GNU Linear Programming Kit

Reference Manual

Version 4.32

(Draft Edition, October 2008)

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

Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, 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)

$$z = c_1 x_{m+1} + c_2 x_{m+2} + \ldots + c_n x_{m+n} + c_0 \tag{1.1}$$

subject to linear constraints

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

and bounds of variables

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

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

- satisfy to all the linear constraints (1.2), and
- are within their bounds (1.3), and
- provide the smallest (in case of minimization) or the largest (in case of maximization) value of the objective function (1.1).

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 Using the package

1.3.1 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

$$\begin{aligned} x_1 + x_2 + x_3 &\le 100 \\ 10x_1 + 4x_2 + 5x_3 &\le 600 \\ 2x_1 + 2x_2 + 6x_3 &\le 300 \end{aligned}$$

where all variables are non-negative

$$x_1 \ge 0,\ x_2 \ge 0,\ x_3 \ge 0$$

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

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

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.[1]

[1] If you just need to solve LP or MIP instance, you may write it in MPS or CPLEX LP format and then use the GLPK stand-alone solver to obtain a solution. This is much less time-consuming than programming in C with GLPK API routines.

```
/* sample.c */

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

int main(void)
{     glp_prob *lp;
      int ia[1+1000], ja[1+1000];
      double ar[1+1000], z, x1, x2, x3;
s1:   lp = glp_create_prob();
s2:   glp_set_prob_name(lp, "sample");
s3:   glp_set_obj_dir(lp, GLP_MAX);
s4:   glp_add_rows(lp, 3);
s5:   glp_set_row_name(lp, 1, "p");
s6:   glp_set_row_bnds(lp, 1, GLP_UP, 0.0, 100.0);
s7:   glp_set_row_name(lp, 2, "q");
s8:   glp_set_row_bnds(lp, 2, GLP_UP, 0.0, 600.0);
s9:   glp_set_row_name(lp, 3, "r");
s10:  glp_set_row_bnds(lp, 3, GLP_UP, 0.0, 300.0);
s11:  glp_add_cols(lp, 3);
s12:  glp_set_col_name(lp, 1, "x1");
s13:  glp_set_col_bnds(lp, 1, GLP_LO, 0.0, 0.0);
s14:  glp_set_obj_coef(lp, 1, 10.0);
s15:  glp_set_col_name(lp, 2, "x2");
s16:  glp_set_col_bnds(lp, 2, GLP_LO, 0.0, 0.0);
s17:  glp_set_obj_coef(lp, 2, 6.0);
s18:  glp_set_col_name(lp, 3, "x3");
s19:  glp_set_col_bnds(lp, 3, GLP_LO, 0.0, 0.0);
s20:  glp_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:  glp_load_matrix(lp, 9, ia, ja, ar);
```

```
s31:  glp_simplex(lp, NULL);
s32:  z = glp_get_obj_val(lp);
s33:  x1 = glp_get_col_prim(lp, 1);
s34:  x2 = glp_get_col_prim(lp, 2);
s35:  x3 = glp_get_col_prim(lp, 3);
s36:  printf("\nz = %g; x1 = %g; x2 = %g; x3 = %g\n",
         z, x1, x2, x3);
s37:  glp_delete_prob(lp);
      return 0;
}

/* eof */
```

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

The statement s3 calls the routine glp_set_obj_dir in order to set the optimization direction flag, where GLP_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 GLP_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 GLP_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 glp_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 glp_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 writes the optimal solution to the standard output. The printout may look like follows:

```
*     0:   objval =   0.000000000e+00   infeas =   0.000000000e+00 (0)
*     2:   objval =   7.333333333e+02   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND

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

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

1.3.2 Compiling

The GLPK package has the only header file glpk.h, which should be available on compiling a C (or C++) program using GLPK API routines.

If the header file is installed in the default location /usr/local/include, the following typical command may be used to compile, say, the example C program described above with the GNU C compiler:

```
$ gcc -c sample.c
```

If glpk.h is not in the default location, the corresponding directory containing it should be made known to the C compiler through -I option, for example:

```
$ gcc -I/foo/bar/glpk-4.15/include -c sample.c
```

In any case the compilation results in an object file sample.o.

1.3.3 Linking

The GLPK library is a single file libglpk.a. (On systems which support shared libraries there may be also a shared version of the library libglpk.so.)

If the library is installed in the default location `/usr/local/lib`, the following typical command may be used to link, say, the example C program described above against with the library:

```
$ gcc sample.o -lglpk -lm
```

If the GLPK library is not in the default location, the corresponding directory containing it should be made known to the linker through `-L` option, for example:

```
$ gcc -L/foo/bar/glpk-4.15 sample.o -lglpk -lm
```

Depending on configuration of the package linking against with the GLPK library may require the following optional libraries:

`-lgmp`	the GNU MP bignum library;
`-lz`	the zlib data compression library;
`-lltdl`	the GNU ltdl shared support library.

in which case corresponding libraries should be also made known to the linker, for example:

```
$ gcc sample.o -lglpk -lz -lltdl -lm
```

For more details about configuration options of the GLPK package see Appendix A, page 136.

Chapter 2

Basic API Routines

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

Library header

All GLPK API data types and routines are defined in the header file `glpk.h`. It should be included in all source files which use GLPK API, either directly or indirectly through some other header file as follows:

```
#include <glpk.h>
```

Error handling

If some GLPK API routine detects erroneous or incorrect data passed by the application program, it writes appropriate diagnostic messages to the terminal and then abnormally terminates the application program. In most practical cases this allows to simplify programming by 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 the return code.

Thread safety

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

Array indexing

Normally all GLPK API routines start array indexing from 1, not from 0 (except the specially stipulated cases). This means, for example, that if some vector x of the length n is passed as an array to some GLPK API 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 the array to GLPK routines in a usual way.

2.1 Problem object

All GLPK API routines deal with so called *problem object*, which is a program object of type `glp_prob` and intended to represent a particular LP or MIP instance.

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

```
typedef struct { ... } glp_prob;
```

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

The problem object consists of five 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 9). It includes the following components:

- 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 for informational purposes. They also can 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 9).

Scale factors are used internally for scaling 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 9). 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 the current basic solution. It 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 the 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 include primal values of all auxiliary and structural variables for the most recently obtained MIP solution.

Note that in case of MIP problem the basis segment corresponds to the 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.

2.2 Problem creating and modifying routines

2.2.1 Create problem object

Synopsis

```
glp_prob *glp_create_prob(void);
```

Description

The routine `glp_create_prob` creates a new problem object, which initially is "empty", i.e. has no rows and 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 Assign (change) problem name

Synopsis

```
void glp_set_prob_name(glp_prob *lp, const char *name);
```

Description

The routine `glp_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 Assign (change) objective function name

Synopsis

```
void glp_set_obj_name(glp_prob *lp, const char *name);
```

Description

The routine `glp_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 Set (change) optimization direction flag

Synopsis

```
void glp_set_obj_dir(glp_prob *lp, int dir);
```

Description

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

`GLP_MIN` minimization;

`GLP_MAX` maximization.

(Note that by default the problem is minimization.)

2.2.5 Add new rows to problem object

Synopsis

```
int glp_add_rows(glp_prob *lp, int nrs);
```

Description

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

2.2.6 Add new columns to problem object

Synopsis

```
int glp_add_cols(glp_prob *lp, int ncs);
```

Description

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

2.2.7 Assign (change) row name

Synopsis

```
void glp_set_row_name(glp_prob *lp, int i, const char *name);
```

Description

The routine `glp_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 Assign (change) column name

Synopsis

```
void glp_set_col_name(glp_prob *lp, int j, const char *name);
```

Description

The routine `glp_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 Set (change) row bounds

Synopsis

```
void glp_set_row_bnds(glp_prob *lp, int i, int type,
      double lb, double ub);
```

Description

The routine `glp_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
`GLP_FR`	$-\infty < x < +\infty$	Free (unbounded) variable
`GLP_LO`	$lb \leq x < +\infty$	Variable with lower bound
`GLP_UP`	$-\infty < x \leq ub$	Variable with upper bound
`GLP_DB`	$lb \leq x \leq ub$	Double-bounded variable
`GLP_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.

Being added to the problem object each row is initially free, i.e. its type is `GLP_FR`.

2.2.10 Set (change) column bounds

Synopsis

```
void glp_set_col_bnds(glp_prob *lp, int j, int type,
      double lb, double ub);
```

Description

The routine `glp_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
GLP_FR	$-\infty < x < +\infty$	Free (unbounded) variable
GLP_LO	$lb \leq x < +\infty$	Variable with lower bound
GLP_UP	$-\infty < x \leq ub$	Variable with upper bound
GLP_DB	$lb \leq x \leq ub$	Double-bounded variable
GLP_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.

Being added to the problem object each column is initially fixed at zero, i.e. its type is `GLP_FX` and both bounds are 0.

2.2.11 Set (change) objective coefficient or constant term

Synopsis

```
void glp_set_obj_coef(glp_prob *lp, int j, double coef);
```

Description

The routine `glp_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 Set (replace) row of the constraint matrix

Synopsis

```
void glp_set_mat_row(glp_prob *lp, int i, int len,
      const int ind[], const double val[]);
```

Description

The routine `glp_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. Elements with identical column indices are not allowed. Zero elements are allowed, but they are not stored in the constraint matrix.

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

2.2.13 Set (replace) column of the constraint matrix

Synopsis

```
void glp_set_mat_col(glp_prob *lp, int j, int len,
      const int ind[], const double val[]);
```

Description

The routine `glp_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. Elements with identical row indices are not allowed. Zero elements are allowed, but they are not stored in the constraint matrix.

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

2.2.14 Load (replace) the whole constraint matrix

Synopsis

```
void glp_load_matrix(glp_prob *lp, int ne, const int ia[],
      const int ja[], const double ar[]);
```

Description

The routine `glp_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. Coefficients with identical indices are not allowed. Zero coefficients are allowed, however, they are not stored in the constraint matrix.

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

2.2.15 Delete rows from problem object

Synopsis

```
void glp_del_rows(glp_prob *lp, int nrs, const int num[]);
```

Description

The routine `glp_del_rows` deletes rows from the specified problem object. Ordinal numbers of rows to be deleted should 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 Delete columns from problem object

Synopsis

```
void glp_del_cols(glp_prob *lp, int ncs, const int num[]);
```

Description

The routine `glp_del_cols` deletes columns from the specified problem object. Ordinal numbers of columns to be deleted should 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 Erase problem object content

Synopsis

```
void glp_erase_prob(glp_prob *lp);
```

Description

The routine `glp_erase_prob` erases the content of the specified problem object. The effect of this operation is the same as if the problem object would be deleted with the routine `glp_delete_prob` and then created anew with the routine `glp_create_prob`, with the only exception that the handle (pointer) to the problem object remains valid.

2.2.18 Delete problem object

Synopsis

```
void glp_delete_prob(glp_prob *lp);
```

Description

The routine `glp_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 Retrieve problem name

Synopsis

```
const char *glp_get_prob_name(glp_prob *lp);
```

Returns

The routine `glp_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 Retrieve objective function name

Synopsis

```
const char *glp_get_obj_name(glp_prob *lp);
```

Returns

The routine `glp_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 Retrieve optimization direction flag

Synopsis

```
int glp_get_obj_dir(glp_prob *lp);
```

Returns

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

`GLP_MIN` minimization;
`GLP_MAX` maximization.

2.3.4 Retrieve number of rows

Synopsis

```
int glp_get_num_rows(glp_prob *lp);
```

Returns

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

2.3.5 Retrieve number of columns

Synopsis

```
int glp_get_num_cols(glp_prob *lp);
```

Returns

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

2.3.6 Retrieve row name

Synopsis

```
const char *glp_get_row_name(glp_prob *lp, int i);
```

Returns

The routine `glp_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 Retrieve column name

Synopsis

```
const char *glp_get_col_name(glp_prob *lp, int j);
```

Returns

The routine `glp_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 Retrieve row type

Synopsis

```
int glp_get_row_type(glp_prob *lp, int i);
```

Returns

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

GLP_FR	free (unbounded) variable;
GLP_LO	variable with lower bound;
GLP_UP	variable with upper bound;
GLP_DB	double-bounded variable;
GLP_FX	fixed variable.

2.3.9 Retrieve row lower bound

Synopsis

```
double glp_get_row_lb(glp_prob *lp, int i);
```

Returns

The routine glp_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 -DBL_MAX.

2.3.10 Retrieve row upper bound

Synopsis

```
double glp_get_row_ub(glp_prob *lp, int i);
```

Returns

The routine glp_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 +DBL_MAX.

2.3.11 Retrieve column type

Synopsis

```
int glp_get_col_type(glp_prob *lp, int j);
```

Returns

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

GLP_FR	free (unbounded) variable;
GLP_LO	variable with lower bound;
GLP_UP	variable with upper bound;
GLP_DB	double-bounded variable;
GLP_FX	fixed variable.

2.3.12 Retrieve column lower bound

Synopsis

```
double glp_get_col_lb(glp_prob *lp, int j);
```

Returns

The routine glp_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 -DBL_MAX.

2.3.13 Retrieve column upper bound

Synopsis

```
double glp_get_col_ub(glp_prob *lp, int j);
```

Returns

The routine glp_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 +DBL_MAX.

2.3.14 Retrieve objective coefficient or constant term

Synopsis

```
double glp_get_obj_coef(glp_prob *lp, int j);
```

Returns

The routine `glp_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 Retrieve number of constraint coefficients

Synopsis

```
int glp_get_num_nz(glp_prob *lp);
```

Returns

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

2.3.16 Retrieve row of the constraint matrix

Synopsis

```
int glp_get_mat_row(glp_prob *lp, int i, int ind[],
      double val[]);
```

Description

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

2.3.17 Retrieve column of the constraint matrix

Synopsis

```
int glp_get_mat_col(glp_prob *lp, int j, int ind[],
      double val[]);
```

Description

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

2.4 Row and column searching routines

2.4.1 Create the name index

Synopsis

```
void glp_create_index(glp_prob *lp);
```

Description

The routine `glp_create_index` creates the name index for the specified problem object. The name index is an auxiliary data structure, which is intended to quickly (i.e. for logarithmic time) find rows and columns by their names.

This routine can be called at any time. If the name index already exists, the routine does nothing.

2.4.2 Find row by its name

Synopsis

```
int glp_find_row(glp_prob *lp, const char *name);
```

Returns

The routine `glp_find_row` returns the ordinal number of a row, which is assigned (by the routine `glp_set_row_name`) the specified symbolic `name`. If no such row exists, the routine returns 0.

2.4.3 Find column by its name

Synopsis

```
int glp_find_col(glp_prob *lp, const char *name);
```

Returns

The routine `glp_find_col` returns the ordinal number of a column, which is assigned (by the routine `glp_set_col_name`) the specified symbolic `name`. If no such column exists, the routine returns 0.

2.4.4 Delete the name index

Synopsis

```
void glp_delete_index(glp_prob *lp);
```

Description

The routine `glp_delete_index` deletes the name index previously created by the routine `glp_create_index` and frees the memory allocated to this auxiliary data structure.

This routine can be called at any time. If the name index does not exist, the routine does nothing.

2.5 Problem scaling routines

2.5.1 Background

In GLPK the *scaling* means a linear transformation applied to the constraint matrix to improve its numerical properties.[1]

The main equality is the following:

$$\widetilde{A} = RAS, \tag{2.1}$$

where $A = (a_{ij})$ is the original constraint matrix, $R = (r_{ii}) > 0$ is a diagonal matrix used to scale rows (constraints), $S = (s_{jj}) > 0$ is a diagonal matrix used to scale columns (variables), $\widetilde{A}$ is the scaled constraint matrix.

From (2.1) it follows that in the *scaled* problem instance each original constraint coefficient a_{ij} is replaced by corresponding scaled constraint coefficient:

$$\widetilde{a}_{ij} = r_{ii} a_{ij} s_{jj}. \tag{2.2}$$

Note that the scaling is performed internally and therefore transparently to the user. This means that on API level the user always deal with unscaled data.

Scale factors r_{ii} and s_{jj} can be set or changed at any time either directly by the application program in a problem specific way (with the routines `glp_set_rii` and `glp_set_sjj`), or by some API routines intended for automatic scaling.

2.5.2 Set (change) row scale factor

Synopsis

