arbitrary order.

Basis segment The *basis segment* of the problem object keeps information related to a current basic solution. This information includes:

- row and column statuses,
- basic solution statuses,
- factorization of the current basis matrix, and
- basic solution components.

The *row and column statuses* define which rows and columns are basic and which are non-basic. These statuses may be assigned either by the application program of by some API routines. Note that these statuses are always defined independently on whether the corresponding basis is valid or not.

The *basic solution statuses* include the *primal status* and the *dual status*, which are set by the simplex-based solver once the problem has been solved. The primal status shows whether a primal basic solution is feasible, infeasible, or undefined. The dual status shows the same for a dual basic solution.

The *factorization of the basis matrix* is some factorized form (like LU-factorization) of the current basis matrix (defined by the current row and column statuses). The factorization is used by the simplex-based solver and kept when the solver terminates the search. This feature allows efficiently reoptimizing the problem after some modifications (for example, after changing some bounds or objective coefficients). It also allows performing a post-optimal analysis (for example, computing components of the simplex table, etc.).

The *basic solution components* include primal and dual values of all auxiliary and structural variables for the most recently obtained basic solution.

Interior point segment The *interior point segment* is automatically allocated after the problem has been solved using the interior point solver. It contains interior point solution components, which include the solution status, and primal and dual values of all auxiliary and structural variables.

MIP segment The *MIP segment* is used only for MIP problems. This segment includes:

- column kinds,
- MIP solution status, and
- MIP solution components.

The *column kinds* define which columns (i.e. structural variables) are integer and which are continuous.

The *MIP solution status* is set by the MIP solver and shows whether a MIP solution is integer optimal, integer feasible (non-optimal), or undefined.

The *MIP solution components* are computed by the MIP solver and includes primal values of all auxiliary and structural variables for the most recently obtained MIP solution.

Note that in the case of MIP problem the basis segment corresponds to an optimal solution of LP relaxation, which is also available to the application program.

Currently the search tree is not kept in the MIP segment. Therefore if the search has been terminated, it cannot be continued.

Control parameters and statistics segment This segment contains a fixed set of parameters, where each parameter has the following three attributes:

- code,
- type, and
- current value.

The *parameter code* is intended for referencing a particular parameter. All the parameter codes have symbolic names, which are macros defined in the header file `glpk.h`. Note that the parameter codes are distinct positive integers.

The *parameter type* can be integer, real (floating-point), and text (character string).

The *parameter value* is its current value kept in the problem object. Initially (after the problem object has been created) all parameters are assigned some default values.

Parameters are intended for several purposes. Some of them, which are called *control parameters*, affect the behavior of API routines (for example, the parameter `LPX_K_ITLIM` limits maximal number of simplex iterations available to the solver). Others, which are called *statistics*, just represent some additional information about the problem object (for example, the parameter `LPX_K_ITCNT` shows how many simplex iterations were performed for a particular problem object).

2.2 Problem creating and modifying routines

2.2.1 lpx_create_prob — create problem object

Synopsis

```
#include "glpk.h"
LPX *lpx_create_prob(void);
```

Description The routine `lpx_create_prob` creates a new problem object, which is "empty", i.e. has no rows and no columns.

Returns The routine returns a pointer to the created object, which should be used in any subsequent operations on this object.

2.2.2 lpx_set_prob_name — assign (change) problem name

Synopsis

```
#include "glpk.h"
void lpx_set_prob_name(LPX *lp, char *name);
```

Description The routine `lpx_set_prob_name` assigns a given symbolic `name` (1 up to 255 characters) to the specified problem object.

If the parameter `name` is `NULL` or empty string, the routine erases an existing symbolic name of the problem object.

2.2.3 lpx_set_obj_name — assign (change) objective function name

Synopsis

```
#include "glpk.h"
void lpx_set_obj_name(LPX *lp, char *name);
```

Description The routine `lpx_set_obj_name` assigns a given symbolic `name` (1 up to 255 characters) to the objective function of the specified problem object.

If the parameter `name` is `NULL` or empty string, the routine erases an existing symbolic name of the objective function.

2.2.4 lpx_set_obj_dir — set (change) optimization direction flag

Synopsis

```
#include "glpk.h"
void lpx_set_obj_dir(LPX *lp, int dir);
```

Description The routine `lpx_set_obj_dir` sets (changes) the optimization direction flag (i.e. "sense" of the objective function) as specified by the parameter `dir`:

`LPX_MIN` minimization;
`LPX_MAX` maximization.

2.2.5 `lpx_add_rows` — add new rows to problem object

Synopsis

```
#include "glpk.h"
int lpx_add_rows(LPX *lp, int nrs);
```

Description The routine `lpx_add_rows` adds `nrs` rows (constraints) to the specified problem object. New rows are always added to the end of the row list, so the ordinal numbers of existing rows are not changed.

Being added each new row is initially free (unbounded) and has empty list of the constraint coefficients.

Returns The routine `lpx_add_rows` returns the ordinal number of the first new row added to the problem object.

2.2.6 `lpx_add_cols` — add new columns to problem object

Synopsis

```
#include "glpk.h"
int lpx_add_cols(LPX *lp, int ncs);
```

Description The routine `lpx_add_cols` adds `ncs` columns (structural variables) to the specified problem object. New columns are always added to the end of the column list, so the ordinal numbers of existing columns are not changed.

Being added each new column is initially fixed at zero and has empty list of the constraint coefficients.

Returns The routine `lpx_add_cols` returns the ordinal number of the first new column added to the problem object.

2.2.7 `lpx_set_row_name` — assign (change) row name

Synopsis

```
#include "glpk.h"
void lpx_set_row_name(LPX *lp, int i, char *name);
```

Description The routine `lpx_set_row_name` assigns a given symbolic `name` (1 up to 255 characters) to `i`-th row (auxiliary variable) of the specified problem object.

If the parameter `name` is `NULL` or empty string, the routine erases an existing name of i-th row.

2.2.8 `lpx_set_col_name` — assign (change) column name

Synopsis

```
#include "glpk.h"
void lpx_set_col_name(LPX *lp, int j, char *name);
```

Description The routine `lpx_set_col_name` assigns a given symbolic `name` (1 up to 255 characters) to `j`-th column (structural variable) of the specified problem object.

If the parameter `name` is `NULL` or empty string, the routine erases an existing name of j-th column.

2.2.9 `lpx_set_row_bnds` — set (change) row bounds

Synopsis

```
#include "glpk.h"
void lpx_set_row_bnds(LPX *lp, int i, int type, double lb, double ub);
```

Description The routine `lpx_set_row_bnds` sets (changes) the type and bounds of `i`-th row (auxiliary variable) of the specified problem object.

The parameters `type`, `lb`, and `ub` specify the type, lower bound, and upper bound, respectively, as follows:

Type	Bounds	Comment
`LPX_FR`	$-\infty < x < +\infty$	Free (unbounded) variable
`LPX_LO`	$lb \leq x < +\infty$	Variable with lower bound
`LPX_UP`	$-\infty < x \leq ub$	Variable with upper bound
`LPX_DB`	$lb \leq x \leq ub$	Double-bounded variable
`LPX_FX`	$lb = x = ub$	Fixed variable

where x is the auxiliary variable associated with i-th row.

If the row has no lower bound, the parameter `lb` is ignored. If the row has no upper bound, the parameter `ub` is ignored. If the row is an equality constraint (i.e. the corresponding auxiliary variable is of fixed type), only the parameter `lb` is used while the parameter `ub` is ignored.

2.2.10 `lpx_set_col_bnds` — set (change) column bounds

Synopsis

```
#include "glpk.h"
void lpx_set_col_bnds(LPX *lp, int j, int type, double lb, double ub);
```

Description The routine `lpx_set_col_bnds` sets (changes) the type and bounds of `j`-th column (structural variable) of the specified problem object.

The parameters `type`, `lb`, and `ub` specify the type, lower bound, and upper bound, respectively, as follows:

Type	Bounds	Comment
`LPX_FR`	$-\infty < x < +\infty$	Free (unbounded) variable
`LPX_LO`	$lb \leq x < +\infty$	Variable with lower bound
`LPX_UP`	$-\infty < x \leq ub$	Variable with upper bound
`LPX_DB`	$lb \leq x \leq ub$	Double-bounded variable
`LPX_FX`	$lb = x = ub$	Fixed variable

where x is the structural variable associated with j-th column.

If the column has no lower bound, the parameter `lb` is ignored. If the column has no upper bound, the parameter `ub` is ignored. If the column is of fixed type, only the parameter `lb` is used while the parameter `ub` is ignored.

2.2.11 `lpx_set_obj_coef` — set (change) objective coefficient or constant term

Synopsis

```
#include "glpk.h"
void lpx_set_obj_coef(LPX *lp, int j, double coef);
```

Description The routine `lpx_set_obj_coef` sets (changes) the objective coefficient at `j`-th column (structural variable). A new value of the objective coefficient is specified by the parameter `coef`.

If the parameter `j` is 0, the routine sets (changes) the constant term ("shift") of the objective function.

2.2.12 `lpx_set_mat_row` — set (replace) row of the constraint matrix

Synopsis

```
#include "glpk.h"
void lpx_set_mat_row(LPX *lp, int i, int len, int ind[], double val[]);
```

Description The routine `lpx_set_mat_row` stores (replaces) the contents of `i`-th row of the constraint matrix of the specified problem object.

Column indices and numerical values of new row elements must be placed in locations `ind[1]`, ..., `ind[len]` and `val[1]`, ..., `val[len]`, respectively, where $0 \leq$ `len` $\leq n$ is the new length of i-th row, n is the current number of columns in the problem object. Note that zero elements as well as elements with identical column indices are not allowed.

If the parameter `len` is 0, the parameters `ind` and/or `val` can be specified as `NULL`.

2.2.13 `lpx_set_mat_col` — set (replace) column of the constraint matrix

Synopsis

```
#include "glpk.h"
void lpx_set_mat_col(LPX *lp, int j, int len, int ind[], double val[]);
```

Description The routine `lpx_set_mat_col` stores (replaces) the contents of `j`-th column of the constraint matrix of the specified problem object.

Row indices and numerical values of new column elements must be placed in locations `ind[1]`, ..., `ind[len]` and `val[1]`, ..., `val[len]`, respectively, where $0 \leq$ `len` $\leq m$ is the new length of j-th column, m is the current number of rows in the problem object. Note that zero elements as well as elements with identical row indices are not allowed.

If the parameter `len` is 0, the parameters `ind` and/or `val` can be specified as `NULL`.

2.2.14 `lpx_load_matrix` — load (replace) the whole constraint matrix

Synopsis

```
#include "glpk.h"
void lpx_load_matrix(LPX *lp, int ne, int ia[], int ja[], double ar[]);
```

Description The routine `lpx_load_matrix` loads the constraint matrix passed in the arrays `ia`, `ja`, and `ar` into the specified problem object. Before loading the current contents of the constraint matrix is destroyed.

Constraint coefficients (elements of the constraint matrix) must be specified as triplets (`ia[k]`, `ja[k]`, `ar[k]`) for $k = 1, \ldots, ne$, where `ia[k]` is the row index, `ja[k]` is the column index, and `ar[k]` is a numeric value of corresponding constraint coefficient. The parameter `ne` specifies the total number of (non-zero) elements in the matrix to be loaded. Note that coefficients with identical indices as well as zero coefficients are not allowed.

If the parameter `ne` is 0, the parameters `ia`, `ja`, and/or `ar` can be specified as `NULL`.

2.2.15 `lpx_del_rows` — delete rows from problem object

Synopsis

```
#include "glpk.h"
void lpx_del_rows(LPX *lp, int nrs, int num[]);
```

Description The routine `lpx_del_rows` deletes specified rows from a problem object, which the parameter `lp` points to. Ordinal numbers of rows to be deleted must be placed in locations `num[1]`, ..., `num[nrs]`, where `nrs` > 0.

Note that deleting rows involves changing ordinal numbers of other rows remaining in the problem object. New ordinal numbers of the remaining rows are assigned under the assumption that the original order of rows is not changed. Let, for example, before deletion there be five rows a, b, c, d, e with ordinal numbers 1, 2, 3, 4, 5, and let rows b and d have been deleted. Then after deletion the remaining rows a, c, e are assigned new oridinal numbers 1, 2, 3.

2.2.16 `lpx_del_cols` — delete columns from problem object

Synopsis

```
#include "glpk.h"
void lpx_del_cols(LPX *lp, int ncs, int num[]);
```

Description The routine `lpx_del_cols` deletes specified columns from a problem object, which the parameter `lp` points to. Ordinal numbers of columns to be deleted must be placed in locations `num[1]`, . . . , `num[ncs]`, where `ncs` > 0.

Note that deleting columns involves changing ordinal numbers of other columns remaining in the problem object. New ordinal numbers of the remaining columns are assigned under the assumption that the original order of columns is not changed. Let, for example, before deletion there be six columns p, q, r, s, t, u with ordinal numbers 1, 2, 3, 4, 5, 6, and let columns p, q, s have been deleted. Then after deletion the remaining columns r, t, u are assigned new ordinal numbers 1, 2, 3.

2.2.17 `lpx_delete_prob` — delete problem object

Synopsis

```
#include "glpk.h"
void lpx_delete_prob(LPX *lp);
```

Description The routine `lpx_delete_prob` deletes a problem object, which the parameter `lp` points to, freeing all the memory allocated to this object.

2.3 Problem retrieving routines

2.3.1 `lpx_get_prob_name` — retrieve problem name

Synopsis

```
#include "glpk.h"
char *lpx_get_prob_name(LPX *lp);
```

Returns The routine `lpx_get_prob_name` returns a pointer to an internal buffer, which contains symbolic name of the problem. However, if the problem has no assigned name, the routine returns `NULL`.

2.3.2 `lpx_get_obj_name` — retrieve objective function name

Synopsis

```
#include "glpk.h"
char *lpx_get_obj_name(LPX *lp);
```

Returns The routine `lpx_get_obj_name` returns a pointer to an internal buffer, which contains symbolic name assigned to the objective function. However, if the objective function has no assigned name, the routine returns `NULL`.

2.3.3 `lpx_get_obj_dir` — retrieve optimization direction flag

Synopsis

```
#include "glpk.h"
int lpx_get_obj_dir(LPX *lp);
```

Returns The routine `lpx_get_obj_dir` returns the optimization direction flag (i.e. "sense" of the objective function):

`LPX_MIN` minimization;
`LPX_MAX` maximization.

2.3.4 `lpx_get_num_rows` — retrieve number of rows

Synopsis

```
#include "glpk.h"
int lpx_get_num_rows(LPX *lp);
```

Returns The routine `lpx_get_num_rows` returns the current number of rows in the specified problem object.

2.3.5 lpx_get_num_cols — retrieve number of columns

Synopsis

```
#include "glpk.h"
int lpx_get_num_cols(LPX *lp);
```

Returns The routine `lpx_get_num_cols` returns the current number of columns the specified problem object.

2.3.6 lpx_get_row_name — retrieve row name

Synopsis

```
#include "glpk.h"
char *lpx_get_row_name(LPX *lp, int i);
```

Returns The routine `lpx_get_row_name` returns a pointer to an internal buffer, which contains a symbolic name assigned to `i`-th row. However, if the row has no assigned name, the routine returns `NULL`.