```
void glp_set_rii(glp_prob *lp, int i, double rii);
```

Description

The routine `glp_set_rii` sets (changes) the scale factor r_{ii} for i-th row of the specified problem object.

[1] In many cases a proper scaling allows making the constraint matrix to be better conditioned, i.e. decreasing its condition number, that makes computations numerically more stable.

2.5.3 Set (change) column scale factor

Synopsis

```
void glp_set_sjj(glp_prob *lp, int j, double sjj);
```

Description

The routine glp_set_sjj sets (changes) the scale factor s_{jj} for j-th column of the specified problem object.

2.5.4 Retrieve row scale factor

Synopsis

```
double glp_get_rii(glp_prob *lp, int i);
```

Returns

The routine glp_get_rii returns current scale factor r_{ii} for i-th row of the specified problem object.

2.5.5 Retrieve column scale factor

Synopsis

```
double glp_get_sjj(glp_prob *lp, int j);
```

Returns

The routine glp_get_sjj returns current scale factor s_{jj} for j-th column of the specified problem object.

2.5.6 Scale problem data

Synopsis

```
void glp_scale_prob(glp_prob *lp, int flags);
```

Description

The routine glp_scale_prob performs automatic scaling of problem data for the specified problem object.

The parameter `flags` specifies scaling options used by the routine. The options can be combined with the bitwise OR operator and may be the following:

`GLP_SF_GM`	perform geometric mean scaling;
`GLP_SF_EQ`	perform equilibration scaling;
`GLP_SF_2N`	round scale factors to nearest power of two;
`GLP_SF_SKIP`	skip scaling, if the problem is well scaled.

The parameter `flags` may be specified as `GLP_SF_AUTO`, in which case the routine chooses the scaling options automatically.

2.5.7 Unscale problem data

Synopsis

```
void glp_unscale_prob(glp_prob *lp);
```

The routine `glp_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.

2.6 LP basis constructing routines

2.6.1 Background

To start the search the simplex method needs a valid initial basis. In GLPK the basis is completely defined by a set of *statuses* assigned to *all* (auxiliary and structural) variables, where the status may be one of the following:

GLP_BS basic variable;
GLP_NL non-basic variable having active lower bound;
GLP_NU non-basic variable having active upper bound;
GLP_NF non-basic free variable;
GLP_NS non-basic fixed variable.

The basis is *valid*, if the basis matrix, which is a matrix built of columns of the augmented constraint matrix $(I\,|-A)$ corresponding to basic variables, is non-singular. This, in particular, means that the number of basic variables must be the same as the number of rows in the problem object. (For more details see Section 4.1, page 82.)

Any initial basis may be constructed (or restored) with the API routines `glp_set_row_stat` and `glp_set_col_stat` by assigning appropriate statuses to auxiliary and structural variables. Another way to construct an initial basis is to use API routines like `glp_adv_basis`, which implement so called *crashing*.[2] Note that on normal exit the simplex solver remains the basis valid, so in case of reoptimization there is no need to construct an initial basis from scratch.

2.6.2 Set (change) row status

Synopsis

```
void glp_set_row_stat(glp_prob *lp, int i, int stat);
```

Description

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

GLP_BS make the row basic (make the constraint inactive);
GLP_NL make the row non-basic (make the constraint active);

[2]This term is from early linear programming systems and means a heuristic to construct a valid initial basis.

- `GLP_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 `GLP_NL` (only in the case of this routine);
- `GLP_NF` the same as `GLP_NL` (only in the case of this routine);
- `GLP_NS` the same as `GLP_NL` (only in the case of this routine).

2.6.3 Set (change) column status

Synopsis

```
void glp_set_col_stat(glp_prob *lp, int j, int stat);
```

Description

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

- `GLP_BS` make the column basic;
- `GLP_NL` make the column non-basic;
- `GLP_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 `GLP_NL` (only in the case of this routine);
- `GLP_NF` the same as `GLP_NL` (only in the case of this routine);
- `GLP_NS` the same as `GLP_NL` (only in the case of this routine).

2.6.4 Construct standard initial LP basis

Synopsis

```
void glp_std_basis(glp_prob *lp);
```

Description

The routine `glp_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).

2.6.5 Construct advanced initial LP basis

Synopsis

```
void glp_adv_basis(glp_prob *lp, int flags);
```

Description

The routine `glp_adv_basis` constructs an advanced initial LP basis for the specified problem object.

The parameter `flags` is reserved for use in the future and must be specified as zero.

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 the 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.

2.6.6 Construct Bixby's initial LP basis

Synopsis

```
void glp_cpx_basis(glp_prob *lp);
```

Description

The routine `glp_cpx_basis` constructs an initial basis for the specified problem object with the algorithm proposed by R. Bixby.[3]

[3]Robert E. Bixby, "Implementing the Simplex Method: The Initial Basis." ORSA Journal on Computing, Vol. 4, No. 3, 1992, pp. 267-84.

2.7 Simplex method routines

The *simplex method* is a well known efficient numerical procedure to solve LP problems.

On each iteration the simplex method 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:

$$
\begin{array}{rl}
z = & d_1(x_N)_1 + \ d_2(x_N)_2 + \ldots + \ d_n(x_N)_n \\
(x_B)_1 = & \xi_{11}(x_N)_1 + \ \xi_{12}(x_N)_2 + \ldots + \ \xi_{1n}(x_N)_n \\
(x_B)_2 = & \xi_{21}(x_N)_1 + \ \xi_{22}(x_N)_2 + \ldots + \ \xi_{2n}(x_N)_n \\
 & \ldots\ldots\ldots\ldots\ldots\ldots \\
(x_B)_m = & \xi_{m1}(x_N)_1 + \xi_{m2}(x_N)_2 + \ldots + \xi_{mn}(x_N)_n
\end{array}
\tag{2.1}
$$

where: $(x_B)_1, (x_B)_2, \ldots, (x_B)_m$ are basic variables; $(x_N)_1, (x_N)_2, \ldots, (x_N)_n$ are non-basic variables; $d_1, d_2, \ldots, d_n$ are reduced costs; $\xi_{11}, \xi_{12}, \ldots, \xi_{mn}$ are 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 known that if an optimal solution of the LP problem (1.1)—(1.3) exists, it can always be written in the form (2.1), 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.)

2.7.1 Solve LP problem with the simplex method

Synopsis

```
int glp_simplex(glp_prob *lp, const glp_smcp *parm);
```

Description

The routine `glp_simplex` is a driver to the LP solver based on the simplex method. This routine retrieves problem data from the specified problem object, calls the solver to solve the problem instance, and stores results of computations back into the problem object.

The simplex solver has a set of control parameters. Values of the control parameters can be passed in the structure `glp_smcp`, which the parameter `parm` points to. For detailed description of this structure see paragraph "Control parameters" below. Before specifying some control parameters the application program should initialize the structure `glp_smcp` by default values of all control parameters using the routine `glp_init_smcp` (see the next subsection). This is needed for backward compatibility, because in the future there may appear new members in the structure `glp_smcp`.

The parameter `parm` can be specified as `NULL`, in which case the solver uses default settings.

Returns

0	The LP problem instance has been successfully solved. (This code does *not* necessarily mean that the solver has found optimal solution. It only means that the solution process was successful.)
`GLP_EBADB`	Unable to start the search, because the initial basis specified in the problem object is invalid—the number of basic (auxiliary and structural) variables is not the same as the number of rows in the problem object.
`GLP_ESING`	Unable to start the search, because the basis matrix corresponding to the initial basis is singular within the working precision.
`GLP_ECOND`	Unable to start the search, because the basis matrix corresponding to the initial basis is ill-conditioned, i.e. its condition number is too large.
`GLP_EBOUND`	Unable to start the search, because some double-bounded (auxiliary or structural) variables have incorrect bounds.
`GLP_EFAIL`	The search was prematurely terminated due to the solver failure.
`GLP_EOBJLL`	The search was prematurely terminated, because the objective function being maximized has reached its lower limit and continues decreasing (the dual simplex only).
`GLP_EOBJUL`	The search was prematurely terminated, because the objective function being minimized has reached its upper limit and continues increasing (the dual simplex only).
`GLP_EITLIM`	The search was prematurely terminated, because the simplex iteration limit has been exceeded.
`GLP_ETMLIM`	The search was prematurely terminated, because the time limit has been exceeded.
`GLP_ENOPFS`	The LP problem instance has no primal feasible solution (only if the LP presolver is used).
`GLP_ENODFS`	The LP problem instance has no dual feasible solution (only if the LP presolver is used).

Built-in LP presolver

The simplex solver has *built-in LP presolver*. It 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 the corresponding basic solution of the original problem.

Presolving is an optional feature of the routine `glp_simplex`, and by default it is disabled. In order to enable the LP presolver the control parameter `presolve` should be set to `GLP_ON` (see paragraph "Control parameters" below). Presolving may be used when the problem instance is solved for the first time. However, on performing re-optimization the presolver should be disabled.

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 you need to know a basic solution even if it is infeasible or non-optimal, the presolver should be disabled.

Solver terminal output

Solving large problem instances may take a long time, so the solver reports some information about the current basic solution, which is sent to the terminal. This information has the following format:

```
nnn:  obj = xxx  infeas = yyy (ddd)
```

where: '`nnn`' is the iteration number, '`xxx`' is the current value of the objective function (it is is unscaled and has correct sign); '`yyy`' is the current sum of primal or dual infeasibilities (it is scaled and therefore may be used only for visual estimating), '`ddd`' is the current number of fixed basic variables.

The symbol preceding the iteration number indicates which phase of the simplex method is in effect:

Blank means that the solver is searching for primal feasible solution using the primal simplex or for dual feasible solution using the dual simplex;

Asterisk (`*`) means that the solver is searching for optimal solution using the primal simplex;

Vertical dash (`|`) means that the solver is searching for optimal solution using the dual simplex.

Control parameters

This paragraph describes all control parameters currently used in the simplex solver. Symbolic names of control parameters are names of corresponding members in the structure `glp_smcp`.

`int msg_lev` (default: `GLP_MSG_ALL`)
Message level for terminal output:
`GLP_MSG_OFF` — no output;
`GLP_MSG_ERR` — error and warning messages only;
`GLP_MSG_ON` — normal output;
`GLP_MSG_ALL` — full output (including informational messages).

`int meth` (default: `GLP_PRIMAL`)
Simplex method option:
`GLP_PRIMAL` — use two-phase primal simplex;
`GLP_DUAL` — use two-phase dual simplex;
`GLP_DUALP` — use two-phase dual simplex, and if it fails, switch to the primal simplex.

`int pricing` (default: `GLP_PT_PSE`)
Pricing technique:
`GLP_PT_STD` — standard (textbook);
`GLP_PT_PSE` — projected steepest edge.

`int r_test` (default: `GLP_RT_HAR`)
Ratio test technique:
`GLP_RT_STD` — standard (textbook);
`GLP_RT_HAR` — Harris' two-pass ratio test.

`double tol_bnd` (default: `1e-7`)
Tolerance used to check if the basic solution is primal feasible. (Do not change this parameter without detailed understanding its purpose.)

`double tol_dj` (default: `1e-7`)
Tolerance used to check if the basic solution is dual feasible. (Do not change this parameter without detailed understanding its purpose.)

`double tol_piv` (default: `1e-10`)
Tolerance used to choose eligble pivotal elements of the simplex table. (Do not change this parameter without detailed understanding its purpose.)

`double obj_ll` (default: `-DBL_MAX`)
Lower limit of the objective function. If the objective function reaches this limit and continues decreasing, the solver terminates the search. (Used in the dual simplex only.)

`double obj_ul` (default: `+DBL_MAX`)
Upper limit of the objective function. If the objective function reaches this limit and continues increasing, the solver terminates the search. (Used in the dual simplex only.)

`int it_lim` (default: `INT_MAX`)
Simplex iteration limit.

`int tm_lim` (default: `INT_MAX`)
Searching time limit, in milliseconds.

`int out_frq` (default: `200`)
Output frequency, in iterations. This parameter specifies how frequently the solver sends information about the solution process to the terminal.

`int out_dly` (default: `0`)
Output delay, in milliseconds. This parameter specifies how long the solver should delay sending information about the solution process to the terminal.

`int presolve` (default: `GLP_OFF`)
LP presolver option:
`GLP_ON` — enable using the LP presolver;
`GLP_OFF` — disable using the LP presolver.

2.7.2 Initialize simplex method control parameters

Synopsis

```
int glp_init_smcp(glp_smcp *parm);
```

Description

The routine `glp_init_smcp` initializes control parameters, which are used by the simplex solver, with default values.

Default values of the control parameters are stored in a `glp_smcp` structure, which the parameter `parm` points to.

2.7.3 Solve LP problem in exact arithmetic

Synopsis

```
int lpx_exact(glp_prob *lp);
```

Description

The routine `lpx_exact` is an experimental implementation of the primal two-phase simplex method based on exact (rational) arithmetic. It is similar to the routine `glp_simplex`, however, for all internal computations it uses arithmetic of rational numbers, which is exact in mathematical sense, i.e. free of round-off errors unlike floating-point arithmetic.

Returns

The routine `lpx_exact` 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`	either the LP problem has no rows and/or columns, or the initial basis is invalid, or the basis matrix is exactly singular.
`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.

2.7.4 Retrieve generic status of basic solution

Synopsis

```
int glp_get_status(glp_prob *lp);
```

Returns

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

`GLP_OPT`	solution is optimal;
`GLP_FEAS`	solution is feasible;
`GLP_INFEAS`	solution is infeasible;
`GLP_NOFEAS`	problem has no feasible solution;
`GLP_UNBND`	problem has unbounded solution;
`GLP_UNDEF`	solution is undefined.

More detailed information about the status of basic solution can be retrieved with the routines `glp_get_prim_stat` and `glp_get_dual_stat`.

2.7.5 Retrieve status of primal basic solution

Synopsis

```
int glp_get_prim_stat(glp_prob *lp);
```

Returns

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

`GLP_UNDEF`	primal solution is undefined;
`GLP_FEAS`	primal solution is feasible;
`GLP_INFEAS`	primal solution is infeasible;
`GLP_NOFEAS`	no primal feasible solution exists.

2.7.6 Retrieve status of dual basic solution

Synopsis

```
int glp_get_dual_stat(glp_prob *lp);
```

Returns

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

`GLP_UNDEF`	dual solution is undefined;
`GLP_FEAS`	dual solution is feasible;
`GLP_INFEAS`	dual solution is infeasible;
`GLP_NOFEAS`	no dual feasible solution exists.

2.7.7 Retrieve objective value

Synopsis

```
double glp_get_obj_val(glp_prob *lp);
```

Returns

The routine `glp_get_obj_val` returns current value of the objective function.

2.7.8 Retrieve row status

Synopsis

```
int glp_get_row_stat(glp_prob *lp, int i);
```

Returns

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

`GLP_BS`	basic variable;
`GLP_NL`	non-basic variable on its lower bound;
`GLP_NU`	non-basic variable on its upper bound;
`GLP_NF`	non-basic free (unbounded) variable;
`GLP_NS`	non-basic fixed variable.

2.7.9 Retrieve row primal value

Synopsis

```
double glp_get_row_prim(glp_prob *lp, int i);
```

Returns

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

2.7.10 Retrieve row dual value

Synopsis

```
double glp_get_row_dual(glp_prob *lp, int i);
```

Returns

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

2.7.11 Retrieve column status

Synopsis

```
int glp_get_col_stat(glp_prob *lp, int j);
```

Returns

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

- `GLP_BS` basic variable;
- `GLP_NL` non-basic variable on its lower bound;
- `GLP_NU` non-basic variable on its upper bound;
- `GLP_NF` non-basic free (unbounded) variable;
- `GLP_NS` non-basic fixed variable.

2.7.12 Retrieve column primal value

Synopsis

```
double glp_get_col_prim(glp_prob *lp, int j);
```

Returns

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

2.7.13 Retrieve column dual value

Synopsis

```
double glp_get_col_dual(glp_prob *lp, int j);
```

Returns

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

2.7.14 Retrieve non-basic variable causing unboundness

Synopsis

```
int lpx_get_ray_info(glp_prob *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.

2.7.15 Check Karush-Kuhn-Tucker conditions

Synopsis

```
void lpx_check_kkt(glp_prob *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 results of the check. 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:

$$x_R - Ax_S = 0, \qquad \text{(KKT.PE)}$$

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:

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

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 $\tilde{A} = (I|-A)$, 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:

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

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 (KKT.DB):

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

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:

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

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

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:

$$h_k = \begin{cases} 0, & \text{if } l_k \le x_k \le u_k \\ x_k - l_k, & \text{if } x_k < l_k \\ x_k - u_k, & \text{if } x_k > u_k \end{cases}$$

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

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

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

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:

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

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

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:

$$v_k = \begin{cases} 0, & \text{if } d_k \text{ has correct sign} \\ d_k, & \text{if } d_k \text{ has wrong sign} \end{cases}$$

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

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

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

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

Using the relative errors for all the four conditions listed above 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 Interior-point method routines

2.8.1 Solve LP problem with the interior-point method

Synopsis

```
int lpx_interior(glp_prob *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 displays information about every interior point iteration[4]. This information is sent to the output device 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.

[4]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.

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 cannot start the search because the problem is empty, i.e. has no rows and/or columns.
LPX_E_NOFEAS	the problem has no feasible (primal or dual) 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.

2.8.2 Retrieve status of interior-point solution

Synopsis

```
int glp_ipt_status(glp_prob *lp);
```

Returns

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

GLP_UNDEF	interior-point solution is undefined.
GLP_OPT	interior-point solution is optimal.

2.8.3 Retrieve objective value

Synopsis

```
double glp_ipt_obj_val(glp_prob *lp);
```

Returns

The routine glp_ipt_obj_val returns value of the objective function for interior-point solution.

2.8.4 Retrieve row primal value

Synopsis

```
double glp_ipt_row_prim(glp_prob *lp, int i);
```

Returns

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

2.8.5 Retrieve row dual value

Synopsis

```
double glp_ipt_row_dual(glp_prob *lp, int i);
```

Returns

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

2.8.6 Retrieve column primal value

Synopsis

```
double glp_ipt_col_prim(glp_prob *lp, int j);
```

Returns

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

2.8.7 Retrieve column dual value

Synopsis

```
double glp_ipt_col_dual(glp_prob *lp, int j);
```

Returns

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

2.9 Mixed integer programming routines

2.9.1 Set (change) column kind

Synopsis

```
void glp_set_col_kind(glp_prob *mip, int j, int kind);
```

Description

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

`GLP_CV` continuous variable;
`GLP_IV` integer variable;
`GLP_BV` binary variable.

Setting a column to `GLP_BV` has the same effect as if it were set to `GLP_IV`, its lower bound were set 0, and its upper bound were set to 1.

2.9.2 Retrieve column kind

Synopsis

```
int glp_get_col_kind(glp_prob *mip, int j);
```

Returns

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

`GLP_CV` continuous variable;
`GLP_IV` integer variable;
`GLP_BV` binary variable.

2.9.3 Retrieve number of integer columns

Synopsis

```
int glp_get_num_int(glp_prob *mip);
```

Returns

The routine `glp_get_num_int` returns the number of columns (structural variables), which are marked as integer. Note that this number *does* include binary columns.

2.9.4 Retrieve number of binary columns

Synopsis

```
int glp_get_num_bin(glp_prob *mip);
```

Returns

The routine `glp_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.

2.9.5 Solve MIP problem with the branch-and-cut method

Synopsis

```
int glp_intopt(glp_prob *mip, const glp_iocp *parm);
```

Description

The routine `glp_intopt` is a driver to the MIP solver based on the branch-and-cut method, which is a hybrid of branch-and-bound and cutting plane methods.

If the presolver is disabled (see paragraph "Control parameters" below), on entry to the routine `glp_intopt` the problem object, which the parameter `mip` points to, should contain optimal solution to LP relaxation (it can be obtained, for example, with the routine `glp_simplex`). Otherwise, if the presolver is enabled, it is not necessary.

The MIP solver has a set of control parameters. Values of the control parameters can be passed in the structure `glp_iocp`, which the parameter `parm` points to. For detailed description of this structure see paragraph "Control parameters" below. Before specifying some control parameters the application program should initialize the structure `glp_iocp` by default values of all control parameters using the routine `glp_init_iocp` (see the next subsection). This is needed for backward compatibility, because in the future there may appear new members in the structure `glp_iocp`.

The parameter `parm` can be specified as `NULL`, in which case the solver uses default settings.

Note that the GLPK branch-and-cut solver is not perfect, so it is unable to solve hard or very large scale MIP instances for a reasonable time.

Returns

0	The MIP problem instance has been successfully solved. (This code does *not* necessarily mean that the solver has found optimal solution. It only means that the solution process was successful.)
`GLP_EBOUND`	Unable to start the search, because some double-bounded variables have incorrect bounds or some integer variables have non-integer (fractional) bounds.
`GLP_EROOT`	Unable to start the search, because optimal basis for initial LP relaxation is not provided. (This code may appear only if the presolver is disabled.)
`GLP_ENOPFS`	Unable to start the search, because LP relaxation of the MIP problem instance has no primal feasible solution. (This code may appear only if the presolver is enabled.)
`GLP_ENODFS`	Unable to start the search, because LP relaxation of the MIP problem instance has no dual feasible solution. In other word, this code means that if the LP relaxation has at least one primal feasible solution, its optimal solution is unbounded, so if the MIP problem has at least one integer feasible solution, its (integer) optimal solution is also unbounded. (This code may appear only if the presolver is enabled.)
`GLP_EFAIL`	The search was prematurely terminated due to the solver failure.
`GLP_EMIPGAP`	The search was prematurely terminated, because the relative mip gap tolerance has been reached.
`GLP_ETMLIM`	The search was prematurely terminated, because the time limit has been exceeded.
`GLP_ESTOP`	The search was prematurely terminated by application. (This code may appear only if the advanced solver interface is used.)

Built-in MIP presolver

The branch-and-cut solver has *built-in MIP presolver*. It is a subprogram that transforms the original MIP problem specified in the problem object to an equivalent MIP problem, which may be easier for solving with the branch-and-cut method than the original one. For example, the presolver can remove redundant constraints and variables, whose optimal values are known, perform bound and coefficient reduction, etc. Once the transformed

MIP problem has been solved, the presolver transforms its solution back to corresponding solution of the original problem.

Presolving is an optional feature of the routine `glp_intopt`, and by default it is disabled. In order to enable the MIP presolver, the control parameter `presolve` should be set to `GLP_ON` (see paragraph "Control parameters" below).

Advanced solver interface

The routine `glp_intopt` allows the user to control the branch-and-cut search by passing to the solver a user-defined callback routine. For more details see Chapter "Branch-and-Cut API Routines".

Solver terminal output

Solving a MIP problem may take a long time, so the solver reports some information about best known solutions, which is sent to the terminal. 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 $gap = |xxx - yyy|/(|xxx| + \texttt{DBL\_EPSILON}) \cdot 100\%$ (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.

Control parameters

This paragraph describes all control parameters currently used in the MIP solver. Symbolic names of control parameters are names of corresponding members in the structure `glp_iocp`.

`int msg_lev` (default: `GLP_MSG_ALL`)
Message level for terminal output:
`GLP_MSG_OFF` — no output;
`GLP_MSG_ERR` — error and warning messages only;
`GLP_MSG_ON` — normal output;
`GLP_MSG_ALL` — full output (including informational messages).

`int br_tech` (default: `GLP_BR_DTH`)
Branching technique option:
`GLP_BR_FFV` — first fractional variable;
`GLP_BR_LFV` — last fractional variable;
`GLP_BR_MFV` — most fractional variable;
`GLP_BR_DTH` — heuristic by Driebeck and Tomlin.

`int bt_tech` (default: `GLP_BT_BLB`)
Backtracking technique option:
`GLP_BT_DFS` — depth first search;
`GLP_BT_BFS` — breadth first search;
`GLP_BT_BLB` — best local bound;
`GLP_BT_BPH` — best projection heuristic.

`int pp_tech` (default: `GLP_PP_ALL`)
Preprocessing technique option:
`GLP_PP_NONE` — disable preprocessing;
`GLP_PP_ROOT` — perform preprocessing only on the root level;
`GLP_PP_ALL` — perform preprocessing on all levels.

`int gmi_cuts` (default: `GLP_OFF`)
Gomory's mixed integer cut option:
`GLP_ON` — enable generating Gomory's cuts;
`GLP_OFF` — disable generating Gomory's cuts.

`int mir_cuts` (default: `GLP_OFF`)
Mixed integer rounding (MIR) cut option:
`GLP_ON` — enable generating MIR cuts;
`GLP_OFF` — disable generating MIR cuts.

`int cov_cuts` (default: `GLP_OFF`)
Mixed cover cut option:
`GLP_ON` — enable generating mixed cover cuts;
`GLP_OFF` — disable generating mixed cover cuts.

`int clq_cuts` (default: `GLP_OFF`)
Clique cut option:
`GLP_ON` — enable generating clique cuts;
`GLP_OFF` — disable generating clique cuts.

`double tol_int` (default: `1e-5`)
Absolute tolerance used to check if optimal solution to the current LP relaxation is integer feasible. (Do not change this parameter without detailed understanding its purpose.)

`double tol_obj` (default: `1e-7`)
Relative tolerance used to check if the objective value in optimal solution to the current LP relaxation is not better than in the best known integer feasible solution. (Do not change this parameter without detailed understanding its purpose.)

`double mip_gap` (default: `0.0`)
The relative mip gap tolerance. If the relative mip gap for currently known best integer feasible solution falls below this tolerance, the solver terminates the search. This allows obtainig suboptimal integer feasible solutions if solving the problem to optimality takes too long time.

`int tm_lim` (default: `INT_MAX`)
Searching time limit, in milliseconds.

`int out_frq` (default: `5000`)
Output frequency, in milliseconds. This parameter specifies how frequently the solver sends information about the solution process to the terminal.

`int out_dly` (default: `10000`)
Output delay, in milliseconds. This parameter specifies how long the solver should delay sending information about solution of the current LP relaxation with the simplex method to the terminal.

`void (*cb_func)(glp_tree *tree, void *info)` (default: `NULL`)
Entry point to the user-defined callback routine. `NULL` means the advanced solver interface is not used. For more details see Chapter "Branch-and-Cut API Routines".

`void *cb_info` (default: `NULL`)
Transit pointer passed to the routine `cb_func` (see above).

`int cb_size` (default: `0`)

The number of extra (up to 256) bytes allocated for each node of the branch-and-bound tree to store application-specific data. On creating a node these bytes are initialized by binary zeros.

`int presolve` (default: `GLP_OFF`)

MIP presolver option:
`GLP_ON` — enable using the MIP presolver;
`GLP_OFF` — disable using the MIP presolver.

`int binarize` (default: `GLP_OFF`)

Binarization option (used only if the presolver is enabled):
`GLP_ON` — replace general integer variables by binary ones;
`GLP_OFF` — do not use binarization.

2.9.6 Initialize integer optimizer control parameters

Synopsis

```
void glp_init_iocp(glp_iocp *parm);
```

Description

The routine `glp_init_iocp` initializes control parameters, which are used by the branch-and-cut solver, with default values.

Default values of the control parameters are stored in a `glp_iocp` structure, which the parameter `parm` points to.

2.9.7 Retrieve status of MIP solution

Synopsis

```
int glp_mip_status(glp_prob *mip);
```

Returns

The routine `glp_mip_status` reports the status of a MIP solution found by the MIP solver as follows:

`GLP_UNDEF`	MIP solution is undefined.
`GLP_OPT`	MIP solution is integer optimal.
`GLP_FEAS`	MIP solution is integer feasible, however, its optimality (or non-optimality) has not been proven, perhaps due to premature termination of the search.

GLP_NOFEAS	problem has no integer feasible solution (proven by the solver).

2.9.8 Retrieve objective value

Synopsis

```
double glp_mip_obj_val(glp_prob *mip);
```

Returns

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

2.9.9 Retrieve row value

Synopsis

```
double glp_mip_row_val(glp_prob *mip, int i);
```

Returns

The routine `glp_mip_row_val` returns value of the auxiliary variable associated with `i`-th row for MIP solution.

2.9.10 Retrieve column value

Synopsis

```
double glp_mip_col_val(glp_prob *mip, int j);
```

Returns

The routine `glp_mip_col_val` returns value of the structural variable associated with `j`-th column for MIP solution.

Chapter 3

Utility API routines

3.1 Problem data reading/writing routines

3.1.1 Read problem data in MPS format

Synopsis

```
int glp_read_mps(glp_prob *lp, int fmt, const void *parm,
      const char *fname);
```

Description

The routine `glp_read_mps` reads problem data in MPS format from a text file. (The MPS format is described in Appendix B, page 139.)

The parameter `fmt` specifies the MPS format version as follows:

`GLP_MPS_DECK` fixed (ancient) MPS format;

`GLP_MPS_FILE` free (modern) MPS format.

The parameter `parm` is reserved for use in the future and must be specified as `NULL`.

The character string `fname` specifies a name of the text file to be read in. (If the file name ends with suffix '`.gz`', the file is assumed to be compressed, in which case the routine `glp_read_mps` decompresses it "on the fly".)

Note that before reading data the current content of the problem object is completely erased with the routine `glp_erase_prob`.

Returns

If the operation was successful, the routine `glp_read_mps` returns zero. Otherwise, it prints an error message and returns non-zero.

3.1.2 Write problem data in MPS format

Synopsis

```
int glp_write_mps(glp_prob *lp, int fmt, const void *parm,
      const char *fname);
```

Description

The routine glp_write_mps writes problem data in MPS format to a text file. (The MPS format is described in Appendix B, page 139.)

The parameter fmt specifies the MPS format version as follows:

GLP_MPS_DECK fixed (ancient) MPS format;

GLP_MPS_FILE free (modern) MPS format.

The parameter parm is reserved for use in the future and must be specified as NULL.

The character string fname specifies a name of the text file to be written out. (If the file name ends with suffix '.gz', the file is assumed to be compressed, in which case the routine glp_write_mps performs automatic compression on writing it.)

Returns

If the operation was successful, the routine glp_write_mps returns zero. Otherwise, it prints an error message and returns non-zero.

3.1.3 Read problem data in CPLEX LP format

Synopsis

```
int glp_read_lp(glp_prob *lp, const void *parm,
      const char *fname);
```

Description

The routine glp_read_lp reads problem data in CPLEX LP format from a text file. (The CPLEX LP format is described in Appendix C, page 153.)