2.3.7 lpx_get_col_name — retrieve column name

Synopsis

```
#include "glpk.h"
char *lpx_get_col_name(LPX *lp, int j);
```

Returns The routine `lpx_get_col_name` returns a pointer to an internal buffer, which contains a symbolic name assigned to `j`-th column. However, if the column has no assigned name, the routine returns `NULL`.

2.3.8 lpx_get_row_type — retrieve row type

Synopsis

```
#include "glpk.h"
int lpx_get_row_type(LPX *lp, int i);
```

Returns The routine `lpx_get_row_type` returns the type of `i`-th row, i.e. the type of corresponding auxiliary variable, as follows:

- `LPX_FR` free (unbounded) variable;
- `LPX_LO` variable with lower bound;
- `LPX_UP` variable with upper bound;
- `LPX_DB` double-bounded variable;
- `LPX_FX` fixed variable.

2.3.9 lpx_get_row_lb — retrieve row lower bound

Synopsis

```
#include "glpk.h"
double lpx_get_row_lb(LPX *lp, int i);
```

Returns The routine `lpx_get_row_lb` returns the lower bound of `i`-th row, i.e. the lower bound of corresponding auxiliary variable. However, if the row has no lower bound, the routine returns zero.

2.3.10 lpx_get_row_ub — retrieve row upper bound

Synopsis

```
#include "glpk.h"
double lpx_get_row_ub(LPX *lp, int i);
```

Returns The routine `lpx_get_row_ub` returns the upper bound of `i`-th row, i.e. the upper bound of corresponding auxiliary variable. However, if the row has no upper bound, the routine returns zero.

2.3.11 lpx_get_col_type — retrieve column type

Synopsis

```
#include "glpk.h"
int lpx_get_col_type(LPX *lp, int j);
```

Returns The routine `lpx_get_col_type` returns the type of `j`-th column, i.e. the type of corresponding structural variable, as follows:

`LPX_FR` free (unbounded) variable;
`LPX_LO` variable with lower bound;
`LPX_UP` variable with upper bound;
`LPX_DB` double-bounded variable;
`LPX_FX` fixed variable.

2.3.12 lpx_get_col_lb — retrieve column lower bound

Synopsis

```
#include "glpk.h"
double lpx_get_col_lb(LPX *lp, int j);
```

Returns The routine `lpx_get_col_lb` returns the lower bound of `j`-th column, i.e. the lower bound of corresponding structural variable. However, if the column has no lower bound, the routine returns zero.

2.3.13 lpx_get_col_ub — retrieve column upper bound

Synopsis

```
#include "glpk.h"
double lpx_get_col_ub(LPX *lp, int j);
```

Returns The routine `lpx_get_col_ub` returns the upper bound of `j`-th column, i.e. the upper bound of corresponding structural variable. However, if the column has no upper bound, the routine returns zero.

2.3.14 lpx_get_obj_coef — retrieve objective coefficient or constant term

Synopsis

```
#include "glpk.h"
double lpx_get_obj_coef(LPX *lp, int j);
```

Returns The routine `lpx_get_obj_coef` returns the objective coefficient at `j`-th structural variable (column).

If the parameter `j` is 0, the routine returns the constant term ("shift") of the objective function.

2.3.15 lpx_get_num_nz — retrieve number of constraint coefficients

Synopsis

```
#include "glpk.h"
int lpx_get_num_nz(LPX *lp);
```

Returns The routine `lpx_get_num_nz` returns the number of non-zero elements in the constraint matrix of the specified problem object.

2.3.16 lpx_get_mat_row — retrieve row of the constraint matrix

Synopsis

```
#include "glpk.h"
int lpx_get_mat_row(LPX *lp, int i, int ind[], double val[]);
```

Description The routine `lpx_get_mat_row` scans (non-zero) elements of `i`-th row of the constraint matrix of the specified problem object and stores their column indices and numeric values to locations `ind[1]`, ..., `ind[len]` and `val[1]`, ..., `val[len]`, respectively, where $0 \leq$ `len` $\leq n$ is the number of elements in i-th row, n is the number of columns.

The parameter `ind` and/or `val` can be specified as `NULL`, in which case corresponding information is not stored.

Returns The routine `lpx_get_mat_row` returns the length `len`, i.e. the number of (non-zero) elements in `i`-th row.

2.3.17 `lpx_get_mat_col` — retrieve column of the constraint matrix

Synopsis

```
#include "glpk.h"
int lpx_get_mat_col(LPX *lp, int j, int ind[], double val[]);
```

Description The routine `lpx_get_mat_col` scans (non-zero) elements of `j`-th column of the constraint matrix of the specified problem object and stores their row indices and numeric values to locations `ind[1]`, ..., `ind[len]` and `val[1]`, ..., `val[len]`, respectively, where $0 \leq$ `len` $\leq m$ is the number of elements in j-th column, m is the number of rows.

The parameter `ind` and/or `val` can be specified as `NULL`, in which case corresponding information is not stored.

Returns The routine `lpx_get_mat_col` returns the length `len`, i.e. the number of (non-zero) elements in `j`-th column.

2.4 Problem scaling routines

2.4.1 `lpx_scale_prob` — scale problem data

Synopsis

```
#include "glpk.h"
void lpx_scale_prob(LPX *lp);
```

Description The routine `lpx_scale_prob` performs scaling of problem data for the specified problem object.

The purpose of scaling is to provide such scaling (diagonal) matrices R and S that the scaled constraint matrix $A' = RAS$ has better numerical properties than the original constraint matrix A.

Note that the scaling matrices R and S are used only by the solver. On API level the scaling is invisible, since all data stored in the problem object are non-scaled.

2.4.2 `lpx_unscale_prob` — unscale problem data

Synopsis

```
#include "glpk.h"
void lpx_unscale_prob(LPX *lp);
```

The routine `lpx_unscale_prob` performs unscaling of problem data for the specified problem object.

"Unscaling" means replacing the current scaling matrices R and S by unity matrices that cancels the scaling effect.

2.5 LP basis constructing routines

2.5.1 lpx_std_basis — construct standard initial LP basis

Synopsis

```
#include "glpk.h"
void lpx_std_basis(LPX *lp);
```

Description The routine `lpx_std_basis` constructs the "standard" (trivial) initial LP basis for the specified problem object.

In the "standard" LP basis all auxiliary variables (rows) are basic, and all structural variables (columns) are non-basic (so the corresponding basis matrix is unity).

2.5.2 lpx_adv_basis — construct advanced initial LP basis

Synopsis

```
#include "glpk.h"
void lpx_adv_basis(LPX *lp);
```

Description The routine `lpx_adv_basis` build an advanced initial LP basis for the specified problem object.

In order to construct the advanced initial LP basis the routine does the following:

1) includes in the basis all non-fixed auxiliary variables;

2) includes in the basis as many non-fixed structural variables as possible keeping triangular form of the basis matrix;

3) includes in the basis appropriate (fixed) auxiliary variables to complete the basis.

As a result the initial LP basis has as few fixed variables as possible and the corresponding basis matrix is triangular.

2.5.3 lpx_set_row_stat — set (change) row status

Synopsis

```
#include "glpk.h"
void lpx_set_row_stat(LPX *lp, int i, int stat);
```

Description The routine `lpx_set_row_stat` sets (changes) the current status of `i`-th row (auxiliary variable) as specified by the parameter `stat`:

`LPX_BS` make the row basic (make the constraint inactive);
`LPX_NL` make the row non-basic (make the constraint active);
`LPX_NU` make the row non-basic and set it to the upper bound; if the row is not double-bounded, this status is equivalent to `LPX_NL` (only in the case of this routine);
`LPX_NF` the same as `LPX_NL` (only in the case of this routine);
`LPX_NS` the same as `LPX_NL` (only in the case of this routine).

2.5.4 lpx_set_col_stat — set (change) column status

Synopsis

```
#include "glpk.h"
void lpx_set_col_stat(LPX *lp, int j, int stat);
```

Description The routine `lpx_set_col_stat` sets (changes) the current status of `j`-th column (structural variable) as specified by the parameter `stat`:

- `LPX_BS` make the column basic;
- `LPX_NL` make the column non-basic;
- `LPX_NU` make the column non-basic and set it to the upper bound; if the column is not double-bounded, this status is equivalent to `LPX_NL` (only in the case of this routine);
- `LPX_NF` the same as `LPX_NL` (only in the case of this routine);
- `LPX_NS` the same as `LPX_NL` (only in the case of this routine).

2.6 Simplex method routine

2.6.1 `lpx_simplex` — solve LP problem using the simplex method

Synopsis

```
#include "glpk.h"
int lpx_simplex(LPX *lp);
```

Description The routine `lpx_simplex` is an interface to an LP problem solver based on the two-phase revised simplex method.

This routine obtains problem data from the problem object, which the parameter `lp` points to, calls the solver to solve the LP problem, and stores an obtained basic solution and other relevant information back into the problem object.

Since solving of large-scale problems may take a long time, the solver reports some information about the current basic solution, which is sent to the standard output. This information has the following format:

```
*nnn:   objval = xxx   infeas = yyy (ddd)
```

where: '`nnn`' is the iteration number, '`xxx`' is the current value of the objective function (which is unscaled and has correct sign), '`yyy`' is the current sum of primal infeasibilities (which is scaled and therefore may be used for visual estimating only), '`ddd`' is the current number of fixed basic variables. If the asterisk '`*`' precedes to '`nnn`', the solver is searching for an optimal solution (phase II), otherwise the solver is searching for a primal feasible solution (phase I).

Note that the simplex solver currently implemented in GLPK is not perfect. Although it has been successfully tested on a wide set of LP problems, there are hard problems, which cannot be solved by the GLPK simplex solver.

Using built-in LP presolver The simplex solver has *built-in LP presolver*, which is a subprogram that transforms the original LP problem specified in the problem object to an equivalent LP problem, which may be easier for solving with the simplex method than the original one. This is attained mainly due to reducing the problem size and improving its numeric properties (for example, by removing some inactive constraints or by fixing some non-basic variables). Once the transformed LP problem has been solved, the presolver transforms its basic solution back to a corresponding basic solution of the original problem.

Presolving is an optional feature of the routine `lpx_simplex`, and by default it is disabled. In order to enable the LP presolver the user should set the control parameter `LPX_K_PRESOL` on (see Subsection 2.11.6, page 54) before calling the routine `lpx_simplex`. As a rule presolving is useful when the problem is solved for the first time, and it is not recommended to use presolving when the problem should be re-optimized.

The presolving procedure is transparent to the API user in the sense that all necessary processing is performed internally, and a basic solution of the original problem recovered by the presolver is the same as if it were computed directly, i.e. without presolving.

Note that the presolver is able to recover only optimal solutions. If a computed solution is infeasible or non-optimal, the corresponding solution of the original problem cannot be recovered and therefore remains undefined. If the user needs to know a basic solution even if it is infeasible or non-optimal, the presolver must be disabled.

Returns If the LP presolver is disabled (the flag `LPX_K_PRESOL` is off), the routine `lpx_simplex` returns one of the following exit codes:

`LPX_E_OK`	the LP problem has been successfully solved. (Note that, for example, if the problem has no feasible solution, this exit code is reported.)
`LPX_E_FAULT`	unable to start the search because either the problem has no rows/columns, or the initial basis is invalid, or the initial basis matrix is singular or ill-conditioned.
`LPX_E_OBJLL`	the search was prematurely terminated because the objective function being maximized has reached its lower limit and continues decreasing (the dual simplex only).
`LPX_E_OBJUL`	the search was prematurely terminated because the objective function being minimized has reached its upper limit and continues increasing (the dual simplex only).
`LPX_E_ITLIM`	the search was prematurely terminated because the simplex iterations limit has been exceeded.
`LPX_E_TMLIM`	the search was prematurely terminated because the time limit has been exceeded.
`LPX_E_SING`	the search was prematurely terminated due to the solver failure (the current basis matrix got singular or ill-conditioned).

If the LP presolver is enabled (the flag `LPX_K_PRESOL` is on), the routine `lpx_simplex` returns one of the following exit codes:

`LPX_E_OK`	optimal solution of the LP problem has been found.
`LPX_E_FAULT`	the LP problem has no rows and/or columns.
`LPX_E_NOPFS`	the LP problem has no primal feasible solution.
`LPX_E_NODFS`	the LP problem has no dual feasible solution.
`LPX_E_ITLIM`	same as above.
`LPX_E_TMLIB`	same as above.
`LPX_E_SING`	same as above.

2.7 Basic solution retrieving routines

2.7.1 lpx_get_status — retrieve generic status of basic solution

Synopsis

```
#include "glpk.h"
int lpx_get_status(LPX *lp);
```

Returns The routine `lpx_get_status` reports the generic status of the current basic solution for the specified problem object as follows:

`LPX_OPT`	solution is optimal;
`LPX_FEAS`	solution is feasible;
`LPX_INFEAS`	solution is infeasible;
`LPX_NOFEAS`	problem has no feasible solution;
`LPX_UNBND`	problem has unbounded solution;
`LPX_UNDEF`	solution is undefined.

More detailed information about the status of basic solution can be retrieved using the routines `lpx_get_prim_stat` and `lpx_get_dual_stat`.

2.7.2 lpx_get_prim_stat — retrieve primal status of basic solution

Synopsis

```
#include "glpk.h"
int lpx_get_prim_stat(LPX *lp);
```

Returns The routine `lpx_get_prim_stat` reports the primal status of the basic solution for the specified problem object as follows:

`LPX_P_UNDEF`	primal solution is undefined;
`LPX_P_FEAS`	solution is primal feasible;
`LPX_P_INFEAS`	solution is primal infeasible;
`LPX_P_NOFEAS`	no primal feasible solution exists.

2.7.3 lpx_get_dual_stat — retrieve dual status of basic solution

Synopsis

```
#include "glpk.h"
int lpx_get_dual_stat(LPX *lp);
```

Returns The routine `lpx_get_dual_stat` reports the dual status of the basic solution for the specified problem object as follows:

`LPX_D_UNDEF`	dual solution is undefined;
`LPX_D_FEAS`	solution is dual feasible;
`LPX_D_INFEAS`	solution is dual infeasible;
`LPX_D_NOFEAS`	no dual feasible solution exists.

2.7.4 lpx_get_obj_val — retrieve objective value

Synopsis

```
#include "glpk.h"
double lpx_get_obj_val(LPX *lp);
```

Returns The routine `lpx_get_obj_val` returns current value of the objective function.

2.7.5 lpx_get_row_stat — retrieve row status

Synopsis

```
#include "glpk.h"
int lpx_get_row_stat(LPX *lp, int i);
```

Returns The routine `lpx_get_row_stat` returns current status assigned to the auxiliary variable associated with `i`-th row as follows:

`LPX_BS`	basic variable;
`LPX_NL`	non-basic variable on its lower bound;
`LPX_NU`	non-basic variable on its upper bound;
`LPX_NF`	non-basic free (unbounded) variable;
`LPX_NS`	non-basic fixed variable.