The parameter parm is reserved for use in the future and must be specified as NULL.

The character string fname specifies a name of the text file to be read in. (If the file name ends with suffix '.gz', the file is assumed to be compressed, in which case the routine glp_read_lp decompresses it "on the fly".)

Note that before reading data the current content of the problem object is completely erased with the routine glp_erase_prob.

Returns

If the operation was successful, the routine glp_read_lp returns zero. Otherwise, it prints an error message and returns non-zero.

3.1.4 Write problem data in CPLEX LP format

Synopsis

```
int glp_write_lp(glp_prob *lp, const void *parm,
      const char *fname);
```

Description

The routine glp_write_lp writes problem data in CPLEX LP format to a text file. (The CPLEX LP format is described in Appendix C, page 153.)

The parameter parm is reserved for use in the future and must be specified as NULL.

The character string fname specifies a name of the text file to be written out. (If the file name ends with suffix '.gz', the file is assumed to be compressed, in which case the routine glp_write_lp performs automatic compression on writing it.)

Returns

If the operation was successful, the routine glp_write_lp returns zero. Otherwise, it prints an error message and returns non-zero.

3.1.5 Read model in GNU MathProg modeling language

Synopsis

```
glp_prob *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.[1]

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

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 output device and returns `NULL`.

3.2 Problem solution reading/writing routines

3.2.1 Write basic solution in printable format

Synopsis

```
int lpx_print_sol(glp_prob *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.

Returns

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

3.2.2 Write bounds sensitivity information

Synopsis

```
int lpx_print_sens_bnds(glp_prob *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.

3.2.3 Write interior point solution in printable format

Synopsis

```
int lpx_print_ips(glp_prob *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.

3.2.4 Write MIP solution in printable format

Synopsis

```
int lpx_print_mip(glp_prob *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.

3.2.5 Read basic solution from text file

Synopsis

```
int glp_read_sol(glp_prob *lp, const char *fname);
```

Description

The routine `glp_read_sol` reads basic solution from a text file whose name is specified by the parameter `fname` into the problem object.

For the file format see description of the routine `glp_write_sol`.

Returns

On success the routine returns zero, otherwise non-zero.

3.2.6 Write basic solution to text file

Synopsis

```
int glp_write_sol(glp_prob *lp, const char *fname);
```

Description

The routine `glp_write_sol` writes the current basic solution to a text file whose name is specified by the parameter `fname`. This file can be read back with the routine `glp_read_sol`.

Returns

On success the routine returns zero, otherwise non-zero.

File format

The file created by the routine `glp_write_sol` is a plain text file, which contains the following information:

```
m n
p_stat d_stat obj_val
r_stat[1] r_prim[1] r_dual[1]
. . .
r_stat[m] r_prim[m] r_dual[m]
c_stat[1] c_prim[1] c_dual[1]
```

```
. . .
c_stat[n] c_prim[n] c_dual[n]
```

where:
m is the number of rows (auxiliary variables);
n is the number of columns (structural variables);
`p_stat` is the primal status of the basic solution (`GLP_UNDEF` = 1, `GLP_FEAS` = 2, `GLP_INFEAS` = 3, or `GLP_NOFEAS` = 4);
`d_stat` is the dual status of the basic solution (`GLP_UNDEF` = 1, `GLP_FEAS` = 2, `GLP_INFEAS` = 3, or `GLP_NOFEAS` = 4);
`obj_val` is the objective value;
`r_stat[i]`, $i = 1, \ldots, m$, is the status of i-th row (`GLP_BS` = 1, `GLP_NL` = 2, `GLP_NU` = 3, `GLP_NF` = 4, or `GLP_NS` = 5);
`r_prim[i]`, $i = 1, \ldots, m$, is the primal value of i-th row;
`r_dual[i]`, $i = 1, \ldots, m$, is the dual value of i-th row;
`c_stat[j]`, $j = 1, \ldots, n$, is the status of j-th column (`GLP_BS` = 1, `GLP_NL` = 2, `GLP_NU` = 3, `GLP_NF` = 4, or `GLP_NS` = 5);
`c_prim[j]`, $j = 1, \ldots, n$, is the primal value of j-th column;
`c_dual[j]`, $j = 1, \ldots, n$, is the dual value of j-th column.

3.2.7 Read interior-point solution from text file

Synopsis

```
int glp_read_ipt(glp_prob *lp, const char *fname);
```

Description

The routine `glp_read_ipt` reads interior-point solution from a text file whose name is specified by the parameter `fname` into the problem object.

For the file format see description of the routine `glp_write_ipt`.

Returns

On success the routine returns zero, otherwise non-zero.

3.2.8 Write interior-point solution to text file

Synopsis

```
int glp_write_ipt(glp_prob *lp, const char *fname);
```

Description

The routine `glp_write_ipt` writes the current interior-point solution to a text file whose name is specified by the parameter `fname`. This file can be read back with the routine `glp_read_ipt`.

Returns

On success the routine returns zero, otherwise non-zero.

File format

The file created by the routine `glp_write_ipt` is a plain text file, which contains the following information:

```
m n
stat obj_val
r_prim[1] r_dual[1]
. . .
r_prim[m] r_dual[m]
c_prim[1] c_dual[1]
. . .
c_prim[n] c_dual[n]
```

where:
m is the number of rows (auxiliary variables);
n is the number of columns (structural variables);
`stat` is the solution status (`GLP_UNDEF` = 1 or `GLP_OPT` = 5);
`obj_val` is the objective value;
`r_prim[i]`, $i = 1, \ldots, m$, is the primal value of i-th row;
`r_dual[i]`, $i = 1, \ldots, m$, is the dual value of i-th row;
`c_prim[j]`, $j = 1, \ldots, n$, is the primal value of j-th column;
`c_dual[j]`, $j = 1, \ldots, n$, is the dual value of j-th column.

3.2.9 Read MIP solution from text file

Synopsis

```
int glp_read_mip(glp_prob *mip, const char *fname);
```

Description

The routine `glp_read_mip` reads MIP solution from a text file whose name is specified by the parameter `fname` into the problem object.

For the file format see description of the routine `glp_write_mip`.

Returns

On success the routine returns zero, otherwise non-zero.

3.2.10 Write MIP solution to text file

Synopsis

```
int glp_write_mip(glp_prob *mip, const char *fname);
```

Description

The routine `glp_write_mip` writes the current MIP solution to a text file whose name is specified by the parameter `fname`. This file can be read back with the routine `glp_read_mip`.

Returns

On success the routine returns zero, otherwise non-zero.

File format

The file created by the routine `glp_write_sol` is a plain text file, which contains the following information:

```
m n
stat obj_val
r_val[1]
. . .
r_val[m]
c_val[1]
```

```
. . .
c_val[n]
```

where:
m is the number of rows (auxiliary variables);
n is the number of columns (structural variables);
`stat` is the solution status (`GLP_UNDEF` $= 1$, `GLP_FEAS` $= 2$, `GLP_NOFEAS` $=$ 4, or `GLP_OPT` $= 5$);
`obj_val` is the objective value;
`r_val[i]`, $i = 1, \ldots, m$, is the value of i-th row;
`c_val[j]`, $j = 1, \ldots, n$, is the value of j-th column.

Chapter 4

Advanced API Routines

4.1 LP basis and simplex tableau routines

4.1.1 Background

Using vector and matrix notations LP problem (1.1)—(1.3) (see Section 1.1, page 9) can be stated as follows:

minimize (or maximize)

$$z = c^T x_S + c_0 \tag{3.1}$$

subject to linear constraints

$$x_R = A x_S \tag{3.2}$$

and bounds of variables

$$\begin{array}{l} l_R \leq x_R \leq u_R \\ l_S \ \leq x_S \ \leq u_S \end{array} \tag{3.3}$$

where:
$x_R = (x_1, x_2, \dots, x_m)$ is the vector of auxiliary variables;
$x_S = (x_{m+1}, x_{m+2}, \dots, x_{m+n})$ is the vector of structural variables;
z is the objective function;
$c = (c_1, c_2, \dots, c_n)$ is the vector of objective coefficients;
c_0 is the constant term ("shift") of the objective function;
$A = (a_{11}, a_{12}, \dots, a_{mn})$ is the constraint matrix;
$l_R = (l_1, l_2, \dots, l_m)$ is the vector of lower bounds of auxiliary variables;
$u_R = (u_1, u_2, \dots, u_m)$ is the vector of upper bounds of auxiliary variables;

$l_S = (l_{m+1}, l_{m+2}, \dots, l_{m+n})$ is the vector of lower bounds of structural variables;
$u_S = u_{m+1}, u_{m+2}, \dots, u_{m+n}$ is the vector of upper bounds of structural variables.

From the simplex method's standpoint there is no difference between auxiliary and structural variables. This allows combining all these variables into one vector that leads to the following problem statement:

minimize (or maximize)

$$z = (0 \mid c)^T x + c_0 \tag{3.4}$$

subject to linear constraints

$$(I \mid -A)x = 0 \tag{3.5}$$

and bounds of variables

$$l \leq x \leq u \tag{3.6}$$

where:
$x = (x_R \mid x_S)$ is the $(m+n)$-vector of (all) variables;
$(0 \mid c)$ is the $(m+n)$-vector of objective coefficients;[1]
$(I \mid -A)$ is the *augmented* constraint $m \times (m+n)$-matrix;[2]
$l = (l_R \mid l_S)$ is the $(m+n)$-vector of lower bounds of (all) variables;
$u = (u_R \mid u_S)$ is the $(m+n)$-vector of upper bounds of (all) variables.

By definition an *LP basic solution* geometrically is a point in the space of all variables, which is the intersection of planes corresponding to active constraints[3]. The space of all variables has the dimension $m+n$, therefore, to define some basic solution we have to define $m+n$ active constraints. Note that m constraints (3.5) being linearly independent equalities are always active, so remaining n active constraints can be chosen only from bound constraints (3.6).

A variable is called *non-basic*, if its (lower or upper) bound is active, otherwise it is called *basic*. Since, as was said above, exactly n bound constraints must be active, in any basic solution there are always n non-basic

[1] Subvector 0 corresponds to objective coefficients at auxiliary variables.

[2] Note that due to auxiliary variables matrix $(I \mid -A)$ contains the unity submatrix and therefore has full rank. This means, in particular, that the system (3.5) has no linearly dependent constraints.

[3] A constraint is called *active* if in a given point it is satisfied as equality, otherwise it is called *inactive*.

variables and m basic variables. (Note that a free variable also can be non-basic. Although such variable has no bounds, we can think it as the difference between two non-negative variables, which both are non-basic in this case.)

Now consider how to determine numeric values of all variables for a given basic solution.

Let Π be an appropriate permutation matrix of the order $(m+n)$. Then we can write:

$$\begin{pmatrix} x_B \\ x_N \end{pmatrix} = \Pi \begin{pmatrix} x_R \\ x_S \end{pmatrix} = \Pi x, \tag{3.7}$$

where x_B is the vector of basic variables, x_N is the vector of non-basic variables, $x = (x_R \mid x_S)$ is the vector of all variables in the original order. In this case the system of linear constraints (3.5) can be rewritten as follows:

$$(I \mid -A)\Pi^T \Pi x = 0 \quad \Rightarrow \quad (B \mid N) \begin{pmatrix} x_B \\ x_N \end{pmatrix} = 0, \tag{3.8}$$

where

$$(B \mid N) = (I \mid -A)\Pi^T. \tag{3.9}$$

Matrix B is a square non-singular $m \times m$-matrix, which is composed from columns of the augmented constraint matrix corresponding to basic variables. It is called the *basis matrix* or simply the *basis*. Matrix N is a rectangular $m \times n$-matrix, which is composed from columns of the augmented constraint matrix corresponding to non-basic variables.

From (3.8) it follows that:

$$Bx_B + Nx_N = 0, \tag{3.10}$$

therefore,

$$x_B = -B^{-1}Nx_N. \tag{3.11}$$

Thus, the formula (3.11) shows how to determine numeric values of basic variables x_B assuming that non-basic variables x_N are fixed on their active bounds.

The $m \times n$-matrix

$$\Xi = -B^{-1}N, \tag{3.12}$$

which appears in (3.11), is called the *simplex tableau*.[4] It shows how basic variables depend on non-basic variables:

$$x_B = \Xi x_N. \tag{3.13}$$

[4] This definition corresponds to the GLPK implementation.

The system (3.13) is equivalent to the system (3.5) in the sense that they both define the same set of points in the space of (primal) variables, which satisfy to these systems. If, moreover, values of all basic variables satisfy to their bound constraints (3.3), the corresponding basic solution is called *(primal) feasible*, otherwise *(primal) infeasible*. It is understood that any (primal) feasible basic solution satisfy to all constraints (3.2) and (3.3).

The LP theory says that if LP has optimal solution, it has (at least one) basic feasible solution, which corresponds to the optimum. And the most natural way to determine whether a given basic solution is optimal or not is to use the Karush—Kuhn—Tucker optimality conditions.

For the problem statement (3.4)—(3.6) the optimality conditions are the following:[5]

$$(I \mid -A)x = 0 \tag{3.14}$$

$$(I \mid -A)^T\pi + \lambda_l + \lambda_u = \nabla z = (0 \mid c)^T \tag{3.15}$$

$$l \leq x \leq u \tag{3.16}$$

$$\lambda_l \geq 0, \quad \lambda_u \leq 0 \quad \text{(minimization)} \tag{3.17}$$

$$\lambda_l \leq 0, \quad \lambda_u \geq 0 \quad \text{(maximization)} \tag{3.18}$$

$$(\lambda_l)_k(x_k - l_k) = 0, \quad (\lambda_u)_k(x_k - u_k) = 0, \quad k = 1, 2, \dots, m+n \tag{3.19}$$

where: $\pi = (\pi_1, \pi_2, \dots, \pi_m)$ is a m-vector of Lagrange multipliers for equality constraints (3.5); $\lambda_l = [(\lambda_l)_1, (\lambda_l)_2, \dots, (\lambda_l)_n]$ is a n-vector of Lagrange multipliers for lower bound constraints (3.6); $\lambda_u = [(\lambda_u)_1, (\lambda_u)_2, \dots, (\lambda_u)_n]$ is a n-vector of Lagrange multipliers for upper bound constraints (3.6).

Condition (3.14) is the *primal* (original) system of equality constraints (3.5).

Condition (3.15) is the *dual* system of equality constraints. It requires the gradient of the objective function to be a linear combination of normals to the planes defined by constraints of the original problem.

Condition (3.16) is the primal (original) system of bound constraints (3.6).

Condition (3.17) (or (3.18) in case of maximization) is the dual system of bound constraints.

Condition (3.19) is the *complementary slackness condition*. It requires, for each original (auxiliary or structural) variable x_k, that either its (lower or upper) bound must be active, or zero bound of the corresponding Lagrange multiplier ($(\lambda_l)_k$ or $(\lambda_u)_k$) must be active.

[5] These conditions can be appiled to any solution, not only to a basic solution.

In GLPK two multipliers $(\lambda_l)_k$ and $(\lambda_u)_k$ for each primal (original) variable x_k, $k = 1, 2, \ldots, m+n$, are combined into one multiplier:

$$\lambda_k = (\lambda_l)_k + (\lambda_u)_k, \tag{3.20}$$

which is called a *dual variable* for x_k. This *cannot* lead to the ambiguity, because both lower and upper bounds of x_k cannot be active at the same time,[6] so at least one of $(\lambda_l)_k$ and $(\lambda_u)_k$ must be equal to zero, and because these multipliers have different signs, the combined multiplier, which is their sum, uniquely defines each of them.

Using dual variables λ_k the dual system of bound constraints (3.17) and (3.18) can be written in the form of so called *"rule of signs"* as follows:

Original bound constraint	Minimization			Maximization		
	$(\lambda_l)_k$	$(\lambda_u)_k$	$(\lambda_l)_k + (\lambda_u)_k$	$(\lambda_l)_k$	$(\lambda_u)_k$	$(\lambda_l)_k + (\lambda_u)_k$
$-\infty < x_k < +\infty$	$= 0$	$= 0$	$\lambda_k = 0$	$= 0$	$= 0$	$\lambda_k = 0$
$x_k \geq l_k$	≥ 0	$= 0$	$\lambda_k \geq 0$	≤ 0	$= 0$	$\lambda_k \leq 0$
$x_k \leq u_k$	$= 0$	≤ 0	$\lambda_k \leq 0$	$= 0$	≥ 0	$\lambda_k \geq 0$
$l_k \leq x_k \leq u_k$	≥ 0	≤ 0	$-\infty < \lambda_k < +\infty$	≤ 0	≥ 0	$-\infty < \lambda_k < +\infty$
$x_k = l_k = u_k$	≥ 0	≤ 0	$-\infty < \lambda_k < +\infty$	≤ 0	≥ 0	$-\infty < \lambda_k < +\infty$

May note that each primal variable x_k has its dual counterpart λ_k and vice versa. This allows applying the same partition for the vector of dual variables as (3.7):

$$\begin{pmatrix} \lambda_B \\ \lambda_N \end{pmatrix} = \Pi\lambda, \tag{3.21}$$

where λ_B is a vector of dual variables for basic variables x_B, λ_N is a vector of dual variables for non-basic variables x_N.

By definition, bounds of basic variables are inactive constraints, so in any basic solution $\lambda_B = 0$. Corresponding values of dual variables λ_N for non-basic variables x_N can be determined in the following way. From the dual system (3.15) we have:

$$(I \mid -A)^T \pi + \lambda = (0 \mid c)^T, \tag{3.22}$$

so multiplying both sides of (3.22) by matrix Π gives:

$$\Pi(I \mid -A)^T \pi + \Pi\lambda = \Pi(0 \mid c)^T. \tag{3.23}$$

[6] If x_k is a fixed variable, we can think it as double-bounded variable $l_k \leq x_k \leq u_k$, where $l_k = u_k$.

From (3.9) it follows that

$$\Pi(I \mid -A)^T = [(I \mid -A)\Pi^T]^T = (B \mid N)^T. \tag{3.24}$$

Further, we can apply the partition (3.7) also to the vector of objective coefficients (see (3.4)):

$$\begin{pmatrix} c_B \\ c_N \end{pmatrix} = \Pi \begin{pmatrix} 0 \\ c \end{pmatrix}, \tag{3.25}$$

where c_B is a vector of objective coefficients at basic variables, c_N is a vector of objective coefficients at non-basic variables. Now, substituting (3.24), (3.21), and (3.25) into (3.23), leads to:

$$(B \mid N)^T \pi + (\lambda_B \mid \lambda_N)^T = (c_B \mid c_N)^T, \tag{3.26}$$

and transposing both sides of (3.26) gives the system:

$$\begin{pmatrix} B^T \\ N^T \end{pmatrix} \pi + \begin{pmatrix} \lambda_B \\ \lambda_N \end{pmatrix} = \begin{pmatrix} c_B \\ c_T \end{pmatrix}, \tag{3.27}$$

which can be written as follows:

$$\left\{ \begin{array}{l} B^T\pi \,+\, \lambda_B = c_B \\ N^T\pi \,+\, \lambda_N = c_N \end{array} \right. \tag{3.28}$$

Lagrange multipliers $\pi = (\pi_i)$ correspond to equality constraints (3.5) and therefore can have any sign. This allows resolving the first subsystem of (3.28) as follows:[7]

$$\pi = B^{-T}(c_B - \lambda_B) = -B^{-T}\lambda_B + B^{-T}c_B, \tag{3.29}$$

and substitution of π from (3.29) into the second subsystem of (3.28) gives:

$$\lambda_N = -N^T\pi + c_N = N^T B^{-T}\lambda_B + (c_N - N^T B^{-T} c_B). \tag{3.30}$$

The latter system can be written in the following final form:

$$\lambda_N = -\Xi^T \lambda_B + d, \tag{3.31}$$

where Ξ is the simplex tableau (see (3.12)), and

$$d = c_N - N^T B^{-T} c_B = c_N + \Xi^T c_B \tag{3.32}$$

is the vector of so called *reduced costs* of non-basic variables.

[7] B^{-T} means $(B^T)^{-1} = (B^{-1})^T$.

Above it was said that in any basic solution $\lambda_B = 0$, so $\lambda_N = d$ as it follows from (3.31).

The system (3.31) is equivalent to the system (3.15) in the sense that they both define the same set of points in the space of dual variables λ, which satisfy to these systems. If, moreover, values of all dual variables λ_N (i.e. reduced costs d) satisfy to their bound constraints (i.e. to the "rule of signs"; see the table above), the corresponding basic solution is called *dual feasible*, otherwise *dual infeasible*. It is understood that any dual feasible solution satisfy to all constraints (3.15) and (3.17) (or (3.18) in case of maximization).

It can be easily shown that the complementary slackness condition (3.19) is always satisfied for *any* basic solution. Therefore, a basic solution[8] is *optimal* if and only if it is primal and dual feasible, because in this case it satifies to all the optimality conditions (3.14)—(3.19).

The meaning of reduced costs $d = (d_j)$ of non-basic variables can be explained in the following way. From (3.4), (3.7), and (3.25) it follows that:

$$z = c_B^T x_B + c_N^T x_N + c_0. \tag{3.33}$$

Substituting x_B from (3.11) into (3.33) we can eliminate basic variables and express the objective only through non-basic variables:

$$\begin{aligned} z &= c_B^T(-B^{-1}Nx_N) + c_N^T x_N + c_0 = \\ &= (c_N^T - c_B^T B^{-1} N) x_N + c_0 = \\ &= (c_N - N^T B^{-T} c_B)^T x_N + c_0 = \\ &= d^T x_N + c_0. \end{aligned} \tag{3.34}$$

From (3.34) it is seen that reduced cost d_j shows how the objective function z depends on non-basic variable $(x_N)_j$ in the neighborhood of the current basic solution, i.e. while the current basis remains unchanged.

[8]It is assumed that a complete basic solution has the form (x, λ), i.e. it includes primal as well as dual variables.

4.1.2 Check if the basis factorization exists

Synopsis