2.7.6 lpx_get_row_prim — retrieve row primal value

Synopsis

```
#include "glpk.h"
double lpx_get_row_prim(LPX *lp, int i);
```

Returns The routine `lpx_get_row_prim` returns primal value of the auxiliary variable associated with `i`-th row.

2.7.7 lpx_get_row_dual — retrieve row dual value

Synopsis

```
#include "glpk.h"
double lpx_get_row_dual(LPX *lp, int i);
```

Returns The routine `lpx_get_row_dual` returns dual value (i.e. reduced cost) of the auxiliary variable associated with `i`-th row.

2.7.8 lpx_get_col_stat — retrieve column status

Synopsis

```
#include "glpk.h"
int lpx_get_col_stat(LPX *lp, int j);
```

Returns The routine `lpx_get_col_stat` returns current status assigned to the structural variable associated with `j`-th column as follows:

`LPX_BS`	basic variable;
`LPX_NL`	non-basic variable on its lower bound;
`LPX_NU`	non-basic variable on its upper bound;
`LPX_NF`	non-basic free (unbounded) variable;
`LPX_NS`	non-basic fixed variable.

2.7.9 lpx_get_col_prim — retrieve column primal value

Synopsis

```
#include "glpk.h"
double lpx_get_col_prim(LPX *lp, int j);
```

Returns The routine `lpx_get_col_prim` returns primal value of the structural variable associated with `j`-th column.

2.7.10 lpx_get_col_dual — retrieve column dual value

Synopsis

```
#include "glpk.h"
double lpx_get_col_dual(LPX *lp, int j);
```

Returns The routine `lpx_get_col_dual` returns dual value (i.e. reduced cost) of the structural variable associated with `j`-th column.

2.7.11 lpx_get_ray_info — retrieve non-basic variable which causes unboundness

Synopsis

```
#include "glpk.h"
int lpx_get_ray_info(LPX *lp);
```

Returns The routine `lpx_get_ray_info` returns the number k of some non-basic variable x_k, which causes primal unboundness. If such a variable cannot be identified, the routine returns zero.

If $1 \leq k \leq m$, x_k is k-th auxiliary variable, and if $m+1 \leq k \leq m+n$, x_k is $(k-m)$-th structural variable, where m is the number of rows, n is the number of columns in the specified problem object.

"Unboundness" means that the variable x_k is non-basic and able to *infinitely* change in a feasible direction improving the objective function.

2.7.12 `lpx_check_kkt` — check Karush-Kuhn-Tucker conditions

Synopsis

```
#include "glpk.h"
void lpx_check_kkt(LPX *lp, int scaled, LPXKKT *kkt);
```

Description The routine `lpx_check_kkt` checks Karush-Kuhn-Tucker optimality conditions for basic solution. It is assumed that both primal and dual components of basic solution are valid.

If the parameter `scaled` is zero, the optimality conditions are checked for the original, unscaled LP problem. Otherwise, if the parameter `scaled` is non-zero, the routine checks the conditions for an internally scaled LP problem.

The parameter `kkt` is a pointer to the structure `LPXKKT`, to which the routine stores the results of checking. Members of this structure are shown in the table below.

Condition	Member	Comment
(KKT.PE)	`pe_ae_max`	Largest absolute error
	`pe_ae_row`	Number of row with largest absolute error
	`pe_re_max`	Largest relative error
	`pe_re_row`	Number of row with largest relative error
	`pe_quality`	Quality of primal solution
(KKT.PB)	`pb_ae_max`	Largest absolute error
	`pb_ae_ind`	Number of variable with largest absolute error
	`pb_re_max`	Largest relative error
	`pb_re_ind`	Number of variable with largest relative error
	`pb_quality`	Quality of primal feasibility
(KKT.DE)	`de_ae_max`	Largest absolute error
	`de_ae_col`	Number of column with largest absolute error
	`de_re_max`	Largest relative error
	`de_re_col`	Number of column with largest relative error
	`de_quality`	Quality of dual solution
(KKT.DB)	`db_ae_max`	Largest absolute error
	`db_ae_ind`	Number of variable with largest absolute error
	`db_re_max`	Largest relative error
	`db_re_ind`	Number of variable with largest relative error
	`db_quality`	Quality of dual feasibility

The routine performs all computations using only components of the given LP problem and the current basic solution.

Background The first condition checked by the routine is:

$$x_R - Ax_S = 0, \tag{KKT.PE}$$

where x_R is the subvector of auxiliary variables (rows), x_S is the subvector of structural variables (columns), A is the constraint matrix. This condition expresses the requirement that all primal variables must satisfy to the system of equality constraints of the original LP problem. In case of exact arithmetic this condition would be satisfied for any basic solution; however, in case of inexact (floating-point) arithmetic, this condition shows how accurate the primal basic solution is, that depends on accuracy of a representation of the basis matrix used by the simplex method routines.

The second condition checked by the routine is:

$$l_k \leq x_k \leq u_k \quad \text{for all } k = 1, \ldots, m+n, \tag{KKT.PB}$$

where x_k is auxiliary ($1 \leq k \leq m$) or structural ($m+1 \leq k \leq m+n$) variable, l_k and u_k are, respectively, lower and upper bounds of the variable x_k (including cases of infinite bounds). This condition expresses the requirement that all primal variables must satisfy to bound constraints of the original LP problem. Since in case of basic solution all non-basic variables are placed on their bounds, actually the condition (KKT.PB) needs to be checked for basic variables only. If the primal basic solution has sufficient accuracy, this condition shows primal feasibility of the solution.

The third condition checked by the routine is:

$$\text{grad } Z = c = (\tilde{A})^T \pi + d,$$

where Z is the objective function, c is the vector of objective coefficients, $(\tilde{A})^T$ is a matrix transposed to the expanded constraint matrix $\tilde{A} = (I \mid -A)$, π is a vector of Lagrange multipliers that correspond to equality constraints of the original LP problem, d is a vector of Lagrange multipliers that correspond to bound constraints for all (auxiliary and structural) variables of the original LP problem. Geometrically the third condition expresses the requirement that the gradient of the objective function must belong to the orthogonal complement of a linear subspace defined by the equality and active bound constraints, i.e. that the gradient must be a linear combination of normals to the constraint planes, where Lagrange multipliers π and d are coefficients of that linear combination.

To eliminate the vector π the third condition can be rewritten as:

$$\begin{pmatrix} I \\ -A^T \end{pmatrix} \pi = \begin{pmatrix} d_R \\ d_S \end{pmatrix} + \begin{pmatrix} c_R \\ c_S \end{pmatrix},$$

or, equivalently:

$$\begin{aligned} \pi + d_R &= c_R, \\ -A^T\pi + d_S &= c_S. \end{aligned}$$

Then substituting the vector π from the first equation into the second one we have:

$$A^T(d_R - c_R) + (d_S - c_S) = 0, \tag{KKT.DE}$$

where d_R is the subvector of reduced costs of auxiliary variables (rows), d_S is the subvector of reduced costs of structural variables (columns), c_R and c_S are subvectors of objective coefficients at, respectively, auxiliary and structural variables, A^T is a matrix transposed to the constraint matrix of the original LP problem. In case of exact arithmetic this condition would be satisfied for any basic solution; however, in case of inexact (floating-point) arithmetic, this condition shows how accurate the dual basic solution is, that depends on accuracy of a representation of the basis matrix used by the simplex method routines.

The last, fourth condition checked by the routine is:

$$\begin{array}{rll} d_k = 0, & \text{if } x_k \text{ is basic or free non-basic variable} \\ 0 \leq d_k < +\infty & \text{if } x_k \text{ is non-basic on its lower (minimization)} \\ & \text{or upper (maximization) bound} \\ -\infty < d_k \leq 0 & \text{if } x_k \text{ is non-basic on its upper (minimization)} \\ & \text{or lower (maximization) bound} \\ -\infty < d_k < +\infty & \text{if } x_k \text{ is non-basic fixed variable} \end{array} \tag{KKT.DB}$$

for all $k = 1, \ldots, m+n$, where d_k is a reduced cost (Lagrange multiplier) of auxiliary $(1 \leq k \leq m)$ or structural $(m+1 \leq k \leq m+n)$ variable x_k. Geometrically this condition expresses the requirement that constraints of the original problem must "hold" the point preventing its movement along the anti-gradient (in case of minimization) or the gradient (in case of maximization) of the objective function. Since in case of basic solution reduced costs of all basic variables are placed on their (zero) bounds, actually the condition (KKT.DB) needs to be checked for non-basic variables only. If the dual basic solution has sufficient accuracy, this condition shows dual feasibility of the solution.

Should note that the complete set of Karush-Kuhn-Tucker optimality conditions also includes the fifth, so called complementary slackness condition, which expresses the requirement that at least either a primal variable x_k or its dual counterpart d_k must be on its bound for all $k = 1, \ldots, m+n$. However, being always satisfied by definition for any basic solution that condition is not checked by the routine.

To check the first condition (KKT.PE) the routine computes a vector of residuals:

$$g = x_R - Ax_S,$$

determines component of this vector that correspond to largest absolute and relative errors:

$$\texttt{pe\_ae\_max} = \max_{1 \leq i \leq m} |g_i|,$$

$$\texttt{pe\_re\_max} = \max_{1 \leq i \leq m} \frac{|g_i|}{1 + |(x_R)_i|},$$

and stores these quantities and corresponding row indices to the structure `LPXKKT`.

To check the second condition (KKT.PB) the routine computes a vector of residuals:

$$h_k = \left\{ \begin{array}{ll} 0, & \text{if } l_k \leq x_k \leq u_k \\ x_k - l_k, & \text{if } x_k < l_k \\ x_k - u_k, & \text{if } x_k > u_k \end{array} \right.$$

for all $k = 1, \ldots, m+n$, determines components of this vector that correspond to largest absolute and relative errors:

$$\texttt{pb\_ae\_max} = \max_{1 \leq k \leq m+n} |h_k|,$$

$$\texttt{pb\_re\_max} = \max_{1 \leq k \leq m+n} \frac{|h_k|}{1 + |x_k|},$$

and stores these quantities and corresponding variable indices to the structure `LPXKKT`.

To check the third condition (KKT.DE) the routine computes a vector of residuals:

$$u = A^T(d_R - c_R) + (d_S - c_S),$$

determines components of this vector that correspond to largest absolute and relative errors:

$$\texttt{de\_ae\_max} = \max_{1\leq j\leq n} |u_j|,$$

$$\texttt{de\_re\_max} = \max_{1\leq j\leq n} \frac{|u_j|}{1+|(d_S)_j - (c_S)_j|},$$

and stores these quantities and corresponding column indices to the structure `LPXKKT`.

To check the fourth condition (KKT.DB) the routine computes a vector of residuals:

$$v_k = \left\{ \begin{array}{ll} 0, & \mbox{if } d_k \mbox{ has correct sign} \\ d_k, & \mbox{if } d_k \mbox{ has wrong sign} \end{array} \right.$$

for all $k = 1, \dots, m+n$, determines components of this vector that correspond to largest absolute and relative errors:

$$\texttt{db\_ae\_max} = \max_{1\leq k\leq m+n} |v_k|,$$

$$\texttt{db\_re\_max} = \max_{1\leq k\leq m+n} \frac{|v_k|}{1+|d_k - c_k|},$$

and stores these quantities and corresponding variable indices to the structure `LPXKKT`.

Using the relative errors for all the four conditions the routine `lpx_check_kkt` also estimates a "quality" of the basic solution from the standpoint of these conditions and stores corresponding quality indicators to the structure `LPXKKT`:

`pe_quality` — quality of primal solution;
`pb_quality` — quality of primal feasibility;
`de_quality` — quality of dual solution;
`db_quality` — quality of dual feasibility.

Each of these indicators is assigned to one of the following four values:

`'H'` means high quality,
`'M'` means medium quality,
`'L'` means low quality, or
`'?'` means wrong or infeasible solution.

If all the indicators show high or medium quality (for an internally scaled LP problem, i.e. when the parameter `scaled` in a call to the routine `lpx_check_kkt` is non-zero), the user can be sure that the obtained basic solution is quite accurate.

If some of the indicators show low quality, the solution can still be considered as relevant, though an additional analysis is needed depending on which indicator shows low quality.

If the indicator `pe_quality` is assigned to `'?'`, the primal solution is wrong. If the indicator `de_quality` is assigned to `'?'`, the dual solution is wrong.

If the indicator `db_quality` is assigned to `'?'` while other indicators show a good quality, this means that the current basic solution being primal feasible is not dual feasible. Similarly, if the indicator `pb_quality` is assigned to `'?'` while other indicators are not, this means that the current basic solution being dual feasible is not primal feasible.

2.8 LP basis and simplex table routines

2.8.1 lpx_warm_up — "warm up" LP basis

Synopsis

```
#include "glpk.h"
int lpx_warm_up(LPX *lp);
```

Description The routine `lpx_warm_up` "warms up" the LP basis for the specified problem object using current statuses assigned to rows and columns (i.e. to auxiliary and structural variables).

"Warming up" includes reinverting (factorizing) the basis matrix (if neccesary), computing primal and dual components as well as determining primal and dual statuses of the basic solution.

Returns The routine `lpx_warm_up` returns one of the following exit codes:

`LPX_E_OK`	the LP basis has been successfully "warmed up".
`LPX_E_EMPTY`	the problem has no rows and/or no columns.
`LPX_E_BADB`	the LP basis is invalid, because the number of basic variables is not the same as the number of rows.
`LPX_E_SING`	the basis matrix is numerically singular or ill-conditioned.

2.8.2 lpx_eval_tab_row — compute row of the simplex table

Synopsis

```
#include "glpk.h"
int lpx_eval_tab_row(LPX *lp, int k, int ind[], double val[]);
```

Description The routine `lpx_eval_tab_row` computes a row of the current simplex table for the basic variable, which is specified by the number `k`: if $1 \leq k \leq m$, x_k is k-th auxiliary variable; if $m+1 \leq k \leq m+n$, x_k is $(k-m)$-th structural variable, where m is the number of rows, n is the number of columns. The current basis must be available.

The routine stores column indices and numerical values of non-zero elements of the computed row in sparse format to locations `ind[1]`, ..., `ind[len]` and `val[1]`, ..., `val[len]`, respectively, where $0 \leq$ `len` $\leq n$ is the number of non-zeros returned on exit.

Element indices stored in the array `ind` have the same sense as the index `k`, i.e. indices 1 to m denote auxiliary variables and indices $m+1$ to $m+n$ denote structural ones (all these variables are non-basic by definition).

The computed row shows how the specified basic variable $x_k = (x_B)_i$ depends on non-basic variables:

$$(x_B)_i = \alpha_{i1}(x_N)_1 + \alpha_{i2}(x_N)_2 + \ldots + \alpha_{in}(x_N)_n,$$

where α_{ij} are elements of the simplex table row, $(x_N)_j$ are non-basic (auxiliary and structural) variables.

Returns The routine `lpx_eval_tab_row` returns the number of non-zero elements in the simplex table row stored in the arrays `ind` and `val`.

2.8.3 `lpx_eval_tab_col` — compute column of the simplex table

Synopsis

```
#include "glpk.h"
int lpx_eval_tab_col(LPX *lp, int k, int ind[], double val[]);
```

Description The routine `lpx_eval_tab_col` computes a column of the current simplex table for the non-basic variable, which is specified by the number `k`: if $1 \leq k \leq m$, x_k is k-th auxiliary variable; if $m+1 \leq k \leq m+n$, x_k is $(k-m)$-th structural variable, where m is the number of rows, n is the number of columns. The current basis must be valid.

The routine stores row indices and numerical values of non-zero elements of the computed column in sparse format to locations `ind[1]`, ..., `ind[len]` and `val[1]`, ..., `val[len]`, respectively, where $0 \leq$ `len` $\leq m$ is the number of non-zeros returned on exit.

Element indices stored in the array `ind` have the same sense as the index `k`, i.e. indices 1 to m denote auxiliary variables and indices $m+1$ to $m+n$ denote structural ones (all these variables are basic by definition).

The computed column shows how the basic variables depend on the specified non-basic variable $x_k = (x_N)_j$:

$$\begin{array}{rl} (x_B)_1 & = \ldots + \alpha_{1j}(x_N)_j + \ldots \\ (x_B)_2 & = \ldots + \alpha_{2j}(x_N)_j + \ldots \\ & \ldots\ldots\ldots \\ (x_B)_m & = \ldots + \alpha_{mj}(x_N)_j + \ldots \end{array}$$

where α_{ij} are elements of the simplex table column, $(x_B)_i$ are basic (auxiliary and structural) variables.

Returns The routine `lpx_eval_tab_col` returns the number of non-zero elements in the simplex table column stored in the arrays `ndx` and `val`.

2.8.4 `lpx_transform_row` — transform explicitly specified row

Synopsis

```
#include "glpk.h"
int lpx_transform_row(LPX *lp, int len, int ind[], double val[]);
```

Description The routine `lpx_transform_row` performs the same operation as the routine `lpx_eval_tab_row`, except that the transformed row is specified explicitly.

The explicitly specified row may be thought as a linear form:

$$x = a_1 x_{m+1} + a_2 x_{m+2} + \ldots + a_n x_{m+n}, \tag{1}$$

where x is an auxiliary variable for this row, a_j are coefficients of the linear form, x_{m+j} are structural variables.

On entry column indices and numerical values of non-zero coefficients a_j of the transformed row should be placed in locations ind[1], ..., ind[len] and val[1], ..., val[len], where len is number of non-zero coefficients.

This routine uses the system of equality constraints and the current basis in order to express the auxiliary variable x in (1) through the current non-basic variables (as if the transformed row were added to the problem object and the auxiliary variable x were basic), i.e. the resultant row has the form:

$$x = \alpha_1(x_N)_1 + \alpha_2(x_N)_2 + \ldots + \alpha_n(x_N)_n, \tag{2}$$

where α_j are influence coefficients, $(x_N)_j$ are non-basic (auxiliary and structural) variables, n is number of columns in the specified problem object.

On exit the routine stores indices and numerical values of non-zero coefficients α_j of the resultant row (2) in locations ind[1], ..., ind[len'] and val[1], ..., val[len'], where $0 \leq$ len$' \leq n$ is the number of non-zero coefficients in the resultant row returned by the routine. Note that indices of non-basic variables stored in the array ind correspond to original ordinal numbers of variables: indices 1 to m mean auxiliary variables and indices $m+1$ to $m+n$ mean structural ones.

Returns The routine lpx_transform_row returns len', the number of non-zero coefficients in the resultant row stored in the arrays ind and val.

2.8.5 lpx_transform_col — transform explicitly specified column

Synopsis

```
#include "glpk.h"
int lpx_transform_col(LPX *lp, int len, int ind[], double val[]);
```

Description The routine lpx_transform_col performs the same operation as the routine lpx_eval_tab_col, except that the transformed column is specified explicitly.

The explicitly specified column may be thought as it were added to the original system of equality constraints:

$$\begin{array}{ll} x_1 & = a_{11}x_{m+1} + \ldots + a_{1n}x_{m+n} + a_1 x \\ x_2 & = a_{21}x_{m+1} + \ldots + a_{2n}x_{m+n} + a_2 x \\ & \ldots\ldots\ldots \\ x_m & = a_{m1}x_{m+1} + \ldots + a_{mn}x_{m+n} + a_m x \end{array} \tag{1}$$

where x_i are auxiliary variables, x_{m+j} are structural variables (presented in the problem object), x is a structural variable for the explicitly specified column, a_i are constraint coefficients for x.

On entry row indices and numerical values of non-zero coefficients a_i of the transformed column should be placed in locations ind[1], ..., ind[len] and val[1], ..., val[len], where len is number of non-zero coefficients.

This routine uses the system of equality constraints and the current basis in order to express the current basic variables through the structural variable x in (1) (as if the transformed column were added to the problem object and the variable x were non-basic):

$$
\begin{array}{ll}
(x_B)_1 & = \ldots + \alpha_1 x \\
(x_B)_2 & = \ldots + \alpha_2 x \\
 & \ldots\ldots\ldots \\
(x_B)_m & = \ldots + \alpha_m x
\end{array}
\tag{2}
$$

where α_i are influence coefficients, x_B are basic (auxiliary and structural) variables, m is number of rows in the specified problem object.

On exit the routine stores indices and numerical values of non-zero coefficients α_i of the resultant column (2) in locations `ind[1]`, ..., `ind[len']` and `val[1]`, ..., `val[len']`, where $0 \leq \texttt{len}' \leq m$ is the number of non-zero coefficients in the resultant column returned by the routine. Note that indices of basic variables stored in the array `ind` correspond to original ordinal numbers of variables, i.e. indices 1 to m mean auxiliary variables, indices $m + 1$ to $m + n$ mean structural ones.

Returns The routine `lpx_transform_col` returns `len'`, the number of non-zero coefficients in the resultant column stored in the arrays `ind` and `val`.

2.8.6 `lpx_prim_ratio_test` — perform primal ratio test

Synopsis

```
#include "glpk.h"
int lpx_prim_ratio_test(LPX *lp, int len, int ind[], double val[],
      int how, double tol);
```

Description The routine `lpx_prim_ratio_test` performs the primal ratio test for an explicitly specified column of the simplex table.

The primal basic solution associated with an LP problem object, which the parameter `lp` points to, should be feasible. No components of the LP problem object are changed by the routine.

The explicitly specified column of the simplex table shows how the basic variables x_B depend on some non-basic variable y (which is not necessarily presented in the problem object):

$$
\begin{array}{ll}
(x_B)_1 & = \ldots + \alpha_1 y \\
(x_B)_2 & = \ldots + \alpha_2 y \\
 & \ldots\ldots\ldots \\
(x_B)_m & = \ldots + \alpha_m y
\end{array}
\tag{1}
$$

The column (1) is specifed on entry to the routine using the sparse format. Ordinal numbers of basic variables $(x_B)_i$ should be placed in locations `ind[1]`, ..., `ind[len]`, where ordinal number 1 to m denote auxiliary variables, and ordinal numbers $m + 1$ to $m + n$ denote structural variables. The corresponding non-zero coefficients α_i should be placed in locations `val[1]`, ..., `val[len]`. The arrays `ind` and `val` are not changed by the routine.

The parameter `how` specifies in which direction the variable y changes on entering the basis: $+1$ means increasing, -1 means decreasing.

The parameter `tol` is a relative tolerance (small positive number) used by the routine to skip small α_i in the column (1).

The routine determines the ordinal number of some basic variable (among specified in `ind[1]`, ..., `ind[len]`), which reaches its (lower or upper) bound first before any other basic variables do and which therefore should leave the basis instead the variable y in order to keep primal feasibility, and returns it on exit. If the choice cannot be made (i.e. if the adjacent basic solution is primal unbounded due to y), the routine returns zero.

Note If the non-basic variable y is presented in the LP problem object, the column (1) can be computed using the routine `lpx_eval_tab_col`. Otherwise it can be computed using the routine `lpx_transform_col`.

Returns The routine `lpx_prim_ratio_test` returns the ordinal number of some basic variable $(x_B)_i$, which should leave the basis instead the variable y in order to keep primal feasibility. If the adjacent basic solution is primal unbounded and therefore the choice cannot be made, the routine returns zero.

2.8.7 `lpx_dual_ratio_test` — perform dual ratio test

Synopsis

```
#include "glpk.h"
int lpx_dual_ratio_test(LPX *lp, int len, int ind[], double val[],
      int how, double tol);
```

Description The routine `lpx_dual_ratio_test` performs the dual ratio test for an explicitly specified row of the simplex table.

The dual basic solution associated with an LP problem object, which the parameter `lp` points to, should be feasible. No components of the LP problem object are changed by the routine.

The explicitly specified row of the simplex table is a linear form, which shows how some basic variable y (not necessarily presented in the problem object) depends on non-basic variables x_N:

$$y = \alpha_1 (x_N)_1 + \alpha_2 (x_N)_2 + \ldots + \alpha_n (x_N)_n. \qquad (1)$$

The linear form (1) is specified on entry to the routine using the sparse format. Ordinal numbers of non-basic variables $(x_N)_j$ should be placed in locations `ind[1]`, ..., `ind[len]`, where ordinal numbers 1 to m denote auxiliary variables, and ordinal numbers $m+1$ to $m+n$ denote structural variables. The corresponding non-zero coefficients α_j should be placed in locations `val[1]`, ..., `val[len]`. The arrays `ind` and `val` are not changed by the routine.

The parameter `how` specifies in which direction the variable y changes on leaving the basis: $+1$ means increasing, -1 means decreasing.

The parameter `tol` is a relative tolerance (small positive number) used by the routine to skip small α_j in the form (1).

The routine determines the ordinal number of some non-basic variable (among specified in `ind[1]`, ..., `ind[len]`), whose reduced cost reaches its (zero) bound first before this happens for any other non-basic variables and which therefore should enter the basis instead the variable y in order to keep dual feasibility, and returns it on exit. If the choice cannot be made (i.e. if the adjacent basic solution is dual unbounded due to y), the routine returns zero.

Note If the basic variable y is presented in the LP problem object, the row (1) can be computed using the routine `lpx_eval_tab_row`. Otherwise it can be computed using the routine `lpx_transform_row`.

Returns The routine `lpx_dual_ratio_test` returns the ordinal number of some non-basic variable $(x_N)_j$, which should enter the basis instead the variable y in order to keep dual feasibility. If the adjacent basic solution is dual unbounded and therefore the choice cannot be made, the routine returns zero.

2.9 Interior-point method routines

2.9.1 lpx_interior — solve LP problem using the primal-dual interior-point method

Synopsis

```
#include "glpk.h"
int lpx_interior(LPX *lp);
```

Description The routine `lpx_interior` is an interface to the LP problem solver based on the primal-dual interior-point method.

This routine obtains problem data from the problem object, which the parameter `lp` points to, calls the solver to solve the LP problem, and stores the found solution back in the problem object.

Interior-point methods (also known as barrier methods) are more modern and more powerful numerical methods for large-scale linear programming. They especially fit for very sparse LP problems and allow solving such problems much faster than the simplex method.

Solving large LP problems may take a long time, so the routine `lpx_interior` displays information about every interior point iteration[1]. This information is sent to the standard output and has the following format:

```
nnn: F = fff; rpi = ppp; rdi = ddd; gap = ggg
```

where `nnn` is iteration number, `fff` is the current value of the objective function (in the case of maximization it has wrong sign), `ppp` is the current relative primal infeasibility, `ddd` is the current relative dual infeasibility, and `ggg` is the current primal-dual gap.

Should note that currently the GLPK interior-point solver does not include many important features, in particular:

it is not able to process dense columns. Thus, if the constraint matrix of the LP problem has dense columns, the solving process will be inefficient;

it has no features against numerical instability. For some LP problems premature termination may happen if the matrix ADA^T becomes singular or ill-conditioned;

it is not able to identify the optimal basis, which corresponds to the found interior-point solution.

Returns The routine `lpx_interior` returns one of the following exit codes:

`LPX_E_OK`	the LP problem has been successfully solved (to optimality).
`LPX_E_FAULT`	the solver can't start the search because either the problem has no rows and/or no columns, or some row has non-zero objective coefficient.
`LPX_E_NOFEAS`	the problem has no feasible (primal or dual) solution.

[1]Unlike the simplex method the interior point method usually needs 30—50 iterations (independently on the problem size) in order to find an optimal solution.

`LPX_E_NOCONV`	the search was prematurely terminated due to very slow convergence or divergence.
`LPX_E_ITLIM`	the search was prematurely terminated because the simplex iterations limit has been exceeded.
`LPX_E_INSTAB`	the search was prematurely terminated due to numerical instability on solving Newtonian system.

2.9.2 lpx_ipt_status — retrieve status of interior-point solution

Synopsis

```
#include "glpk.h"
int lpx_ipt_status(LPX *lp);
```

Returns The routine `lpx_ipt_status` reports the status of a solution found by the interior-point solver as follows:

`LPX_T_UNDEF`	interior-point solution is undefined.
`LPX_T_OPT`	interior-point solution is optimal.

2.9.3 lpx_ipt_obj_val — retrieve objective value

Synopsis

```
#include "glpk.h"
double lpx_ipt_obj_val(LPX *lp);
```

Returns The routine `lpx_ipt_obj_val` returns value of the objective function for interior-point solution.

2.9.4 lpx_ipt_row_prim — retrieve row primal value

Synopsis

```
#include "glpk.h"
double lpx_ipt_row_prim(LPX *lp, int i);
```

Returns The routine `lpx_ipt_row_prim` returns primal value of the auxiliary variable associated with `i`-th row.

2.9.5 lpx_ipt_row_dual — retrieve row dual value

Synopsis

```
#include "glpk.h"
double lpx_ipt_row_dual(LPX *lp, int i);
```

Returns The routine lpx_ipt_row_dual returns dual value (i.e. reduced cost) of the auxiliary variable associated with i-th row.

2.9.6 lpx_ipt_col_prim — retrieve column primal value

Synopsis

```
#include "glpk.h"
double lpx_ipt_col_prim(LPX *lp, int j);
```

Returns The routine lpx_ipt_col_prim returns primal value of the structural variable associated with j-th column.

2.9.7 lpx_ipt_col_dual — retrieve column dual value

Synopsis

```
#include "glpk.h"
double lpx_ipt_col_dual(LPX *lp, int j);
```

Returns The routine lpx_ipt_col_dual returns dual value (i.e. reduced cost) of the structural variable associated with j-th column.

2.10 MIP routines

2.10.1 lpx_set_class — set (change) problem class

Synopsis

```
#include "glpk.h"
void lpx_set_class(LPX *lp, int klass);
```

Description The routine `lpx_set_class` sets (changes) the class of the problem object as specified by the parameter `klass`:

`LPX_LP` pure linear programming (LP) problem;
`LPX_MIP` mixed integer programming (MIP) problem.

2.10.2 lpx_get_class — retrieve problem class

Synopsis

```
#include "glpk.h"
int lpx_get_class(LPX *lp);
```

Returns The routine `lpx_get_class` returns the class of the specified problem object:

`LPX_LP` pure linear programming (LP) problem;
`LPX_MIP` mixed integer programming (MIP) problem.