```
int glp_bf_exists(glp_prob *lp);
```

Returns

If the basis factorization for the current basis associated with the specified problem object exists and therefore is available for computations, the routine `glp_bf_exists` returns non-zero. Otherwise the routine returns zero.

Comments

Let the problem object have m rows and n columns. In GLPK the *basis matrix* B is a square non-singular matrix of the order m, whose columns correspond to basic (auxiliary and/or structural) variables. It is defined by the following main equality:[9]

$$(B \mid N) = (I \mid -A)\Pi^T,$$

where I is the unity matrix of the order m, whose columns correspond to auxiliary variables; A is the original constraint $m \times n$-matrix, whose columns correspond to structural variables; $(I \mid -A)$ is the augmented constraint $m \times (m+n)$-matrix, whose columns correspond to all (auxiliary and structural) variables following in the original order; Π is a permutation matrix of the order $m+n$; and N is a rectangular $m \times n$-matrix, whose columns correspond to non-basic (auxiliary and/or structural) variables.

For various reasons it may be necessary to solve linear systems with matrix B. To provide this possibility the GLPK implementation maintains an invertable form of B (that is, some representation of B^{-1}) called the *basis factorization*, which is an internal component of the problem object. Typically, the basis factorization is computed by the simplex solver, which keeps it in the problem object to be available for other computations.

Should note that any changes in the problem object, which affects the basis matrix (e.g. changing the status of a row or column, changing a basic column of the constraint matrix, removing an active constraint, etc.), invalidates the basis factorization. So before calling any API routine, which uses the basis factorization, the application program must make sure (using the routine `glp_bf_exists`) that the factorization exists and therefore available for computations.

[9]For more details see Subsection 4.1.1, page 82.

4.1.3 Compute the basis factorization

Synopsis

```
int glp_factorize(glp_prob *lp);
```

Description

The routine `glp_factorize` computes the basis factorization for the current basis associated with the specified problem object.[10]

The basis factorization is computed from "scratch" even if it exists, so the application program may use the routine `glp_bf_exists`, and, if the basis factorization already exists, not to call the routine `glp_factorize` to prevent an extra work.

The routine `glp_factorize` *does not* compute components of the basic solution (i.e. primal and dual values).

Returns

0	The basis factorization has been successfully computed.
`GLP_EBADB`	The basis matrix is invalid, because the number of basic (auxiliary and structural) variables is not the same as the number of rows in the problem object.
`GLP_ESING`	The basis matrix is singular within the working precision.
`GLP_ECOND`	The basis matrix is ill-conditioned, i.e. its condition number is too large.

[10] The current basis is defined by the current statuses of rows (auxiliary variables) and columns (structural variables).

4.1.4 Check if the basis factorization has been updated

Synopsis

```
int glp_bf_updated(glp_prob *lp);
```

Returns

If the basis factorization has been just computed from "scratch", the routine `glp_bf_updated` returns zero. Otherwise, if the factorization has been updated at least once, the routine returns non-zero.

Comments

Updating the basis factorization means recomputing it to reflect changes in the basis matrix. For example, on every iteration of the simplex method some column of the current basis matrix is replaced by a new column that gives a new basis matrix corresponding to the adjacent basis. In this case computing the basis factorization for the adjacent basis from "scratch" (as the routine `glp_factorize` does) would be too time-consuming.

On the other hand, since the basis factorization update is a numeric computational procedure, applying it many times may lead to accumulating round-off errors. Therefore the basis is periodically refactorized (reinverted) from "scratch" (with the routine `glp_factorize`) that allows improving its numerical properties.

The routine `glp_bf_updated` allows determining if the basis factorization has been updated at least once since it was computed from "scratch".

4.1.5 Retrieve basis factorization control parameters

Synopsis

```
void glp_get_bfcp(glp_prob *lp, glp_bfcp *parm);
```

Description

The routine `glp_get_bfcp` retrieves control parameters, which are used on computing and updating the basis factorization associated with the specified problem object.

Current values of the control parameters are stored in a `glp_bfcp` structure, which the parameter `parm` points to. For a detailed description of the structure `glp_bfcp` see comments to the routine `glp_set_bfcp` in the next subsection.

Comments

The purpose of the routine `glp_get_bfcp` is two-fold. First, it allows the application program obtaining current values of control parameters used by internal GLPK routines, which compute and update the basis factorization.

The second purpose of this routine is to provide proper values for all fields of the structure `glp_bfcp` in the case when the application program needs to change some control parameters.

4.1.6 Change basis factorization control parameters

Synopsis

```
void glp_set_bfcp(glp_prob *lp, const glp_bfcp *parm);
```

Description

The routine `glp_set_bfcp` changes control parameters, which are used by internal GLPK routines on computing and updating the basis factorization associated with the specified problem object.

New values of the control parameters should be passed in a structure `glp_bfcp`, which the parameter `parm` points to. For a detailed description of the structure `glp_bfcp` see paragraph "Control parameters" below.

The parameter `parm` can be specified as `NULL`, in which case all control parameters are reset to their default values.

Comments

Before changing some control parameters with the routine `glp_set_bfcp` the application program should retrieve current values of all control parameters with the routine `glp_get_bfcp`. This is needed for backward compatibility, because in the future there may appear new members in the structure `glp_bfcp`.

Note that new values of control parameters come into effect on a next computation of the basis factorization, not immediately.

Example

```
glp_prob *lp;
glp_bfcp parm;
. . .
/* retrieve current values of control parameters */
glp_get_bfcp(lp, &parm);
/* change the threshold pivoting tolerance */
parm.piv_tol = 0.05;
/* set new values of control parameters */
glp_set_bfcp(lp, &parm);
. . .
```

Control parameters

This paragraph describes all basis factorization control parameters currently used in the package. Symbolic names of control parameters are names of corresponding members in the structure `glp_bfcp`.

`int type` (default: `GLP_BF_FT`)

Basis factorization type:
`GLP_BF_FT` — LU + Forrest–Tomlin update;
`GLP_BF_BG` — LU + Schur complement + Bartels–Golub update;
`GLP_BF_GR` — LU + Schur complement + Givens rotation update.
In case of `GLP_BF_FT` the update is applied to matrix U, while in cases of `GLP_BF_BG` and `GLP_BF_GR` the update is applied to the Schur complement.

`int lu_size` (default: `0`)

The initial size of the Sparse Vector Area, in non-zeros, used on computing LU-factorization of the basis matrix for the first time. If this parameter is set to 0, the initial SVA size is determined automatically.

`double piv_tol` (default: `0.10`)

Threshold pivoting (Markowitz) tolerance, $0 <$ `piv_tol` < 1, used on computing LU-factorization of the basis matrix. Element u_{ij} of the active submatrix of factor U fits to be pivot if it satisfies to the stability criterion $|u_{ij}| >=$ `piv_tol` $\cdot \max |u_{i*}|$, i.e. if it is not very small in the magnitude among other elements in the same row. Decreasing this parameter may lead to better sparsity at the expense of numerical accuracy, and vice versa.

`int piv_lim` (default: `4`)

This parameter is used on computing LU-factorization of the basis matrix and specifies how many pivot candidates needs to be considered on choosing a pivot element, `piv_lim` ≥ 1. If `piv_lim` candidates have been considered, the pivoting routine prematurely terminates the search with the best candidate found.

`int suhl` (default: `GLP_ON`)

This parameter is used on computing LU-factorization of the basis matrix. Being set to `GLP_ON` it enables applying the following heuristic proposed by Uwe Suhl: if a column of the active submatrix has no eligible pivot candidates, it is no more considered until it becomes a column singleton. In many cases this allows reducing the time needed for pivot searching. To disable this heuristic the parameter `suhl` should be set to `GLP_OFF`.

`double eps_tol` (default: `1e-15`)

Epsilon tolerance, `eps_tol` ≥ 0, used on computing LU-factorization of the basis matrix. If an element of the active submatrix of factor U is less than `eps_tol` in the magnitude, it is replaced by exact zero.

`double max_gro` (default: `1e+10`)

Maximal growth of elements of factor U, `max_gro` ≥ 1, allowable on computing LU-factorization of the basis matrix. If on some elimination step the ratio u_{big}/b_{max} (where u_{big} is the largest magnitude of elements of factor U appeared in its active submatrix during all the factorization process, b_{max} is the largest magnitude of elements of the basis matrix to be factorized), the basis matrix is considered as ill-conditioned.

`int nfs_max` (default: `50`)

Maximal number of additional row-like factors (entries of the eta file), `nfs_max` ≥ 1, which can be added to LU-factorization of the basis matrix on updating it with the Forrest–Tomlin technique. This parameter is used only once, before LU-factorization is computed for the first time, to allocate working arrays. As a rule, each update adds one new factor (however, some updates may need no addition), so this parameter limits the number of updates between refactorizations.

`double upd_tol` (default: `1e-6`)

Update tolerance, $0 <$ `upd_tol` < 1, used on updating LU-factorization of the basis matrix with the Forrest–Tomlin technique. If after updating the magnitude of some diagonal element u_{kk} of factor U becomes less than `upd_tol` $\cdot \max(|u_{k*}|, |u_{*k}|)$, the factorization is considered as inaccurate.

`int nrs_max` (default: `50`)

Maximal number of additional rows and columns, `nrs_max` ≥ 1, which can be added to LU-factorization of the basis matrix on updating it with the Schur complement technique. This parameter is used only once, before LU-factorization is computed for the first time, to allocate working arrays. As a rule, each update adds one new row and column (however, some updates may need no addition), so this parameter limits the number of updates between refactorizations.

`int rs_size` (default: `0`)

The initial size of the Sparse Vector Area, in non-zeros, used to store non-zero elements of additional rows and columns introduced on updating LU-factorization of the basis matrix with the Schur complement technique. If this parameter is set to 0, the initial SVA size is determined automatically.

4.1.7 Retrieve the basis header information

Synopsis