2.10.3 lpx_set_col_kind — set (change) column kind

Synopsis

```
#include "glpk.h"
void lpx_set_col_kind(LPX *lp, int j, int kind);
```

Description The routine `lpx_set_col_kind` sets (changes) the kind of `j`-th column (structural variable) as specified by the parameter `kind`:

`LPX_CV` continuous variable;
`LPX_IV` integer variable.

2.10.4 lpx_get_col_kind — retrieve column kind

Synopsis

```
#include "glpk.h"
int lpx_get_col_kind(LPX *lp, int j);
```

Returns The routine `lpx_get_col_kind` returns the kind of `j`-th column (structural variable) as follows:

`LPX_CV` continuous variable;
`LPX_IV` integer variable.

2.10.5 `lpx_get_num_int` — retrieve number of integer columns

Synopsis

```
#include "glpk.h"
int lpx_get_num_int(LPX *lp);
```

Returns The routine `lpx_get_num_int` returns the number of columns (structural variables), which are marked as integer.

2.10.6 `lpx_get_num_bin` — retrieve number of binary columns

Synopsis

```
#include "glpk.h"
int lpx_get_num_bin(LPX *lp);
```

Returns The routine `lpx_get_num_bin` returns the number of columns (structural variables), which are marked as integer and whose lower bound is zero and upper bound is one.

2.10.7 `lpx_integer` — solve MIP problem using the branch-and-bound method

Synopsis

```
#include "glpk.h"
int lpx_integer(LPX *lp);
```

Description The routine `lpx_integer` is an interface to the MIP problem solver based on the branch-and-bound method.

This routine obtains problem data from the problem object, which the parameter `lp` points to, calls the solver to solve the MIP problem, and stores an obtained solution and other relevant information back in the problem object.

On entry to this routine the problem object must contain an optimal basic solution for LP relaxation, which can be obtained by means of the simplex-based solver (see the routine `lpx_simplex`).

Solving many MIP problems may take a long time, so the solver reports some information about best known solution, which is sent to the standard output. This information has the following format:

```
+nnn: mip = xxx <rho> yyy gap (ppp; qqq)
```

where 'nnn' is the simplex iteration number; 'xxx' is a value of the objective function for the best known integer feasible solution (if no integer feasible solution has been found yet, 'xxx' is the text 'not found yet'); 'rho' is the string '>=' (in case of minimization) or '<=' (in case of maximization); 'yyy' is a global bound for exact integer optimum (i.e. the exact integer optimum is always in the range from 'xxx' to 'yyy'); 'gap' is the relative mip gap, in percents, computed as $gap = |xxx - yyy|/(|xxx| + DBL\_EPSILON) \cdot 100\%$ (if gap is greater than 999.9%, it is not printed); 'ppp' is the number of subproblems in the active list, 'qqq' is the number of subproblems which have been already fathomed and therefore removed from the branch-and-bound search tree.

Note that the branch-and-bound solver implemented in GLPK uses easy heuristics for branching and backtracking, and therefore it is not perfect. Most probably this solver can be used for solving MIP problems with one or two hundreds of integer variables. Hard or very large scale MIP problems cannot be solved by this routine.

Returns The routine `lpx_integer` returns one of the following exit codes:

`LPX_E_OK`	the MIP problem has been successfully solved. (Note that, for example, if the problem has no integer feasible solution, this exit code is reported.)
`LPX_E_FAULT`	unable to start the search because either: the problem is not of MIP class, or the problem object doesn't contain optimal solution for LP relaxation, or some integer variable has non-integer lower or upper bound, or some row has non-zero objective coefficient.
`LPX_E_ITLIM`	the search was prematurely terminated because the simplex iterations limit has been exceeded.
`LPX_E_TMLIM`	the search was prematurely terminated because the time limit has been exceeded.
`LPX_E_SING`	the search was prematurely terminated due to the solver failure (the current basis matrix got singular or ill-conditioned).

2.10.8 `lpx_mip_status` — retrieve status of MIP solution

Synopsis

```
#include "glpk.h"
int lpx_mip_status(LPX *lp);
```

Returns The routine `lpx_mip_status` reports the status of a MIP solution found by the branch-and-bound solver as follows:

`LPX_I_UNDEF`	MIP solution is undefined.
`LPX_I_OPT`	MIP solution is integer optimal.
`LPX_I_FEAS`	MIP solution is integer feasible, however its optimality has not been proven, perhaps due to premature termination of the search.
`LPX_I_NOFEAS`	problem has no integer feasible solution (proven by the solver).

2.10.9 lpx_mip_obj_val — retrieve objective value

Synopsis

```
#include "glpk.h"
double lpx_mip_obj_val(LPX *lp);
```

Returns The routine `lpx_mip_obj_val` returns value of the objective function for MIP solution.

2.10.10 lpx_mip_row_val — retrieve row value

Synopsis

```
#include "glpk.h"
double lpx_mip_row_val(LPX *lp, int i);
```

Returns The routine `lpx_mip_row_val` returns value of the auxiliary variable associated with `i`-th row.

2.10.11 lpx_mip_col_val — retrieve column value

Synopsis

```
#include "glpk.h"
double lpx_mip_col_val(LPX *lp, int j);
```

Returns The routine `lpx_mip_col_val` returns value of the structural variable associated with `j`-th column.

2.11 Control parameters and statistics routines

2.11.1 lpx_reset_parms — reset control parameters to default values

Synopsis

```
#include "glpk.h"
void lpx_reset_parms(LPX *lp);
```

Description The routine `lpx_reset_parms` resets all control parameters associated with a problem object, which the parameter `lp` points to, to their default values.

2.11.2 lpx_set_int_parm — set (change) integer control parameter

Synopsis

```
#include "glpk.h"
void lpx_set_int_parm(LPX *lp, int parm, int val);
```

Description The routine `lpx_set_int_parm` sets (changes) the current value of an integer control parameter `parm`. The parameter `val` specifies a new value of the control parameter.

2.11.3 lpx_get_int_parm — query integer control parameter

Synopsis

```
#include "glpk.h"
int lpx_get_int_parm(LPX *lp, int parm);
```

Returns The routine `lpx_get_int_parm` returns the current value of an integer control parameter `parm`.

2.11.4 lpx_set_real_parm — set (change) real control parameter

Synopsis

```
#include "glpk.h"
void lpx_set_real_parm(LPX *lp, int parm, double val);
```

Description The routine `lpx_set_real_parm` sets (changes) the current value of a real (floating point) control parameter `parm`. The parameter `val` specifies a new value of the control parameter.

2.11.5 lpx_get_real_parm — query real control parameter

Synopsis

```
#include "glpk.h"
double lpx_get_real_parm(LPX *lp, int parm);
```

Returns The routine `lpx_get_real_parm` returns the current value of a real (floating point) control parameter `parm`.

2.11.6 Parameter list

This subsection describes all control parameters currently implemented in the package. Symbolic names of control parameters (which are macros defined in the header file `glpk.h`) are given on the left. Types, default values, and descriptions are given on the right.

`LPX_K_MSGLEV`	type: integer, default: 3 Level of messages output by solver routines: 0 — no output 1 — error messages only 2 — normal output 3 — full output (includes informational messages)
`LPX_K_SCALE`	type: integer, default: 1 Scaling option: 0 — no scaling 1 — equilibration scaling 2 — geometric mean scaling 3 — geometric mean scaling, then equilibration scaling
`LPX_K_DUAL`	type: integer, default: 0 Dual simplex option: 0 — do not use the dual simplex 1 — if initial basic solution is dual feasible, use the dual simplex
`LPX_K_PRICE`	type: integer, default: 1 Pricing option (for both primal and dual simplex): 0 — textbook pricing 1 — steepest edge pricing
`LPX_K_RELAX`	type: real, default: 0.07 Relaxation parameter used in the ratio test. If it is zero, the textbook ratio test is used. If it is non-zero (should be positive), Harris' two-pass ratio test is used. In the latter case on the first pass of the ratio test basic variables (in the case of primal simplex) or reduced costs of non-basic variables (in the case of dual simplex) are allowed to slightly violate their bounds, but not more than (`RELAX` · `TOLBND`) or (`RELAX` · `TOLDJ`) (thus, `RELAX` is a percentage of `TOLBND` or `TOLDJ`).
`LPX_K_TOLBND`	type: real, default: 10^{-7} Relative tolerance used to check if the current basic solution is primal feasible. (Do not change this parameter without detailed understanding its purpose.)

`LPX_K_TOLDJ` type: real, default: 10^{-7}
Absolute tolerance used to check if the current basic solution is dual feasible. (Do not change this parameter without detailed understanding its purpose.)

`LPX_K_TOLPIV` type: real, default: 10^{-9}
Relative tolerance used to choose eligible pivotal elements of the simplex table. (Do not change this parameter without detailed understanding its purpose.)

`LPX_K_ROUND` type: integer, default: 0
Solution rounding option:
0 — report all primal and dual values "as is"
1 — replace tiny primal and dual values by exact zero

`LPX_K_OBJLL` type: real, default: `-DBL_MAX`
Lower limit of the objective function. If on the phase II the objective function reaches this limit and continues decreasing, the solver stops the search. (Used in the dual simplex only.)

`LPX_K_OBJUL` type: real, default: `+DBL_MAX`
Upper limit of the objective function. If on the phase II the objective function reaches this limit and continues increasing, the solver stops the search. (Used in the dual simplex only.)

`LPX_K_ITLIM` type: integer, default: -1
Simplex iterations limit. If this value is positive, it is decreased by one each time when one simplex iteration has been performed, and reaching zero value signals the solver to stop the search. Negative value means no iterations limit.

`LPX_K_ITCNT` type: integer, initial: 0
Simplex iterations count. This count is increased by one each time when one simplex iteration has been performed.

`LPX_K_TMLIM` type: real, default: -1.0
Searching time limit, in seconds. If this value is positive, it is decreased each time when one simplex iteration has been performed by the amount of time spent for the iteration, and reaching zero value signals the solver to stop the search. Negative value means no time limit.

`LPX_K_OUTFRQ` type: integer, default: 200
Output frequency, in iterations. This parameter specifies how frequently the solver sends information about the solution to the standard output.

`LPX_K_OUTDLY` type: real, default: 0.0
Output delay, in seconds. This parameter specifies how long the solver should delay sending information about the solution to the standard output. Non-positive value means no delay.

`LPX_K_BRANCH` type: integer, default: 2
Branching heuristic option (for MIP only):
0 — branch on the first variable
1 — branch on the last variable
2 — branch using a heuristic by Driebeck and Tomlin

`LPX_K_BTRACK`	type: integer, default: 2 Backtracking heuristic option (for MIP only): 0 — depth first search 1 — breadth first search 2 — backtrack using the best projection heuristic
`LPX_K_TOLINT`	type: real, default: 10^{-5} Absolute tolerance used to check if the current basic solution is integer feasible. (Do not change this parameter without detailed understanding its purpose.)
`LPX_K_TOLOBJ`	type: real, default: 10^{-7} Relative tolerance used to check if the value of the objective function is not better than in the best known integer feasible solution. (Do not change this parameter without detailed understanding its purpose.)
`LPX_K_MPSINFO`	type: int, default: 1 If this flag is set, the routine `lpx_write_mps` writes several comment cards, which contains some information about the problem. Otherwise the routine writes no comment cards. This flag also affects the routine `lpx_write_bas`.
`LPX_K_MPSOBJ`	type: int, default: 2 This parameter tells the routine `lpx_write_mps` how to output the objective function row: 0 — never output objective function row 1 — always output objective function row 2 — output objective function row if the problem has no free rows
`LPX_K_MPSORIG`	type: int, default: 0 If this flag is set, the routine `lpx_write_mps` uses the original symbolic names of rows and columns. Otherwise the routine generates plain names using ordinal numbers of rows and columns. This flag also affects the routines `lpx_read_bas` and `lpx_write_bas`.
`LPX_K_MPSWIDE`	type: int, default: 1 If this flag is set, the routine `lpx_write_mps` uses all data fields. Otherwise the routine keeps fields 5 and 6 empty.
`LPX_K_MPSFREE`	type: int, default: 0 If this flag is set, the routine `lpx_write_mps` omits column and vector names every time when possible (free style). Otherwise the routine never omits these names (pedantic style).
`LPX_K_MPSSKIP`	type: int, default: 0 If this flag is set, the routine `lpx_write_mps` skips empty columns (i.e. which has no constraint coefficients). Otherwise the routine outputs all columns.
`LPX_K_PRESOL`	type: int, default: 0 If this flag is set, the routine `lpx_simplex` solves the problem using the built-in LP presolver. Otherwise the LP presolver is not used.

2.12 Utility routines

2.12.1 lpx_read_mps — read problem data in fixed MPS format

Synopsis

```
#include "glpk.h"
LPX *lpx_read_mps(char *fname);
```

Description The routine `lpx_read_mps` reads LP/MIP problem data in fixed MPS format from an input text file whose name is the character string `fname`. (The MPS format is described in Appendix B, page 66.)

Behavior of the routine `lpx_read_mps` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no error occurred, the routine returns a pointer to the created problem object. Otherwise the routine returns `NULL`.

2.12.2 lpx_write_mps — write problem data in fixed MPS format

Synopsis

```
#include "glpk.h"
int lpx_write_mps(LPX *lp, char *fname);
```

Description The routine `lpx_write_mps` writes problem data in fixed MPS format to an output text file whose name is the character string `fname`. (The MPS format is described in Appendix B, page 66.)

Behavior of the routine `lpx_write_mps` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.3 lpx_read_bas — read LP basis in fixed MPS format

Synopsis

```
#include "glpk.h"
int lpx_read_bas(LPX *lp, char *fname);
```

Description The routine `lpx_read_bas` reads LP basis in fixed MPS format from an input text file whose name is the character string `fname`. (About this feature of the MPS format see Section B.13, page 75.)

Behavior of the routine `lpx_read_bas` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.4 lpx_write_bas — write LP basis in fixed MPS format

Synopsis

```
#include "glpk.h"
int lpx_write_bas(LPX *lp, char *fname);
```

Description The routine `lpx_write_bas` writes current LP basis in fixed MPS format to an output text file whose name is the character string `fname`. (About this feature of the MPS format see Section B.13, page 75.)

Behavior of the routine `lpx_write_bas` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.5 lpx_read_freemps — read problem data in free MPS format

Synopsis

```
#include "glpk.h"
LPX *lpx_read_freemps(char *fname);
```

Description The routine `lpx_read_freemps` reads LP/MIP problem data in free MPS format from an input text file whose name is the character string `fname`. (The MPS format is described in Appendix B, page 66.)

Behavior of the routine `lpx_read_freemps` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no error occurred, the routine returns a pointer to the created problem object. Otherwise the routine returns `NULL`.

2.12.6 lpx_write_freemps — write problem data in free MPS format

Synopsis

```
#include "glpk.h"
int lpx_write_freemps(LPX *lp, char *fname);
```

Description The routine `lpx_write_freemps` writes problem data in fixed MPS format to an output text file whose name is the character string `fname`. (The MPS format is described in Appendix B, page 66.)