```
int glp_get_bhead(glp_prob *lp, int k);
```

Description

The routine `glp_get_bhead` returns the basis header information for the current basis associated with the specified problem object.

Returns

If basic variable $(x_B)_k$, $1 \leq k \leq m$, is i-th auxiliary variable ($1 \leq i \leq m$), the routine returns i. Otherwise, if $(x_B)_k$ is j-th structural variable ($1 \leq j \leq n$), the routine returns $m+j$. Here m is the number of rows and n is the number of columns in the problem object.

Comments

Sometimes the application program may need to know which original (auxiliary and structural) variable correspond to a given basic variable, or, that is the same, which column of the augmented constraint matrix $(I \mid -A)$ correspond to a given column of the basis matrix B.

The correspondence is defined as follows:[11]

$$\left(\begin{array}{c}x_B\\x_N\end{array}\right) = \Pi \left(\begin{array}{c}x_R\\x_S\end{array}\right) \quad \Leftrightarrow \quad \left(\begin{array}{c}x_R\\x_S\end{array}\right) = \Pi^T \left(\begin{array}{c}x_B\\x_N\end{array}\right),$$

where x_B is the vector of basic variables, x_N is the vector of non-basic variables, x_R is the vector of auxiliary variables following in their original order,[12] x_S is the vector of structural variables following in their original order, Π is a permutation matrix (which is a component of the basis factorization).

Thus, if $(x_B)_k = (x_R)_i$ is i-th auxiliary variable, the routine returns i, and if $(x_B)_k = (x_S)_j$ is j-th structural variable, the routine returns $m + j$, where m is the number of rows in the problem object.

[11] For more details see Subsection 4.1.1, page 82.

[12] The original order of auxiliary and structural variables is defined by the ordinal numbers of corresponding rows and columns in the problem object.

4.1.8 Retrieve row index in the basis header

Synopsis

```
int glp_get_row_bind(glp_prob *lp, int i);
```

Returns

The routine `glp_get_row_bind` returns the index k of basic variable $(x_B)_k$, $1 \leq k \leq m$, which is i-th auxiliary variable (that is, the auxiliary variable corresponding to i-th row), $1 \leq i \leq m$, in the current basis associated with the specified problem object, where m is the number of rows. However, if i-th auxiliary variable is non-basic, the routine returns zero.

Comments

The routine `glp_get_row_bind` is an inverse to the routine `glp_get_bhead`: if `glp_get_bhead`(lp, k) returns i, `glp_get_row_bind`(lp, i) returns k, and vice versa.

4.1.9 Retrieve column index in the basis header

Synopsis

```
int glp_get_col_bind(glp_prob *lp, int j);
```

Returns

The routine `glp_get_col_bind` returns the index k of basic variable $(x_B)_k$, $1 \leq k \leq m$, which is j-th structural variable (that is, the structural variable corresponding to j-th column), $1 \leq j \leq n$, in the current basis associated with the specified problem object, where m is the number of rows, n is the number of columns. However, if j-th structural variable is non-basic, the routine returns zero.

Comments

The routine `glp_get_col_bind` is an inverse to the routine `glp_get_bhead`: if `glp_get_bhead`(lp, k) returns $m + j$, `glp_get_col_bind`(lp, j) returns k, and vice versa.

4.1.10 Perform forward transformation (FTRAN)

Synopsis

```
void glp_ftran(glp_prob *lp, double x[]);
```

Description

The routine `glp_ftran` performs forward transformation (FTRAN), i.e. it solves the system $Bx = b$, where B is the basis matrix associated with the specified problem object, x is the vector of unknowns to be computed, b is the vector of right-hand sides.

On entry to the routine elements of the vector b should be stored in locations `x[1]`, ..., `x[m]`, where m is the number of rows. On exit the routine stores elements of the vector x in the same locations.

4.1.11 Perform backward transformation (BTRAN)

Synopsis

```
void glp_btran(glp_prob *lp, double x[]);
```

Description

The routine `glp_btran` performs backward transformation (BTRAN), i.e. it solves the system $B^Tx = b$, where B^T is a matrix transposed to the basis matrix B associated with the specified problem object, x is the vector of unknowns to be computed, b is the vector of right-hand sides.

On entry to the routine elements of the vector b should be stored in locations `x[1]`, ..., `x[m]`, where m is the number of rows. On exit the routine stores elements of the vector x in the same locations.

4.1.12 Warm up LP basis

Synopsis

```
int lpx_warm_up(glp_prob *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.

4.1.13 Compute row of the simplex tableau

Synopsis

```
int glp_eval_tab_row(glp_prob *lp, int k, int ind[],
      double val[]);
```

Description

The routine `glp_eval_tab_row` computes a row of the current simplex tableau (see Subsection 3.1.1, formula (3.12)), which (row) corresponds to some basic variable specified by the parameter k as follows: if $1 \leq k \leq m$, the basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object. The basis factorization must exist.

The computed row shows how the specified basic variable depends on non-basic variables:

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

where ξ_{i1}, ξ_{i2}, ..., ξ_{in} are elements of the simplex table row, $(x_N)_1$, $(x_N)_2$, ..., $(x_N)_n$ are non-basic (auxiliary and structural) variables.

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

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

Returns

The routine `glp_eval_tab_row` returns `len`, which is the number of non-zero elements in the simplex table row stored in the arrays `ind` and `val`.

Comments

A row of the simplex table is computed as follows. At first, the routine checks that the specified variable x_k is basic and uses the permutation matrix Π (3.7) to determine index i of basic variable $(x_B)_i$, which corresponds to x_k.

The row to be computed is i-th row of the matrix Ξ (3.12), therefore:

$$\xi_i = e_i^T \Xi = -e_i^T B^{-1} N = -(B^{-T} e_i)^T N,$$

where e_i is i-th unity vector. So the routine performs BTRAN to obtain i-th row of the inverse B^{-1}:

$$\varrho_i = B^{-T} e_i,$$

and then computes elements of the simplex table row as inner products:

$$\xi_{ij} = -\varrho_i^T N_j, \quad j = 1, 2, \ldots, n,$$

where N_j is j-th column of matrix N (3.9), which (column) corresponds to non-basic variable $(x_N)_j$. The permutation matrix Π is used again to convert indices j of non-basic columns to original ordinal numbers of auxiliary and structural variables.

4.1.14 Compute column of the simplex tableau

Synopsis

```
int glp_eval_tab_col(glp_prob *lp, int k, int ind[],
      double val[]);
```

Description

The routine `glp_eval_tab_col` computes a column of the current simplex tableau (see Subsection 3.1.1, formula (3.12)), which (column) corresponds to some non-basic variable specified by the parameter k: if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m+1 \leq k \leq m+n$, the non-basic variable is $(k-m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object. The basis factorization must exist.

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

$$\begin{array}{l} (x_B)_1 = \ldots + \xi_{1j}(x_N)_j \ + \ldots \\ (x_B)_2 = \ldots + \xi_{2j}(x_N)_j \ + \ldots \\ \quad . \ . \ . \ . \ . \ . \ . \ . \ . \ . \\ (x_B)_m = \ldots + \xi_{mj}(x_N)_j + \ldots \end{array}$$

where ξ_{1j}, ξ_{2j}, ..., ξ_{mj} are elements of the simplex table column, $(x_B)_1$, $(x_B)_2$, ..., $(x_B)_m$ are basic (auxiliary and structural) variables.

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

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

Returns

The routine `glp_eval_tab_col` returns `len`, which is the number of non-zero elements in the simplex table column stored in the arrays `ind` and `val`.

Comments

A column of the simplex table is computed as follows. At first, the routine checks that the specified variable x_k is non-basic and uses the permutation matrix Π (3.7) to determine index j of non-basic variable $(x_N)_j$, which corresponds to x_k.

The column to be computed is j-th column of the matrix Ξ (3.12), therefore:

$$\Xi_j = \Xi e_j = -B^{-1}Ne_j = -B^{-1}N_j,$$

where e_j is j-th unity vector, N_j is j-th column of matrix N (3.9). So the routine performs FTRAN to transform N_j to the simplex table column $\Xi_j = (\xi_{ij})$ and uses the permutation matrix Π to convert row indices i to original ordinal numbers of auxiliary and structural variables.

4.1.15 Transform explicitly specified row

Synopsis

```
int lpx_transform_row(glp_prob *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:

$$x = a_1x_{m+1} + a_2x_{m+2} + \ldots + a_nx_{m+n}, \qquad (1)$$

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:

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

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`.

4.1.16 Transform explicitly specified column

Synopsis

```
int lpx_transform_col(glp_prob *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:

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

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 $0 \leq$ `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`.

4.1.17 Perform primal ratio test

Synopsis

```
int lpx_prim_ratio_test(glp_prob *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 a 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.

4.1.18 Perform dual ratio test

Synopsis

```
int lpx_dual_ratio_test(glp_prob *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.

4.2 Library environment routines

4.2.1 64-bit integer data type

Some GLPK API routines use 64-bit integer data type, which is declared in the header `glpk.h` as follows:

```
typedef struct { int lo, hi; } glp_long;
```

where `lo` contains low 32 bits, and `hi` contains high 32 bits of 64-bit integer value.[13]

4.2.2 Determine library version

Synopsis

```
const char *glp_version(void);
```

Returns

The routine `glp_version` returns a pointer to a null-terminated character string, which specifies the version of the GLPK library in the form `"X.Y"`, where '`X`' is the major version number, and '`Y`' is the minor version number, for example, `"4.16"`.

Example

```
printf("GLPK version is %s\n", glp_version());
```

4.2.3 Enable/disable terminal output

Synopsis

```
void glp_term_out(int flag);
```

Description

Depending on the parameter flag the routine `glp_term_out` enables or disables terminal output performed by glpk routines:

`GLP_ON` — enable terminal output;
`GLP_OFF` — disable terminal output.

[13] GLPK conforms to ILP32, LLP64, and LP64 programming models, where the built-in type `int` corresponds to 32-bit integers.

4.2.4 Install hook to intercept terminal output

Synopsis

```
void glp_term_hook(int (*func)(void *info, const char *s),
      void *info);
```

Description

The routine `glp_term_hook` installs the user-defined hook routine to intercept all terminal output performed by GLPK routines.

The parameter *func* specifies the user-defined hook routine. It is called from an internal printing routine, which passes to it two parameters: *info* and *s*. The parameter *info* is a transit pointer specified in corresponding call to the routine `glp_term_hook`; it may be used to pass some additional information to the hook routine. The parameter *s* is a pointer to the null terminated character string, which is intended to be written to the terminal. If the hook routine returns zero, the printing routine writes the string *s* to the terminal in a usual way; otherwise, if the hook routine returns non-zero, no terminal output is performed.

To uninstall the hook routine both parameters *func* and *info* should be specified as `NULL`.

Example

```
static int hook(void *info, const char *s)
{     FILE *foo = info;
      fputs(s, foo);
      return 1;
}

int main(void)
{     FILE *foo;
      . . .
      /* redirect terminal output */
      glp_term_hook(hook, foo);
      . . .
      /* resume terminal output */
      glp_term_hook(NULL, NULL);
      . . .
}
```

4.2.5 Get memory usage information

Synopsis

```
void glp_mem_usage(int *count, int *cpeak, glp_long *total,
      glp_long *tpeak);
```

Description

The routine glp_mem_usage reports some information about utilization of the memory by GLPK routines. Information is stored to locations specified by corresponding parameters (see below). Any parameter can be specified as NULL, in which case corresponding information is not stored.

*count is the number of currently allocated memory blocks.

*cpeak is the peak value of *count reached since the initialization of the GLPK library environment.

*total is the total amount, in bytes, of currently allocated memory blocks.

*tpeak is the peak value of *total reached since the initialization of the GLPK library envirionment.

Example

```
glp_mem_usage(&count, NULL, NULL, NULL);
printf("%d memory block(s) are still allocated\n", count);
```

4.2.6 Set memory usage limit

Synopsis

```
void glp_mem_limit(int limit);
```

Description

The routine glp_mem_limit limits the amount of memory available for dynamic allocation (in GLPK routines) to limit megabytes.

4.2.7 Free GLPK library environment

Synopsis

```
void glp_free_env(void);
```

Description

The routine `glp_free_env` frees all resources used by GLPK routines (memory blocks, etc.) which are currently still in use.

Usage notes

Normally the application program does not need to call this routine, because GLPK routines always free all unused resources. However, if the application program even has deleted all problem objects, there will be several memory blocks still allocated for the internal library needs. For some reasons the application program may want GLPK to free this memory, in which case it should call `glp_free_env`.

Note that a call to `glp_free_env` invalidates all problem objects which still exist.

Chapter 5

Branch-and-Cut API Routines

5.1 Introduction

5.1.1 Using the callback routine

The GLPK MIP solver based on the branch-and-cut method allows the application program to control the solution process. This is attained by means of the user-defined callback routine, which is called by the solver at various points of the branch-and-cut algorithm.

The callback routine passed to the MIP solver should be written by the user and has the following specification:[1]

```
void foo_bar(glp_tree *tree, void *info);
```

where `tree` is a pointer to the data structure `glp_tree`, which should be used on subsequent calls to branch-and-cut interface routines, and `info` is a transit pointer passed to the routine `glp_intopt`, which may be used by the application program to pass some external data to the callback routine.

The callback routine is passed to the MIP solver through the control parameter structure `glp_iocp` (see Chapter "Basic API Routines", Section "Mixed integer programming routines", Subsection "Solve MIP problem with the branch-and-cut method") as follows:

[1] The name `foo_bar` used here is a placeholder for the callback routine name.

```
glp_prob *mip;
glp_iocp parm;
. . .
glp_init_iocp(&parm);
. . .
parm.cb_func = foo_bar;
parm.cb_info = ... ;
ret = glp_intopt(mip, &parm);
. . .
```

To determine why it is being called by the MIP solver the callback routine should use the routine `glp_ios_reason` (described in this section below), which returns a code indicating the reason for calling. Depending on the reason the callback routine may perform necessary actions to control the solution process.

The reason codes, which correspond to various point of the branch-and-cut algorithm implemented in the MIP solver, are described in Subsection "..." below.

To ignore calls for reasons, which are not processed by the callback routine, it should just return to the MIP solver doing nothing. For example:

```
void foo_bar(glp_tree *tree, void *info)
{     . . .
      switch (glp_ios_reason(tree))
      {  case GLP_IBRANCH:
            . . .
            break;
         case GLP_ISELECT:
            . . .
            break;
         default:
            /* ignore call for other reasons */
            break;
      }
      return;
}
```

To control the solution process as well as to obtain necessary information the callback routine may use the branch-and-cut API routines described in this chapter. Names of all these routines begin with '`glp_ios_`'.

5.1.2 Branch-and-cut algorithm

This section gives a schematic description of the branch-and-cut algorithm as it is implemented in the GLPK MIP solver.

1. Initialization

Set $L := \{P_0\}$, where L is the *active list* (i.e. the list of active subproblems), P_0 is the original MIP problem to be solved.

Set $\overline{z} := +\infty$ (in case of minimization) or $\overline{z} := -\infty$ (in case of maximization), where $\overline{z}$ is *incumbent value*, i.e. an upper (minimization) or lower (maximization) global bound for z^*, the optimal objective value for P^0.