Behavior of the routine `lpx_write_freemps` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.7 lpx_read_cpxlp — read problem data in CPLEX LP format

Synopsis

```
#include "glpk.h"
LPX *lpx_read_cpxlp(char *fname);
```

Description The routine `lpx_read_cpxlp` reads LP/MIP problem data in CPLEX LP format from an input text file whose name is the character string `fname`. (The CPLEX LP format is described in Appendix C, page 77.)

Returns Behavior of the routine `lpx_read_cpxlp` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no error occurred, the routine returns a pointer to the created problem object. Otherwise the routine returns `NULL`.

2.12.8 lpx_write_cpxlp — write problem data in CPLEX LP format

Synopsis

```
#include "glpk.h"
int lpx_write_cpxlp(LPX *lp, char *fname);
```

Description The routine `lpx_write_cpxlp` writes problem data in CPLEX LP format to an output text file whose name is the character string `fname`. (The CPLEX LP format is described in Appendix C, page 77.)

Behavior of the routine `lpx_write_cpxlp` depends on some control parameters (see Subsection 2.11.6, page 54.)

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.9 lpx_read_model — read model written in GNU MathProg modeling language

Synopsis

```
#include "glpk.h"
LPX *lpx_read_model(char *model, char *data, char *output);
```

Description The routine lpx_read_model reads and translates LP/MIP model (problem) written in the GNU MathProg modeling language.[2]

The character string model specifies name of input text file, which contains model section and, optionally, data section. This parameter cannot be NULL.

The character string data specifies name of input text file, which contains data section. This parameter can be NULL. (If the data file is specified and the model file also contains data section, that section is ignored and data section from the data file is used.)

The character string output specifies name of output text file, to which the output produced by display statements is written. If the parameter output is NULL, the display output is sent to stdout via the routine print.

The routine lpx_read_model is an interface to the model translator, which is a program that parses model description and translates it to some internal data structures.

For detailed description of the modeling language see the document "GLPK: Modeling Language GNU MathProg" included in the GLPK distribution.

Returns If no errors occurred, the routine returns a pointer to the created problem object. Otherwise the routine sends diagnostics to the standard output and returns NULL.

2.12.10 lpx_print_prob — write problem data in plain text format

Synopsis

```
#include "glpk.h"
int lpx_print_prob(LPX *lp, char *fname);
```

Description The routine lpx_print_prob writes data from a problem object, which the parameter lp points to, to an output text file, whose name is the character string fname, in plain text format.

Information reported by the routine lpx_print_prob is intended mainly for visual analysis.

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.11 lpx_print_sol — write basic solution in printable format

Synopsis

```
#include "glpk.h"
int lpx_print_sol(LPX *lp, char *fname);
```

Description The routine lpx_print_sol writes the current basic solution of an LP problem, which is specified by the pointer lp, to a text file, whose name is the character string fname, in printable format.

Information reported by the routine lpx_print_sol is intended mainly for visual analysis.

[2]The GNU MathProg modeling language is a subset of the AMPL language.

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.12 lpx_print_sens_bnds — write bounds sensitivity information

Synopsis

```
#include "glpk.h"
int lpx_print_sens_bnds(LPX *lp, char *fname);
```

Description The routine `lpx_print_sens_bnds` writes the bounds for objective coefficients, right-hand-sides of constraints, and variable bounds for which the current optimal basic solution remains optimal (for LP only).

The LP is given by the pointer `lp`, and the output is written to the file specified by `fname`. The current contents of the file will be overwritten.

Information reported by the routine `lpx_print_sens_bnds` is intended mainly for visual analysis.

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.13 lpx_print_ips — write interior point solution in printable format

Synopsis

```
#include "glpk.h"
int lpx_print_ips(LPX *lp, char *fname);
```

Description The routine `lpx_print_ips` writes the current interior point solution of an LP problem, which the parameter `lp` points to, to a text file, whose name is the character string `fname`, in printable format.

Information reported by the routine `lpx_print_ips` is intended mainly for visual analysis.

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

2.12.14 lpx_print_mip — write MIP solution in printable format

Synopsis

```
#include "glpk.h"
int lpx_print_mip(LPX *lp, char *fname);
```

Description The routine lpx_print_mip writes a best known integer solution of a MIP problem, which is specified by the pointer lp, to a text file, whose name is the character string fname, in printable format.

Information reported by the routine lpx_print_mip is intended mainly for visual analysis.

Returns If no errors occurred, the routine returns zero. Otherwise the routine prints an error message and returns non-zero.

Appendix A

Installing GLPK on Your Computer

A.1 Obtaining GLPK distribution file

The distrubution file for the most recent version of the GLPK package can be downloaded from <ftp://ftp.gnu.org/gnu/glpk/> or from some mirror GNU ftp sites; for details see <http://www.gnu.org/order/ftp.html>.

A.2 Unpacking the distribution file

The GLPK package (like all other GNU software) is distributed in the form of packed archive. This is one file named `glpk-x.y.tar.gz`, where x is the major version number and y is the minor version number.

In order to prepare the distribution for installation you should:

1. Copy the GLPK distribution file to some subdirectory.

2. Enter the command `gzip -d glpk-x.y.tar.gz` in order to unpack the distribution file. After unpacking the name of the distribution file will be automatically changed to `glpk-x.y.tar`.

3. Enter the command `tar -x < glpk-x.y.tar` in order to unarchive the distribution. After this operation the subdirectory `glpk-x.y`, which is the GLPK distribution, will be automatically created.

A.3 Configuring the package

After you have unpacked and unarchived GLPK distribution you should configure the package, i.e. automatically tune it for your computer (platform).

Normally, you should just `cd` to the subdirectory `glpk-x.y` and enter the command `./configure`. If you are using `csh` on an old version of System V, you might need to type `sh configure` instead to prevent `csh` from trying execute `configure` itself.

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation, and creates `Makefile`. It also creates a file `config.status` that you can run in the future to recreate the current configuration.

Running `configure` takes about a few minutes. While it is running, it displays some informational messages that tell you what it is doing. If you don't want to see these messages, run `configure` with its standard output redirected to `dev/null`; for example, `./configure >/dev/null`.

A.4 Compiling and checking the package

Normally, in order to compile the package you should just enter the command `make`. This command reads `Makefile` generated by `configure` and automatically performs all necessary job.

The result of compilation is:

• the file `libglpk.a`, which is a library archive that contains object code for all GLPK routines; and

• the program `glpsol`, which is a stand-alone LP/MIP solver.

If you want, you can override the `make` variables `CFLAGS` and `LDFLAGS` like this:

```
make CFLAGS=-O2 LDFLAGS=-s
```

To compile the package in a different directory from the one containing the source code, you must use a version of `make` that supports `VPATH` variable, such as GNU make. `cd` to the directory where you want the object files and executables to go and run the `configure` script. `configure` automatically checks for the source code in the directory that `configure` is in and in '`..`'. If for some reason `configure` is not in the source code directory that you are configuring, then it will report that it can't find the source code. In that case, run `configure` with the option `--srcdir=DIR`, where `DIR` is the directory that contains the source code.

On systems that require unusual options for compilation or linking the package's `configure` script does not know about, you can give `configure` initial values for variables by setting them in the environment. In Bourne-compatible shells you can do that on the command line like this:

```
CC='gcc -traditional' LIBS=-lposix ./configure
```

Here are the `make` variables that you might want to override with environment variables when running `configure`.

For these variables, any value given in the environment overrides the value that `configure` would choose:

• variable `CC`: C compiler program. The default is `cc`.

• variable `INSTALL`: program to use to install files. The default value is `install` if you have it, otherwise `cp`.

For these variables, any value given in the environment is added to the value that `configure` chooses:

• variable DEFS: configuration options, in the form '`-Dfoo -Dbar` ...'.

• variable LIBS: libraries to link with, in the form '`-lfoo -lbar` ...'.

In order to check the package (running some tests included in the distribution) you can just enter the command `make check`.

A.5 Installing the package

Normally, in order to install the GLPK package (i.e. copy GLPK library, header files, and the solver to the system places) you should just enter the command `make install` (note that you should be the root user or a superuser).

By default, `make install` will install the package's files in the subdirectories `usr/local/bin`, `usr/local/lib`, etc. You can specify an installation prefix other than `/usr/local` by giving `configure` the option `--prefix=PATH`. Alternately, you can do so by consistently giving a value for the `prefix` variable when you run `make`, e.g.

```
make prefix=/usr/gnu
make prefix=/usr/gnu install
```

After installing you can remove the program binaries and object files from the source directory by typing `make clean`. To remove all files that `configure` created (`Makefile`, `config.status`, etc.), just type `make distclean`.

The file `configure.in` is used to create `configure` by a program called `autoconf`. You only need it if you want to remake `configure` using a newer version of `autoconf`.

A.6 Uninstalling the package

In order to uninstall the GLPK package (i.e. delete all GLPK files from the system places) you can enter the command `make uninstall`.

Appendix B

MPS Format

B.1 Fixed MPS Format

The MPS format[1] is intended for coding LP/MIP problem data. This format assumes the formulation of LP/MIP problem (1.1)—(1.3) (see Section 1.1, page 7).

MPS file is a text file, which contains two types of cards[2]: indicator cards and data cards.

Indicator cards determine a kind of succeeding data. Each indicator card has one word in uppercase letters beginning in column 1.

Data cards contain problem data. Each data card is divided into six fixed fields:

	Field 1	Field 2	Field 3	Field 4	Field 5	Feld 6
Columns	2—3	5—12	15—22	25—36	40—47	50—61
Contents	Code	Name	Name	Number	Name	Number

On a particular data card some fields may be optional.

Names are used to identify rows, columns, and some vectors (see below).

Aligning the indicator code in the field 1 to the left margin is optional.

All names specified in the fields 2, 3, and 5 should contain from 1 up to 8 arbitrary characters (except control characters). If a name is placed in the field 3 or 5, its first character should not be the dollar sign '`$`'. If a name contains spaces, the spaces are ignored.

All numerical values in the fields 4 and 6 should be coded in the form sxxEsyy, where s is the plus '`+`' or the minus '`-`' sign, xx is a real number with optional decimal point, yy is an integer decimal exponent. Any number should contain up to 12 characters. If the sign s is omitted, the plus sign is assumed. The exponent part is optional. If a number contains spaces, the spaces are ignored.

If a card has the asterisk '`*`' in the column 1, this card is considered as a comment and ignored. Besides, if the first character in the field 3 or 5 is the dollar sign '`$`', all characters from the dollar sign to the end of card are considered as a comment and ignored.

[1]The MPS format was developed in 1960's by IBM as input format for their mathematical programming system MPS/360. Today the MPS format is a most widely used format understood by most mathematical programming packages. This appendix describes only the features of the MPS format, which are implemented in the GLPK package.

[2]In 1960's MPS file was a deck of 80-column punched cards, so the author decided to keep the word "card", which may be understood as "line of text file".

MPS file should contain cards in the following order:
- NAME indicator card;
- ROWS indicator card;
- data cards specifying rows (constraints);
- COLUMNS indicator card;
- data cards specifying columns (structural variables) and constraint coefficients;
- RHS indicator card;
- data cards specifying right-hand sides of constraints;
- RANGES indicator card;
- data cards specifying ranges for double-bounded constraints;
- BOUNDS indicator card;
- data cards specifying types and bounds of structural variables;
- ENDATA indicator card.

Section is a group of cards consisting of an indicator card and data cards succeeding this indicator card. For example, the ROWS section consists of the ROWS indicator card and data cards specifying rows.

The sections RHS, RANGES, and BOUNDS are optional and may be omitted.

B.2 Free MPS Format

Free MPS format is an improved version of the standard (fixed) MPS format described above.[3] Note that all changes in free MPS format concern only the coding of data while the structure of data is the same for both fixed and free versions of the MPS format.

In free MPS format indicator and data records[4] may have arbitrary length not limited to 80 characters. Fields of data records have no predefined positions, i.e. the fields may begin in any position, except position 1, which must be blank, and must be separated from each other by one or more blanks. However, the fields must appear in the same order as in fixed MPS format.

Symbolic names in fields 2, 3, and 5 may be longer than 8 characters[5] and must not contain embedded blanks.

Numeric values in fields 4 and 6 are limited to 12 characters and must not contain embedded blanks.

Only six fields on each data record are used. Any other fields are ignored.

If the first character of any field (not necessarily fields 3 and 5) is the dollar sign ($), all characters from the dollar sign to the end of record are considered as a comment and ignored.

B.3 NAME indicator card

The NAME indicator card should be the first card in the MPS file (except optional comment cards, which may precede the NAME card). This card should contain the word `NAME` in the columns 1—4 and the problem name in the field 3. The problem name is optional and may be omitted.

[3] This format was developed in the beginning of 1990's by IBM as an alternative to the standard fixed MPS format for Optimization Subroutine Library (OSL).

[4] *Record* in free MPS format has the same meaning as *card* in fixed MPS format.

[5] GLPK allows symbolic names having up to 255 characters.

B.4 ROWS section

The ROWS section should start with the indicator card, which contains the word `ROWS` in the columns 1—4.

Each data card in the ROWS section specifies one row (constraint) of the problem. All these data cards have the following format.

'`N`' in the field 1 means that the row is free (unbounded):

$$-\infty < x_i = a_{i1}x_{m+1} + a_{i2}x_{m+2} + \ldots + a_{in}x_{m+n} < +\infty;$$

'`L`' in the field 1 means that the row is of "less than or equal to" type:

$$-\infty < x_i = a_{i1}x_{m+1} + a_{i2}x_{m+2} + \ldots + a_{in}x_{m+n} \leq b_i;$$

'`G`' in the field 1 means that the row is of "greater than or equal to" type:

$$b_i \leq x_i = a_{i1}x_{m+1} + a_{i2}x_{m+2} + \ldots + a_{in}x_{m+n} < +\infty;$$

'`E`' in the field 1 means that the row is of "equal to" type:

$$x_i = a_{i1}x_{m+1} + a_{i2}x_{m+2} + \ldots + a_{in}x_{m+n} \leq b_i,$$

where b_i is a right-hand side. Note that each constraint has a corresponding implictly defined auxiliary variable (x_i above), whose value is a value of the corresponding linear form, therefore row bounds can be considered as bounds of such auxiliary variable.

The filed 2 specifies a row name (which is considered as the name of the corresponding auxiliary variable).

The fields 3, 4, 5, and 6 are not used and should be empty.

Numerical values of all non-zero right-hand sides b_i should be specified in the RHS section (see below). All double-bounded (ranged) constraints should be specified in the RANGES section (see below).

B.5 COLUMNS section

The COLUMNS section should start with the indicator card, which contains the word `COLUMNS` in the columns 1—7.

Each data card in the COLUMNS section specifies one or two constraint coefficients a_{ij} and also introduces names of columns, i.e. names of structural variables. All these data cards have the following format.

The field 1 is not used and should be empty.

The field 2 specifies a column name. If this field is empty, the column name from the immediately preceeding data card is assumed.

The field 3 specifies a row name defined in the ROWS section.

The field 4 specifies a numerical value of the constraint coefficient a_{ij}, which is placed in the corresponding row and column.