2. Subproblem selection

If $L = \emptyset$ then GO TO 9.

Select $P \in L$, i.e. make active subproblem P current.

3. Solving LP relaxation

Solve P_{LP}, which is LP relaxation of P.

If P_{LP} has no primal feasible solution then GO TO 8.

Let z^*_{LP} be the optimal objective value for P_{LP}.

If $z^*_{LP} \geq \overline{z}$ (in case of minimization) or $z^*_{LP} \leq \overline{z}$ (in case of maximization) then GO TO 8.

4. Adding "lazy" constraints

Let x^*_{LP} be the optimal solution to P_{LP}.

If there are "lazy" constraints (i.e. essential constraints not included in the original MIP problem P_0), which are violated at the optimal point x^*_{LP}, add them to P, and GO TO 3.

5. Check for integrality

Let x_j be a variable, which is required to be integer, and let $x^*_j \in x^*_{LP}$ be its value in the optimal solution to P_{LP}.

If x^*_j is integral for all integer variables, then a better integer feasible solution is found. Store its components, set $\overline{z} := z^*_{LP}$, and GO TO 8.

6. Adding cutting planes

If there are cutting planes (i.e. valid constraints for P), which are violated at the optimal point x^*_{LP}, add them to P, and GO TO 3.

7. Branching

Select *branching variable* x_j, i.e. a variable, which is required to be integer, and whose value $x^*_j \in x^*_{LP}$ is fractional in the optimal solution to P_{LP}.

Create new subproblem P_D (so called *down branch*), which is identical to the current subproblem P with exception that the upper bound of x_j is replaced by $\lfloor x_j^* \rfloor$. (For example, if $x_j^* = 3.14$, the new upper bound of x_j in the down branch will be $\lfloor 3.14 \rfloor = 3$.)

Create new subproblem P_U (so called *up branch*), which is identical to the current subproblem P with exception that the lower bound of x_j is replaced by $\lceil x_j^* \rceil$. (For example, if $x_j^* = 3.14$, the new lower bound of x_j in the up branch will be $\lceil 3.14 \rceil = 4$.)

Set $L := L \backslash \{P\} \cup \{P_D, P_U\}$, i.e. remove the current subproblem P from the active list and add two new subproblems P_D and P_U to the active list. Then GO TO 2.

8. Pruning

Remove from the active list L all subproblems (including the current one), whose local bound $\widetilde{z}$ is not better than the global bound $\overline{z}$, i.e. set $L := L \backslash \{P\}$ for all P, where $\widetilde{z} \geq \overline{z}$ (in case of minimization) or $\widetilde{z} \leq \overline{z}$ (in case of maximization), and then GO TO 2.

The local bound $\widetilde{z}$ for subproblem P is an lower (minimization) or upper (maximization) bound for integer optimal solution to *this* subproblem (not to the original problem). This bound is local in the sense that only subproblems in the subtree rooted at node P cannot have better integer feasible solutions. Note that the local bound is not necessarily the optimal objective value to LP relaxation P_{LP}.

9. Termination

If $\overline{z} = +\infty$ (in case of minimization) or $\overline{z} = -\infty$ (in case of maximization), the original problem P_0 has no integer feasible solution. Otherwise, the last integer feasible solution stored on step 5 is the integer optimal solution to the original problem P_0. STOP.

5.1.3 The search tree

On the branching step of the branch-and-cut algorithm the current subproblem is divided into two[2] new subproblems, so the set of all subproblems can be represented in the form of a rooted tree, which is called the *search* or *branch-and-bound* tree. An example of the search tree is shown on Fig. 1. Each node of the search tree corresponds to a subproblem, so the terms 'node' and 'subproblem' may be used synonymously.

[2]In more general cases the current subproblem may be divided into more than two subproblems. However, currently such feature is not used in GLPK.

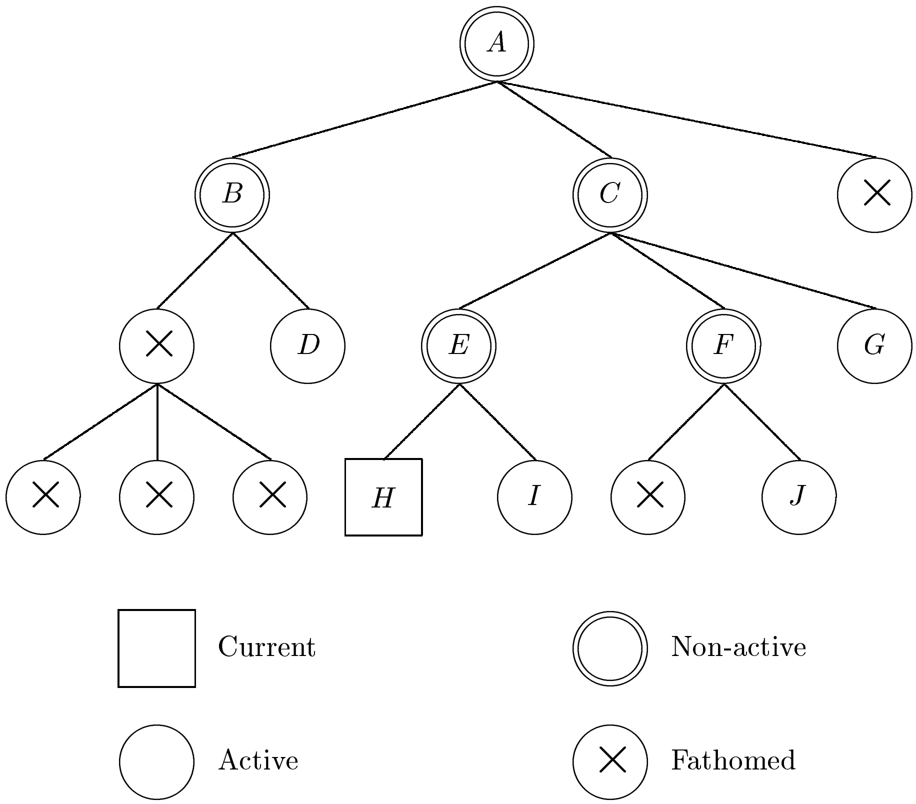

Fig. 1. An example of the search tree

In GLPK each node may have one of the following four statuses:

- *current node* is the active node currently being processed;
- *active node* is a leaf node, which still has to be processed;
- *non-active node* is a node, which has been processed, but not fathomed;
- *fathomed node* is a node, which has been processed and fathomed.

In the data structure representing the search tree GLPK keeps only current, active, and non-active nodes. Once a node has been fathomed, it is removed from the tree data structure.

Being created each node of the search tree is assigned a distinct positive integer called the *subproblem reference number*, which may be used by the application program to specify a particular node of the tree. The root node corresponding to the original problem to be solved is always assigned the reference number 1.

5.1.4 Current subproblem

The current subproblem is a MIP problem corresponding to the current node of the search tree. It is represented as the GLPK problem object (`glp_prob`) that allows the application program using API routines to access its content in the standard way. If the MIP presolver is not used, it is the original problem object passed to the routine `glp_intopt`; otherwise, it is an internal problem object built by the MIP presolver.

Note that the problem object is used by the MIP solver itself during the solution process for various purposes (to solve LP relaxations, to perfom branching, etc.), and even if the MIP presolver is not used, the current content of the problem object may differ from its original content. For example, it may have additional rows, bounds of some rows and columns may be changed, etc. In particular, LP segment of the problem object corresponds to LP relaxation of the current subproblem. However, on exit from the MIP solver the content of the problem object is restored to its original state.

To obtain information from the problem object the application program may use any API routines, which do not change the object. Using API routines, which change the problem object, is restricted to stipulated cases.

5.1.5 The cut pool

The *cut pool* is a set of cutting plane constraints maintained by the MIP solver. It is used by the GLPK cut generation routines and may be used by the application program in the same way, i.e. rather than to add cutting plane constraints directly to the problem object the application program may store them to the cut pool. In the latter case the solver looks through the cut pool, selects efficient constraints, and adds them to the problem object.

5.1.6 Reasons for calling the callback routine

The callback routine may be called by the MIP solver for the following reasons.

Request for subproblem selection

The callback routine is called with the reason code `GLP_ISELECT` if the current subproblem has been fathomed and therefore there is no current subproblem.

In response the callback routine may select some subproblem from the active list and pass its reference number to the solver using the routine `glp_ios_select_node`, in which case the solver continues the search from the specified active subproblem. If no selection is made by the callback routine, the solver uses a backtracking technique specified by the control parameter `bt_tech`.

To explore the active list (i.e. active nodes of the branch-and-bound tree) the callback routine may use the routines `glp_ios_next_node` and `glp_ios_prev_node`.

Request for preprocessing

The callback routine is called with the reason code `GLP_IPREPRO` if the current subproblem has just been selected from the active list and its LP relaxation is not solved yet.

In response the callback routine may perform some preprocessing of the current subproblem like tightening bounds of some variables or removing bounds of some redundant constraints.

Request for row generation

The callback routine is called with the reason code `GLP_IROWGEN` if LP relaxation of the current subproblem has just been solved to optimality and its objective value is better than the best known integer feasible solution.

In response the callback routine may add one or more "lazy" constraints (rows), which are violated by the current optimal solution of LP relaxation, using API routines `glp_add_rows`, `glp_set_row_name`, `glp_set_row_bnds`, and `glp_set_mat_row`, in which case the solver will perform re-optimization of LP relaxation. If there are no violated constraints, the callback routine should just return.

Optimal solution components for LP relaxation can be obtained with API routines `glp_get_obj_val`, `glp_get_row_prim`, `glp_get_row_dual`, `glp_get_col_prim`, and `glp_get_col_dual`.

Request for heuristic solution

The callback routine is called with the reason code `GLP_IHEUR` if LP relaxation of the current subproblem being solved to optimality is integer infeasible (i.e. values of some structural variables of integer kind are fractional), though its objective value is better than the best known integer feasible solution.

In response the callback routine may try applying a primal heuristic to find an integer feasible solution,[3] which is better than the best known one. In case of success the callback routine may store such better solution in the problem object using the routine `glp_ios_heur_sol`.

Request for cut generation

The callback routine is called with the reason code `GLP_ICUTGEN` if LP relaxation of the current subproblem being solved to optimality is integer infeasible (i.e. values of some structural variables of integer kind are fractional), though its objective value is better than the best known integer feasible solution.

In response the callback routine may reformulate the *current* subproblem (before it will be splitted up due to branching) by adding to the problem object one or more *cutting plane constraints*, which cut off the fractional optimal point from the MIP polytope.[4]

Adding cutting plane constraints may be performed in two ways. One way is the same as for the reason code `GLP_IROWGEN` (see above), in which case the callback routine adds new rows corresponding to cutting plane constraints directly to the current subproblem.

The other way is to add cutting plane constraints to the *cut pool*, a set of cutting plane constraints maintained by the solver, rather than directly to the current subproblem. In this case after return from the callback routine the solver looks through the cut pool, selects efficient cutting plane constraints, adds them to the current subproblem, drops other constraints, and then performs re-optimization.

Request for branching

The callback routine is called with the reason code `GLP_IBRANCH` if LP relaxation of the current subproblem being solved to optimality is integer infeasible (i.e. values of some structural variables of integer kind are fractional), though its objective value is better than the best known integer feasible solution.

In response the callback routine may choose some variable suitable for branching (i.e. integer variable, whose value in optimal solution to LP relaxation of the current subproblem is fractional) and pass its ordinal number

[3]Integer feasible to the original MIP problem, not to the current subproblem.

[4]Since these constraints are added to the current subproblem, they may be globally as well as locally valid.

to the solver using the routine glp_ios_branch_upon, in which case the solver splits the current subproblem in two new subproblems and continues the search. If no choice is made by the callback routine, the solver uses a branching technique specified by the control parameter br_tech.

Better integer solution found

The callback routine is called with the reason code GLP_IBINGO if LP relaxation of the current subproblem being solved to optimality is integer feasible (i.e. values of all structural variables of integer kind are integral within the working precision) and its objective value is better than the best known integer feasible solution.

Optimal solution components for LP relaxation can be obtained in the same way as for the reason code GLP_IROWGEN (see above).

Components of the new MIP solution can be obtained with API routines glp_mip_obj_val, glp_mip_row_val, and glp_mip_col_val. Note, however, that due to row/cut generation there may be additional rows in the problem object.

The difference between optimal solution to LP relaxation and corresponding MIP solution is that in the former case some structural variables of integer kind (namely, basic variables) may have values, which are close to nearest integers within the working precision, while in the latter case all such variables have exact integral values.

The reason GLP_IBINGO is intended only for informational purposes, so the callback routine should not modify the problem object in this case.

5.2 Basic routines

5.2.1 Determine reason for calling the callback routine

Synopsis

```
int glp_ios_reason(glp_tree *tree);
```

Returns

The routine `glp_ios_reason` returns a code, which indicates why the user-defined callback routine is being called:

`GLP_ISELECT` — request for subproblem selection;
`GLP_IPREPRO` — request for preprocessing;
`GLP_IROWGEN` — request for row generation;
`GLP_IHEUR` — request for heuristic solution;
`GLP_ICUTGEN` — request for cut generation;
`GLP_IBRANCH` — request for branching;
`GLP_IBINGO` — better integer solution found.

5.2.2 Access the problem object

Synopsis

```
glp_prob *glp_ios_get_prob(glp_tree *tree);
```

Description

The routine `glp_ios_get_prob` can be called from the user-defined callback routine to access the problem object, which is used by the MIP solver. It is the original problem object passed to the routine `glp_intopt` if the MIP presolver is not used; otherwise it is an internal problem object built by the presolver.

Returns

The routine `glp_ios_get_prob` returns a pointer to the problem object used by the MIP solver.

Comments

To obtain various information about the problem instance the callback routine can access the problem object (i.e. the object of type `glp_prob`) using the routine glp_ios_get_prob. It is the original problem object passed to the routine glp_intopt if the MIP presolver is not used; otherwise it is an internal problem object built by the presolver.

5.2.3 Determine additional row attributes

Synopsis

```
void glp_ios_row_attr(glp_tree *tree, int i, glp_attr *attr);
```

Description

The routine glp_ios_row_attr retrieves additional attributes of i-th row of the current subproblem and stores them in the structure glp_attr, which the parameter attr points to.

The structure glp_attr has the following fields:

int level

Subproblem level at which the row was created. (If level = 0, the row was added either to the original problem object passed to the routine glp_intopt or to the root subproblem on generating "lazy" or/and cutting plane constraints.)

int origin

The row origin flag:
GLP_RF_REG — regular constraint;
GLP_RF_LAZY — "lazy" constraint;
GLP_RF_CUT — cutting plane constraint.

int klass

The row class descriptor, which is a number passed to the routine glp_ios_add_row as its third parameter. If the row is a cutting plane constraint generated by the solver, its class may be the following:
GLP_RF_GMI — Gomory's mixed integer cut;
GLP_RF_MIR — mixed integer rounding cut;
GLP_RF_COV — mixed cover cut;
GLP_RF_CLQ — clique cut.

5.2.4 Compute relative MIP gap

Synopsis

```
double glp_ios_mip_gap(glp_tree *tree);
```

Description

The routine glp_ios_mip_gap computes the relative MIP gap (also called *duality gap*) with the following formula:

$$\texttt{gap} = \frac{|\texttt{best\_mip} - \texttt{best\_bnd}|}{|\texttt{best\_mip}| + \texttt{DBL\_EPSILON}}$$

where best_mip is the best integer feasible solution found so far, best_bnd is the best (global) bound. If no integer feasible solution has been found yet, gap is set to DBL_MAX.