The fields 5 and 6 are optional. If they are used, they should contain a second pair "row name—constraint coefficient" for the same column.

Elements of the constraint matrix (i.e. constraint coefficients) should be enumerated in the column wise manner: all elements for the current column should be specified before elements for the next column. However, the order of rows in the COLUMNS section may differ from the order of rows in the ROWS section.

Constraint coefficients not specified in the COLUMNS section are considered as zeros. Therefore zero coefficients may be omitted, although it is allowed to explicitly specify them.

B.6 RHS section

The RHS section should start with the indicator card, which contains the word `RHS` in the columns 1—3.

Each data card in the RHS section specifies one or two right-hand sides b_i (see Section B.4, page 68). All these data cards have the following format.

The field 1 is not used and should be empty.

The field 2 specifies a name of the right-hand side (RHS) vector[6]. If this field is empty, the RHS vector name from the immediately preceeding data card is assumed.

The field 3 specifies a row name defined in the ROWS section.

The field 4 specifies a right-hand side b_i for the row, whose name is specified in the field 3. Depending on the row type b_i is a lower bound (for the row of `G` type), an upper bound (for the row of `L` type), or a fixed value (for the row of `E` type).[7]

The fields 5 and 6 are optional. If they are used, they should contain a second pair "row name—right-hand side" for the same RHS vector.

All right-hand sides for the current RHS vector should be specified before right-hand sides for the next RHS vector. However, the order of rows in the RHS section may differ from the order of rows in the ROWS section.

Right-hand sides not specified in the RHS section are considered as zeros. Therefore zero right-hand sides may be omitted, although it is allowed to explicitly specify them.

B.7 RANGES section

The RANGES section should start with the indicator card, which contains the word `RANGES` in the columns 1—6.

Each data card in the RANGES section specifies one or two ranges for double-side constraints, i.e. for constraints that are of the types `L` and `G` at the same time:

$$l_i \leq x_i = a_{i1}x_{m+1} + a_{i2}x_{m+2} + \ldots + a_{in}x_{m+n} \leq u_i,$$

where l_i is a lower bound, u_i is an upper bound. All these data cards have the following format.

The field 1 is not used and should be empty.

The field 2 specifies a name of the range vector[8]. If this field is empty, the range vector name from the immediately preceeding data card is assumed.

The field 3 specifies a row name defined in the ROWS section.

[6]This feature allows the user to specify several RHS vectors in the same MPS file. However, before solving the problem a particular RHS vector should be chosen.

[7]If the row is of `N` type, b_i is considered as a constant term of the corresponding linear form. Should note, however, this convention is non-standard.

[8]This feature allows the user to specify several range vectors in the same MPS file. However, before solving the problem a particular range vector should be chosen.

The field 4 specifies a range value r_i (see the table below) for the row, whose name is specified in the field 3.

The fields 5 and 6 are optional. If they are used, they should contain a second pair "row name—range value" for the same range vector.

All range values for the current range vector should be specified before range values for the next range vector. However, the order of rows in the RANGES section may differ from the order of rows in the ROWS section.

For each double-side constraint specified in the RANGES section its lower and upper bounds are determined as follows:

Row type	Sign of r_i	Lower bound	Upper bound
`G`	$+$ or $-$	b_i	$b_i + \lvert r_i \rvert$
`L`	$+$ or $-$	$b_i - \lvert r_i \rvert$	b_i
`E`	$+$	b_i	$b_i + \lvert r_i \rvert$
`E`	$-$	$b_i - \lvert r_i \rvert$	b_i

where b_i is a right-hand side specified in the RHS section (if b_i is not specified, it is considered as zero), r_i is a range value specified in the RANGES section.

B.8 BOUNDS section

The BOUNDS section should start with the indicator card, which contains the word `BOUNDS` in the columns 1—6.

Each data card in the BOUNDS section specifies one (lower or upper) bound for one structural variable (column). All these data cards have the following format.

The indicator in the field 1 specifies the bound type:

`LO` lower bound;
`UP` upper bound;
`FX` fixed variable (lower and upper bounds are equal);
`FR` free variable (no bounds);
`MI` no lower bound (lower bound is "minus infinity");
`PL` no upper bound (upper bound is "plus infinity");

The field 2 specifies a name of the bound vector[9]. If this field is empty, the bound vector name from the immediately preceeding data card is assumed.

The field 3 specifies a column name defined in the COLUMNS section.

The field 4 specifies a bound value. If the bound type in the field 1 differs from `LO`, `UP`, and `FX`, the value in the field 4 is ignored and may be omitted.

The fields 5 and 6 are not used and should be empty.

All bound values for the current bound vector should be specified before bound values for the next bound vector. However, the order of columns in the BOUNDS section may differ from the order of columns in the COLUMNS section. Specification of a lower bound should precede specification of an upper bound for the same column (if both the lower and upper bounds are explicitly specified).

By default, all columns (structural variables) are non-negative, i.e. have zero lower bound and no upper bound. Lower (l_j) and upper (u_j) bounds of some column (structural

[9]This feature allows the user to specify several bound vectors in the same MPS file. However, before solving the problem a particular bound vector should be chosen.

variable x_j) are set in the following way, where s_j is a corresponding bound value explicitly specified in the BOUNDS section:

`LO` sets l_j to s_j;
`UP` sets u_j to s_j;
`FX` sets both l_j and u_j to s_j;
`FR` sets l_j to $-\infty$ and u_j to $+\infty$;
`MI` sets l_j to $-\infty$;
`PL` sets u_j to $+\infty$.

B.9 ENDATA indicator card

The ENDATA indicator card should be the last card of MPS file (except optional comment cards, which may follow the ENDATA card). This card should contain the word `ENDATA` in the columns 1—6.

B.10 Specifying objective function

It is impossible to explicitly specify the objective function and optimization direction in the MPS file. However, the following implicit rule is used by default: the first row of `N` type is considered as a row of the objective function (i.e. the objective function is the corresponding auxiliary variable), which should be *minimized.*

GLPK also allows specifying a constant term of the objective function as a right-hand side of the corresponding row in the RHS section.

B.11 Example of MPS file

In order to illustrate what the MPS format is, consider the following example of LP problem:

minimize

$$value = .03\ bin_1 + .08\ bin_2 + .17\ bin_3 + .12\ bin_4 + .15\ bin_5 + .21\ alum + .38\ silicon$$

subject to linear constraints

$$\begin{array}{lllllllll}
yield & = & bin_1 + & bin_2 + & bin_3 + & bin_4 + & bin_5 + & alum + & silicon \\
fe & = & .15\ bin_1 + & .04\ bin_2 + & .02\ bin_3 + & .04\ bin_4 + & .02\ bin_5 + & .01\ alum + & .03\ silicon \\
cu & = & .03\ bin_1 + & .05\ bin_2 + & .08\ bin_3 + & .02\ bin_4 + & .06\ bin_5 + & .01\ alum & \\
mn & = & .02\ bin_1 + & .04\ bin_2 + & .01\ bin_3 + & .02\ bin_4 + & .02\ bin_5 & & \\
mg & = & .02\ bin_1 + & .03\ bin_2 & & + & .01\ bin_5 & & \\
al & = & .70\ bin_1 + & .75\ bin_2 + & .80\ bin_3 + & .75\ bin_4 + & .80\ bin_5 + & .97\ alum & \\
si & = & .02\ bin_1 + & .06\ bin_2 + & .08\ bin_3 + & .12\ bin_4 + & .02\ bin_5 + & .01\ alum + & .97\ silicon
\end{array}$$

and bounds of (auxiliary and structural) variables

$$\begin{array}{rclrcl}
yield &=& 2000 & 0 \leq bin_1 &\leq& 200 \\
-\infty < fe &\leq& 60 & 0 \leq bin_2 &\leq& 2500 \\
-\infty < cu &\leq& 100 & 400 \leq bin_3 &\leq& 800 \\
-\infty < mn &\leq& 40 & 100 \leq bin_4 &\leq& 700 \\
-\infty < mg &\leq& 30 & 0 \leq bin_5 &\leq& 1500 \\
1500 \leq al &<& +\infty & 0 \leq alum &<& +\infty \\
250 \leq si &\leq& 300 & 0 \leq silicon &<& +\infty
\end{array}$$

A complete MPS file which specifies data for this example is shown below (the first two comment lines show card positions).

```
*000000001111111111222222222233333333334444444444555555555566
*234567890123456789012345678901234567890123456789012345678901
NAME          PLAN
ROWS
 N  VALUE
 E  YIELD
 L  FE
 L  CU
 L  MN
 L  MG
 G  AL
 L  SI
COLUMNS
    BIN1      VALUE          .03000   YIELD          1.00000
              FE             .15000   CU              .03000
              MN             .02000   MG              .02000
              AL             .70000   SI              .02000
    BIN2      VALUE          .08000   YIELD          1.00000
              FE             .04000   CU              .05000
              MN             .04000   MG              .03000
              AL             .75000   SI              .06000
    BIN3      VALUE          .17000   YIELD          1.00000
              FE             .02000   CU              .08000
              MN             .01000   AL              .80000
              SI             .08000
    BIN4      VALUE          .12000   YIELD          1.00000
              FE             .04000   CU              .02000
              MN             .02000   AL              .75000
              SI             .12000
    BIN5      VALUE          .15000   YIELD          1.00000
              FE             .02000   CU              .06000
              MN             .02000   MG              .01000
              AL             .80000   SI              .02000
    ALUM      VALUE          .21000   YIELD          1.00000
              FE             .01000   CU              .01000
              AL             .97000   SI              .01000
```

```
    SILICON   VALUE           .38000   YIELD          1.00000
              FE              .03000   SI              .97000
RHS
    RHS1      YIELD       2000.00000   FE            60.00000
              CU           100.00000   MN            40.00000
              SI           300.00000
              MG            30.00000   AL          1500.00000
RANGES
    RNG1      SI            50.00000
BOUNDS
 UP BND1      BIN1         200.00000
 UP           BIN2        2500.00000
 LO           BIN3         400.00000
 UP           BIN3         800.00000
 LO           BIN4         100.00000
 UP           BIN4         700.00000
 UP           BIN5        1500.00000
ENDATA
```

B.12 MIP features

The MPS format provides two ways for introducing integer variables into the problem.

The first way is most general and based on using special marker cards INTORG and INTEND. These marker cards are placed in the COLUMNS section. The INTORG card indicates the start of a group of integer variables (columns), and the card INTEND indicates the end of the group. The MPS file may contain arbitrary number of the marker cards.

The marker cards have the same format as the data cards (see Section B.1, page 66).

The fields 1, 2, and 6 are not used and should be empty.

The field 2 should contain a marker name. This name may be arbitrary.

The field 3 should contain the word `'MARKER'` (including apostrophes).

The field 5 should contain either the word `'INTORG'` (including apostrophes) for the marker card, which begins a group of integer columns, or the word `'INTEND'` (including apostrophes) for the marker card, which ends the group.

The second way is less general but more convenient in some cases. It allows the user to declare integer columns using two additional types of bounds, which are specified in the field 1 of data cards in the BOUNDS section (see Section B.8, page 70):

`UI` upper integer. This bound type specifies that the corresponding column (structural variable), whose name is specified in the field 3, is of integer kind. In this case an upper bound of the column should be specified in the field 4 (like in the case of `UP` bound type).

`BV` binary variable. This bound type specifies that the corresponding column (structural variable), whose name is specified in the field 3, is of integer kind, its lower bound is zero, and its upper bound is one (thus, such variable being of integer kind can have only two values zero and one). In this case a numeric value specified in the field 4 is ignored and may be omitted.

Consider the following example of MIP problem:

minimize

$$Z = 3x_1 + 7x_2 - x_3 + x4$$

subject to linear constraints

$$\begin{aligned} r_1 &= 2x_1 - x_2 + x_3 - x_4 \\ r_2 &= x_1 - x_2 - 6x_3 + 4x_4 \\ r_3 &= 5x_1 + 3x_2 + x_4 \end{aligned}$$

and bound of variables

$$\begin{array}{lll} 1 \leq r_1 < +\infty & 0 \leq x_1 \leq 4 & \text{(continuous)} \\ 8 \leq r_2 < +\infty & 2 \leq x_2 \leq 5 & \text{(integer)} \\ 5 \leq r_3 < +\infty & 0 \leq x_3 \leq 1 & \text{(integer)} \\ & 3 \leq x_4 \leq 8 & \text{(continuous)} \end{array}$$

The corresponding MPS file may look like the following:

```
NAME          SAMP1
ROWS
 N  Z
 G  R1
 G  R2
 G  R3
COLUMNS
    X1        R1                 2.0    R2                 1.0
    X1        R3                 5.0    Z                  3.0
    MARK0001  'MARKER'                  'INTORG'
    X2        R1                -1.0    R2                -1.0
    X2        R3                 3.0    Z                  7.0
    X3        R1                 1.0    R2                -6.0
    X3        Z                 -1.0
    MARK0002  'MARKER'                  'INTEND'
    X4        R1                -1.0    R2                 4.0
    X4        R3                 1.0    Z                  1.0
RHS
    RHS1      R1                 1.0
    RHS1      R2                 8.0
    RHS1      R3                 5.0
BOUNDS
 UP BND1      X1                 4.0
 LO BND1      X2                 2.0
 UP BND1      X2                 5.0
 UP BND1      X3                 1.0
 LO BND1      X4                 3.0
 UP BND1      X4                 8.0
ENDATA
```

The same example may be coded without INTORG/INTEND markers using the bound type UI for the variable x_2 and the bound type BV for the variable x_3:

```
NAME          SAMP2
ROWS
 N  Z
 G  R1
 G  R2
 G  R3
COLUMNS
    X1        R1                 2.0    R2                   1.0
    X1        R3                 5.0    Z                    3.0
    X2        R1                -1.0    R2                  -1.0
    X2        R3                 3.0    Z                    7.0
    X3        R1                 1.0    R2                  -6.0
    X3        Z                 -1.0
    X4        R1                -1.0    R2                   4.0
    X4        R3                 1.0    Z                    1.0
RHS
    RHS1      R1                 1.0
    RHS1      R2                 8.0
    RHS1      R3                 5.0
BOUNDS
 UP BND1      X1                 4.0
 LO BND1      X2                 2.0
 UI BND1      X2                 5.0
 BV BND1      X3
 LO BND1      X4                 3.0
 UP BND1      X4                 8.0
ENDATA
```

B.13 Specifying predefined basis

The MPS format can also be used to specify some predefined basis for an LP problem, i.e. to specify which rows and columns are basic and which are non-basic.

The order of a basis file in the MPS format is:

- NAME indicator card;
- data cards (can appear in arbitrary order);
- ENDATA indicator card.

Each data card specifies either a pair "basic column—non-basic row" or a non-basic column. All the data cards have the following format.

'`XL`' in the field 1 means that a column, whose name is given in the field 2, is basic, and a row, whose name is given in the field 3, is non-basic and placed on its lower bound.

'`XU`' in the field 1 means that a column, whose name is given in the field 2, is basic, and a row, whose name is given in the field 3, is non-basic and placed on its upper bound.

'`LL`' in the field 1 means that a column, whose name is given in the field 3, is non-basic and placed on its lower bound.

'`UL`' in the field 1 means that a column, whose name is given in the field 3, is non-basic and placed on its upper bound.

The field 2 contains a column name.

If the indicator given in the field 1 is '`XL`' or '`XU`', the field 3 contains a row name. Otherwise, if the indicator is '`LL`' or '`UL`', the field 3 is not used and should be empty.

The field 4, 5, and 6 are not used and should be empty.

A basis file in the MPS format acts like a patch: it doesn't specify a basis completely, instead that it is just shows in what a given basis differs from the "standard" basis, where all rows (auxiliary variables) are assumed to be basic and all columns (structural variables) are assumed to be non-basic.

As an example here is a basis file that specifies an optimal basis for the example LP problem given in Section B.11, Page 71:

```
*000000001111111111222222222233333333334444444444555555555566
*234567890123456789012345678901234567890123456789012345678901
NAME          PLAN
 XL BIN2      YIELD
 XL BIN3      FE
 XL BIN4      MN
 XL ALUM      AL
 XL SILICON   SI
 LL BIN1
 LL BIN5
ENDATA
```

Appendix C

CPLEX LP Format

C.1 Prelude

The CPLEX LP format[1] is intended for coding LP/MIP problem data. It is a row-oriented format that assumes the formulation of LP/MIP problem (1.1)—(1.3) (see Section 1.1, page 7).

CPLEX LP file is a plain text file written in CPLEX LP format. Each text line of this file may contain up to 255 characters[2]. Blank lines are ignored. If a line contains the backslash character (\), this character and everything that follows it until the end of line are considered as a comment and also ignored.

An LP file is coded by the user using the following elements:

- keywords;
- symbolic names;
- numeric constants;
- delimiters;
- blanks.

Keywords that may be used in the LP file are the following:

```
minimize        minimum        min
maximize        maximum        max
subject to      such that      s.t.      st.      st
bounds          bound
general         generals       gen
integer         integers       int
binary          binaries       bin
infinity        inf
free
end
```

All the keywords are case insensitive. Keywords given above on the same line are equivalent. Any keyword (except `infinity`, `inf`, and `free`) being used in the LP file must start

[1] The CPLEX LP format was developed in the end of 1980's by CPLEX Optimization, Inc. as an input format for the CPLEX linear programming system. Although the CPLEX LP format is not as widely used as the MPS format, being row-oriented it is more convenient for coding mathematical programming models by human. This appendix describes only the features of the CPLEX LP format which are implemented in the GLPK package.

[2] GLPK allows text lines of arbitrary length.

at the beginning of a text line.

Symbolic names are used to identify the objective function, constraints (rows), and variables (columns). All symbolic names are case sensitive and may contain up to 16 alphanumeric characters[3] (`a`, ..., `z`, `A`, ..., `Z`, `0`, ..., `9`) as well as the following characters:

```
! " # $ % & ( ) / , . ; ? @ _ ‘ ’ { } | ~
```

with exception that no symbolic name can begin with a digit or a period.

Numeric constants are used to denote constraint and objective coefficients, right-hand sides of constraints, and bounds of variables. They are coded in the standard form $xx\texttt{E}syy$, where xx is a real number with optional decimal point, s is a sign (`+` or `-`), yy is an integer decimal exponent. Numeric constants may contain arbitrary number of characters. The exponent part is optional. The letter '`E`' can be coded as '`e`'. If the sign s is omitted, plus is assumed.

Delimiters that may be used in the LP file are the following:

```
:
+
-
<   <=   =<
>   >=   =>
=
```

Delimiters given above on the same line are equivalent. The meaning of the delimiters will be explained below.

Blanks are non-significant characters. They may be used freely to improve readability of the LP file. Besides, blanks should be used to separate elements from each other if there is no other way to do that (for example, to separate a keyword from a following symbolic name).

The order of an LP file is:

- objective function definition;
- constraints section;
- bounds section;
- general, integer, and binary sections (can appear in arbitrary order);
- end keyword.

These components are discussed in following sections.

C.2 Objective function definition

The objective function definition must appear first in the LP file. It defines the objective function and specifies the optimization direction.

The objective function definition has the following form:

$$\left\{\begin{array}{l}\texttt{minimize}\\ \texttt{maximize}\end{array}\right\}\ f : s\ c\ x\ s\ c\ x\ \ldots\ s\ c\ x$$

where f is a symbolic name of the objective function, s is a sign `+` or `-`, c is a numeric constant that denotes an objective coefficient, x is a symbolic name of a variable.

[3] GLPK allows symbolic names having up to 255 characters.

If necessary, the objective function definition can be continued on as many text lines as desired.

The name of the objective function is optional and may be omitted (together with the semicolon that follows it). In this case the default name '`obj`' is assigned to the objective function.

If the very first sign s is omitted, the sign plus is assumed. Other signs cannot be omitted.

If some objective coefficient c is omitted, 1 is assumed.

Symbolic names x used to denote variables are recognized by context and therefore needn't to be declared somewhere else.

Here is an example of the objective function definition:

```
Minimize Z : - x1 + 2 x2 - 3.5 x3 + 4.997e3x(4) + x5 + x6 +
   x7 - .01x8
```

C.3 Constraints section

The constraints section must follow the objective function definition. It defines a system of equality and/or inequality constraints.

The constraint section has the following form:

`subject to`
constraint$_1$
constraint$_2$
. . .
constraint$_m$

where *constraint*$_i$, $i = 1, \ldots, m$, is a particular constraint definition.

Each constraint definition can be continued on as many text lines as desired. However, each constraint definition must begin on a new line except the very first constraint definition which can begin on the same line as the keyword '`subject to`'.

Constraint definitions have the following form:

$$r : s\ c\ x\ s\ c\ x\ \ldots\ s\ c\ x \left\{ \begin{array}{c} \texttt{<=} \\ \texttt{>=} \\ \texttt{=} \end{array} \right\} b$$

where r is a symbolic name of a constraint, s is a sign `+` or `-`, c is a numeric constant that denotes a constraint coefficient, x is a symbolic name of a variable, b is a right-hand side.

The name r of a constraint (which is the name of the corresponding auxiliary variable) is optional and may be omitted (together with the semicolon that follows it). In this case the default names like '`r.nnn`' are assigned to unnamed constraints.

The linear form $s\ c\ x\ s\ c\ x\ \ldots\ s\ c\ x$ in the left-hand side of a constraint definition has exactly the same meaning as in the case of the objective function definition (see above).

After the linear form one of the following delimiters that indicate the constraint sense must be specified:

`<=` means 'less than or equal to'
`>=` means 'greater than or equal to'
`=` means 'equal to'

The right hand side b is a numeric constant with an optional sign.

Here is an example of the constraints section:

```
Subject To
   one: y1 + 3 a1 - a2 - b >= 1.5
   y2 + 2 a3 + 2
      a4 - b >= -1.5
   two : y4 + 3 a1 + 4 a5 - b <= +1
   .20y5 + 5 a2 - b = 0
   1.7 y6 - a6 + 5 a777 - b >= 1
```

(Should note that it is impossible to express ranged constraints in the CPLEX LP format. Each a ranged constraint can be coded as two constraints with identical linear forms in the left-hand side, one of which specifies a lower bound and other does an upper one of the original ranged constraint.)

C.4 Bounds section

The bounds section is intended to define bounds of variables. This section is optional; if it is specified, it must follow the constraints section. If the bound section is omitted, all variables are assumed to be non-negative (i.e. that they have zero lower bound and no upper bound).

The bounds section has the following form:

`bounds`
definition$_1$
definition$_2$
. . .
definition$_p$

where *definition*$_k, k = 1, \ldots, p$, is a particular bound definition.

Each bound definition must begin on a new line[4] except the very first bound definition which can begin on the same line as the keyword '`bounds`'.

Syntactically constraint definitions can have one of the following six forms:

x `>=` l	specifies a lower bound
l `<=` x	specifies a lower bound
x `<=` u	specifies an upper bound
l `<=` x `<=` u	specifies both lower and upper bounds
x `=` t	specifies a fixed value
x `free`	specifies free variable

where x is a symbolic name of a variable, l is a numeric constant with an optional sign that defines a lower bound of the variable or `-inf` that means that the variable has no lower bound, u is a numeric constant with an optional sign that defines an upper bound of the variable or `+inf` that means that the variable has no upper bound, t is a numeric constant with an optional sign that defines a fixed value of the variable.

[4]The GLPK implementation allows several bound definitions to be placed on the same line.

By default all variables are non-negative, i.e. have zero lower bound and no upper bound. Therefore definitions of these default bounds can be omitted in the bounds section.

Here is an example of the bounds section:

```
Bounds
   -inf <= a1 <= 100
   -100 <= a2
   b <= 100
   x2 = +123.456
   x3 free
```

C.5 General, integer, and binary sections

The general, integer, and binary sections are intended to define some variables as integer or binary. All these sections are optional and needed only in case of MIP problems. If they are specified, they must follow the bounds section or, if the latter is omitted, the constraints section.

All the general, integer, and binary sections have the same form as follows:

$$\begin{array}{l} \left\{\begin{array}{l}\texttt{general}\\ \texttt{integer}\\ \texttt{binary}\end{array}\right\} \\ x_1 \\ x_2 \\ \dots \\ x_q \end{array}$$

where x_k is a symbolic name of variable, $k = 1, \dots, q$.

Each symbolic name must begin on a new line[5] except the very first symbolic name which can begin on the same line as the keyword '`general`', '`integer`', or '`binary`'.

If a variable appears in the general or the integer section, it is assumed to be general integer variable. If a variable appears in the binary section, it is assumed to be binary variable, i.e. an integer variable whose lower bound is zero and upper bound is one. (Note that if bounds of a variable are specified in the bounds section and then the variable appears in the binary section, its previously specified bounds are ignored.)

Here is an example of the integer section:

```
Integer
   z12
   z22
   z35
```

C.6 End keyword

The keyword '`end`' is intended to end the LP file. It must begin on a separate line and no other elements (except comments and blank lines) must follow it. Although this keyword is optional, it is strongly recommended to include it in the LP file.

[5]The GLPK implementation allows several symbolic names to be placed on the same line.

C.7 Example of CPLEX LP file

Here is a complete example of CPLEX LP file that corresponds to the example given in Section B.11, page 71.

```
\* plan.lp *\

Minimize
   value: .03 bin1 + .08 bin2 + .17 bin3 + .12 bin4 + .15 bin5 +
          .21 alum + .38 silicon

Subject To
   yield:     bin1 +     bin2 +     bin3 +     bin4 +     bin5 +
              alum +     silicon                                 =  2000

   fe:    .15 bin1 + .04 bin2 + .02 bin3 + .04 bin4 + .02 bin5 +
          .01 alum + .03 silicon                                 <=   60

   cu:    .03 bin1 + .05 bin2 + .08 bin3 + .02 bin4 + .06 bin5 +
          .01 alum                                               <=  100

   mn:    .02 bin1 + .04 bin2 + .01 bin3 + .02 bin4 + .02 bin5   <=   40

   mg:    .02 bin1 + .03 bin2                     + .01 bin5   <=   30

   al:    .70 bin1 + .75 bin2 + .80 bin3 + .75 bin4 + .80 bin5 +
          .97 alum                                               >= 1500

   si1:   .02 bin1 + .06 bin2 + .08 bin3 + .12 bin4 + .02 bin5 +
          .01 alum + .97 silicon                                 >=  250

   si2:   .02 bin1 + .06 bin2 + .08 bin3 + .12 bin4 + .02 bin5 +
          .01 alum + .97 silicon                                 <=  300

Bounds
          bin1 <=  200
          bin2 <= 2500
   400 <= bin3 <=  800
   100 <= bin4 <=  700
          bin5 <= 1500

End

\* eof *\
```

Appendix D

Stand-alone LP/MIP Solver

The GLPK package includes the program `glpsol` which is a stand-alone LP/MIP solver. This program can be invoked from the command line of from the shell to read LP/MIP problem data in any format supported by GLPK, solve the problem, and write the obtained problem solution to a text file in plain format.

Usage

`glpsol` [*options...*] [*filename*]

General options

`--glp`	read LP/MIP model in GNU LP format
`--mps`	read LP/MIP problem in fixed MPS format (default)
`--freemps`	read LP/MIP problem in free MPS format
`--cpxlp`	read LP/MIP problem in CPLEX LP format
`--math`	read LP/MIP model written in GNU MathProg modeling language
`-m` *filename*, `--model` *filename*	
	read model section and optional data section from filename (the same as `--math`)
`-d` *filename*, `--data` *filename*	
	read data section from filename (for `--math` only); if model file also has data section, that section is ignored
`-y` *filename*, `--display` *filename*	
	send display output to filename (for `--math` only); by default the output is sent to `stdout`
`--min`	minimization
`--max`	maximization
`--scale`	scale problem (default)
`--noscale`	do not scale problem
`--simplex`	use simplex method (default)
`--interior`	use interior point method (for pure LP only)
`-o` *filename*, `--output` *filename*	
	write solution to filename in plain text format
`--bounds` *filename*	
	write sensitivity bounds to filename in plain text format (LP only)

`--tmlim` *nnn*	limit solution time to *nnn* seconds (`--tmlim 0` allows obtaining solution at initial point)
`--check`	do not solve problem, check input data only
`--name` *probname*	change problem name to *probname*
`--plain`	use plain names of rows and columns (default)
`--orig`	try using original names of rows and columns (default for `--mps`)
`--wglp` *filename*	write problem to *filename* in GNU LP format
`--wmps` *filename*	write problem to *filename* in fixed MPS format
`--wfreemps` *filename*	write problem to *filename* in free MPS format
`--wcpxlp` *filename*	write problem to *filename* in CPLEX LP format
`--wtxt` *filename*	write problem to *filename* in plain text format
`-h`, `--help`	display this help information and exit
`-v`, `--version`	display program version and exit

Options specific to simplex method

`--std`	use standard initial basis of all slacks
`--adv`	use advanced initial basis (default)
`--bas` *filename*	read initial basis from *filename* in MPS format
`--steep`	use steepest edge technique (default)
`--nosteep`	use standard "textbook" pricing
`--relax`	use Harris' two-pass ratio test (default)
`--norelax`	use standard "textbook" ratio test
`--presol`	use LP presolver (default; assumes `--scale` and `--adv`)
`--nopresol`	do not use LP presolver
`--wbas` *filename*	write final basis to *filename* in MPS format

Options specific to MIP

`--nomip`	consider all integer variables as continuous (allows solving MIP as pure LP)
`--first`	branch on first integer variable
`--last`	branch on last integer variable
`--drtom`	branch using heuristic by Driebeck and Tomlin (default)
`--mostf`	branch on most fractional varaible
`--dfs`	backtrack using depth first search
`--bfs`	backtrack using breadth first search
`--bestp`	backtrack using the best projection heuristic (default)
`--bestb`	backtrack using node with best local bound

For description of the MPS format see Appendix B, page 66.

For description of the CPLEX LP format see Appendix C, page 77.

For description of the modeling language see the document "GLPK: Modeling Language GNU MathProg" included in the GLPK distribution.