Returns

The routine glp_ios_mip_gap returns the relative MIP gap.

Comments

The relative MIP gap is used to measure the quality of the best integer feasible solution found so far, because the optimal solution value z^* for the original MIP problem always lies in the range

$$\texttt{best\_bnd} \leq z^* \leq \texttt{best\_mip}$$

in case of minimization, or in the range

$$\texttt{best\_mip} \leq z^* \leq \texttt{best\_bnd}$$

in case of maximization.

To express the relative MIP gap in percents the value returned by the routine glp_ios_mip_gap should be multiplied by 100%.

5.2.5 Access subproblem application-specific data

Synopsis

```
void *glp_ios_node_data(glp_tree *tree, int p);
```

Description

The routine `glp_ios_node_data` allows the application accessing a memory block allocated for the subproblem (which may be active or inactive), whose reference number is p.

The size of the block is defined by the control parameter `cb_size` passed to the routine `glp_intopt`. The block is initialized by binary zeros on creating corresponding subproblem, and its contents is kept until the subproblem will be removed from the tree.

The application may use these memory blocks to store specific data for each subproblem.

Returns

The routine `glp_ios_node_data` returns a pointer to the memory block for the specified subproblem. Note that if `cb_size` $= 0$, the routine returns a null pointer.

5.2.6 Select subproblem to continue the search

Synopsis

```
void glp_ios_select_node(glp_tree *tree, int p);
```

Description

The routine `glp_ios_select_node` can be called from the user-defined callback routine in response to the reason `GLP_ISELECT` to select an active subproblem, whose reference number is p. The search will be continued from the subproblem selected.

5.2.7 Provide solution found by heuristic

Synopsis

```
int glp_ios_heur_sol(glp_tree *tree, const double x[]);
```

Description

The routine `glp_ios_heur_sol` can be called from the user-defined callback routine in response to the reason `GLP_IHEUR` to provide an integer feasible solution found by a primal heuristic.

Primal values of *all* variables (columns) found by the heuristic should be placed in locations $x[1]$, ..., $x[n]$, where n is the number of columns in the original problem object. Note that the routine `glp_ios_heur_sol` does *not* check primal feasibility of the solution provided.

Using the solution passed in the array x the routine computes value of the objective function. If the objective value is better than the best known integer feasible solution, the routine computes values of auxiliary variables (rows) and stores all solution components in the problem object.

Returns

If the provided solution is accepted, the routine `glp_ios_heur_sol` returns zero. Otherwise, if the provided solution is rejected, the routine returns non-zero.

5.2.8 Check if can branch upon specified variable

Synopsis

```
int glp_ios_can_branch(glp_tree *tree, int j);
```

Returns

If j-th variable (column) can be used to branch upon, the routine returns non-zero, otherwise zero.

5.2.9 Choose variable to branch upon

Synopsis

```
void glp_ios_branch_upon(glp_tree *tree, int j, int sel);
```

Description

The routine `glp_ios_branch_upon` can be called from the user-defined callback routine in response to the reason `GLP_IBRANCH` to choose a branching variable, whose ordinal number is j. Should note that only variables, for which the routine `glp_ios_can_branch` returns non-zero, can be used to branch upon.

The parameter `sel` is a flag that indicates which branch (subproblem) should be selected next to continue the search:

`'D'` — down branch;
`'U'` — up branch;
`'N'` — none of them.

Comments

On branching the solver removes the current active subproblem from the active list and creates two new subproblems (*down-* and *up-branches*), which are added to the end of the active list. Note that the down-branch is created before the up-branch, so the last active subproblem will be the up-branch.

The down- and up-branches are identical to the current subproblem with exception that in the down-branch the upper bound of x_j, the variable chosen to branch upon, is replaced by $\lfloor x_j^* \rfloor$, while in the up-branch the lower bound of x_j is replaced by $\lceil x_j^* \rceil$, where x_j^* is the value of x_j in optimal solution to LP relaxation of the current subproblem. For example, if $x_j^* = 3.14$, the new upper bound of x_j in the down-branch is $\lfloor 3.14 \rfloor = 3$, and the new lower bound in the up-branch is $\lceil 3.14 \rceil = 4$.)

Additionally the callback routine may select either down- or up-branch, from which the solver will continue the search. If none of the branches is selected, a general selection technique will be used.

5.2.10 Terminate the solution process

Synopsis

```
void glp_ios_terminate(glp_tree *tree);
```

Description

The routine `glp_ios_terminate` sets a flag indicating that the MIP solver should prematurely terminate the search.

5.3 The search tree exploring routines

5.3.1 Determine size of the search tree

Synopsis

```
void glp_ios_tree_size(glp_tree *tree, int *a_cnt, int *n_cnt,
      int *t_cnt);
```

Description

The routine `glp_ios_tree_size` stores the following three counts which characterize the current size of the search tree:

`a_cnt` is the current number of active nodes, i.e. the current size of the active list;

`n_cnt` is the current number of all (active and inactive) nodes;

`t_cnt` is the total number of nodes including those which have been already removed from the tree. This count is increased whenever a new node appears in the tree and never decreased.

If some of the parameters `a_cnt`, `n_cnt`, `t_cnt` is a null pointer, the corresponding count is not stored.

5.3.2 Determine current active subproblem

Synopsis

```
int glp_ios_curr_node(glp_tree *tree);
```

Returns

The routine `glp_ios_curr_node` returns the reference number of the current active subproblem. However, if the current subproblem does not exist, the routine returns zero.

5.3.3 Determine next active subproblem

Synopsis

```
int glp_ios_next_node(glp_tree *tree, int p);
```

Returns

If the parameter p is zero, the routine `glp_ios_next_node` returns the reference number of the first active subproblem. However, if the tree is empty, zero is returned.

If the parameter p is not zero, it must specify the reference number of some active subproblem, in which case the routine returns the reference number of the next active subproblem. However, if there is no next active subproblem in the list, zero is returned.

All subproblems in the active list are ordered chronologically, i.e. subproblem A precedes subproblem B if A was created before B.

5.3.4 Determine previous active subproblem

Synopsis

```
int glp_ios_prev_node(glp_tree *tree, int p);
```

Returns

If the parameter p is zero, the routine `glp_ios_prev_node` returns the reference number of the last active subproblem. However, if the tree is empty, zero is returned.

If the parameter p is not zero, it must specify the reference number of some active subproblem, in which case the routine returns the reference number of the previous active subproblem. However, if there is no previous active subproblem in the list, zero is returned.

All subproblems in the active list are ordered chronologically, i.e. subproblem A precedes subproblem B if A was created before B.

5.3.5 Determine parent subproblem

Synopsis

```
int glp_ios_up_node(glp_tree *tree, int p);
```

Returns

The parameter p must specify the reference number of some (active or inactive) subproblem, in which case the routine `iet_get_up_node` returns the reference number of its parent subproblem. However, if the specified subproblem is the root of the tree and, therefore, has no parent, the routine returns zero.

5.3.6 Determine subproblem level

Synopsis

```
int glp_ios_node_level(glp_tree *tree, int p);
```

Returns

The routine `glp_ios_node_level` returns the level of the subproblem, whose reference number is p, in the branch-and-bound tree. (The root subproblem has level 0, and the level of any other subproblem is the level of its parent plus one.)

5.3.7 Determine subproblem local bound

Synopsis

```
double glp_ios_node_bound(glp_tree *tree, int p);
```

Returns

The routine `glp_ios_node_bound` returns the local bound for (active or inactive) subproblem, whose reference number is p.

Comments

The local bound for subproblem p is an lower (minimization) or upper (maximization) bound for integer optimal solution to *this* subproblem (not to the original problem). This bound is local in the sense that only subproblems in the subtree rooted at node p cannot have better integer feasible solutions.

On creating a subproblem (due to the branching step) its local bound is inherited from its parent and then may get only stronger (never weaker). For the root subproblem its local bound is initially set to `-DBL_MAX` (minimization) or `+DBL_MAX` (maximization) and then improved as the root LP relaxation has been solved.

Note that the local bound is not necessarily the optimal objective value to corresponding LP relaxation.

5.3.8 Find active subproblem with best local bound

Synopsis

```
int glp_ios_best_node(glp_tree *tree);
```

Returns

The routine `glp_ios_best_node` returns the reference number of the active subproblem, whose local bound is best (i.e. smallest in case of minimization or largest in case of maximization). However, if the tree is empty, the routine returns zero.

Comments

The best local bound is an lower (minimization) or upper (maximization) bound for integer optimal solution to the original MIP problem.

5.4 The cut pool routines

5.4.1 Determine current size of the cut pool

Synopsis

```
int glp_ios_pool_size(glp_tree *tree);
```

Returns

The routine `glp_ios_pool_size` returns the current size of the cut pool, that is, the number of cutting plane constraints currently added to it.

5.4.2 Add row (constraint) to the cut pool

Synopsis

```
int glp_ios_add_row(glp_tree *tree, const char *name,
      int klass, int flags, int len, const int ind[],
      const double val[], int type, double rhs);
```

Description

The routine `glp_ios_add_row` adds specified row (cutting plane constraint) to the cut pool.

The cutting plane constraint should have the following format:

$$\sum_{j\in J} a_j x_j \left\{ \begin{array}{c} \geq \\ \leq \end{array} \right\} b,$$

where J is a set of indices (ordinal numbers) of structural variables, a_j are constraint coefficients, x_j are structural variables, b is the right-hand side.

The parameter `name` specifies a symbolic name assigned to the constraint (1 up to 255 characters). If it is `NULL` or an empty string, no name is assigned.

The parameter `klass` specifies the constraint class, which must be either zero or a number in the range from 101 to 200. The application may use this attribute to distinguish between cutting plane constraints of different classes.[5]

The parameter `flags` currently is not used and must be zero.

[5]Constraint classes numbered from 1 to 100 are reserved for GLPK cutting plane generators.

Ordinal numbers of structural variables (i.e. column indices) $j \in J$ and numerical values of corresponding constraint coefficients a_j must be placed in locations `ind[1]`, ..., `ind[len]` and `val[1]`, ..., `val[len]`, respectively, where `len` $= |J|$ is the number of constraint coefficients, $0 \leq$ `len` $\leq n$, and n is the number of columns in the problem object. Coefficients with identical column indices are not allowed. Zero coefficients are allowed, however, they are ignored.

The parameter `type` specifies the constraint type as follows:

`GLP_LO` means inequality constraint $\Sigma a_j x_j \geq b$;

`GLP_UP` means inequality constraint $\Sigma a_j x_j \leq b$;

The parameter `rhs` specifies the right-hand side b.

All cutting plane constraints in the cut pool are identified by their ordinal numbers 1, 2, ..., *size*, where *size* is the current size of the cut pool. New constraints are always added to the end of the cut pool, thus, ordinal numbers of previously added constraints are not changed.

Returns

The routine `glp_ios_add_row` returns the ordinal number of the cutting plane constraint added, which is the new size of the cut pool.

Example

```
/* generate triangle cutting plane:
   x[i] + x[j] + x[k] <= 1 */
. . .
/* add the constraint to the cut pool */
ind[1] = i, val[1] = 1.0;
ind[2] = j, val[2] = 1.0;
ind[3] = k, val[3] = 1.0;
glp_ios_add_row(tree, NULL, TRIANGLE_CUT, 0, 3, ind, val,
                GLP_UP, 1.0);
```

Comments

Cutting plane constraints added to the cut pool are intended to be then added only to the *current* subproblem, so these constraints can be globally as well as locally valid. However, adding a constraint to the cut pool does not mean that it will be added to the current subproblem—it depends on the solver's decision: if the constraint seems to be efficient, it is moved from the pool to the current subproblem, otherwise it is simply dropped.[6]

Normally, every time the callback routine is called for cut generation, the cut pool is empty. On the other hand, the solver itself can generate cutting plane constraints (like Gomory's or mixed integer rounding cuts), in which case the cut pool may be non-empty.

5.4.3 Delete row (constraint) from the cut pool

Synopsis

```
void glp_ios_del_row(glp_tree *tree, int i);
```

Description

The routine `glp_ios_del_row` deletes i-th row (cutting plane constraint) from the cut pool, where $1 \leq i \leq size$ is the ordinal number of the constraint in the pool, $size$ is the current size of the cut pool.

Note that deleting a constraint from the cut pool leads to changing ordinal numbers of other constraints remaining in the pool. New ordinal numbers of the remaining constraints are assigned under assumption that the original order of constraints is not changed. Let, for example, there be four constraints a, b, c and d in the cut pool, which have ordinal numbers 1, 2, 3 and 4, respectively, and let constraint b have been deleted. Then after deletion the remaining constraint a, c and d are assigned new ordinal numbers 1, 2 and 3, respectively.

To find the constraint to be deleted the routine `glp_ios_del_row` uses "smart" linear search, so it is recommended to remove constraints in a natural or reverse order and avoid removing them in a random order.

Example

```
/* keep first 10 constraints in the cut pool and remove other
   constraints */
while (glp_ios_pool_size(tree) > 10)
   glp_ios_del_row(tree, glp_ios_pool_size(tree));
```

[6]Globally valid constraints could be saved and then re-used for other subproblems, but currently such feature is not implemented.

5.4.4 Delete all rows (constraints) from the cut pool

Synopsis

```
void glp_ios_clear_pool(glp_tree *tree);
```

Description

The routine `glp_ios_clear_pool` makes the cut pool empty deleting all existing rows (cutting plane constraints) from it.

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 the command `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 9).

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

[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".

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 $sxx\texttt{E}syy$, 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.

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

[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).

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.

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:

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

[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.

'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 141). 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.

---

[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.

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.

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	$b_i + \|r_i\|$
`L`	$+$ or $-$	$b_i - \|r_i\|$	b_i
`E`	$+$	b_i	$b_i + \|r_i\|$
`E`	$-$	$b_i - \|r_i\|$	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");

[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 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 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*.

[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.

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

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

subject to linear constraints

$$\begin{array}{lcl}
yield &=& bin_1 + bin_2 + bin_3 + bin_4 + bin_5 + al + si \\
FE &=& .15\ bin_1 + .04\ bin_2 + .02\ bin_3 + .04\ bin_4 + .02\ bin_5 + .01\ al + .03\ si \\
CU &=& .03\ bin_1 + .05\ bin_2 + .08\ bin_3 + .02\ bin_4 + .06\ bin_5 + .01\ al \\
MN &=& .02\ bin_1 + .04\ bin_2 + .01\ bin_3 + .02\ bin_4 + .02\ bin_5 \\
MG &=& .02\ bin_1 + .03\ bin_2 + .01\ bin_5 \\
AL &=& .70\ bin_1 + .75\ bin_2 + .80\ bin_3 + .75\ bin_4 + .80\ bin_5 + .97\ al \\
SI &=& .02\ bin_1 + .06\ bin_2 + .08\ bin_3 + .12\ bin_4 + .02\ bin_5 + .01\ al + .97\ si
\end{array}$$

and bounds of (auxiliary and structural) variables

$$\begin{array}{rlcrlcl}
 & yield & = 2000 & 0 \leq & bin_1 & \leq & 200 \\
-\infty < & FE & \leq 60 & 0 \leq & bin_2 & \leq & 2500 \\
-\infty < & CU & \leq 100 & 400 \leq & bin_3 & \leq & 800 \\
-\infty < & MN & \leq 40 & 100 \leq & bin_4 & \leq & 700 \\
-\infty < & MG & \leq 30 & 0 \leq & bin_5 & \leq & 1500 \\
1500 \leq & AL & < +\infty & 0 \leq & al & < & +\infty \\
250 \leq & SI & \leq 300 & 0 \leq & si & < & +\infty
\end{array}$$

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 139).

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 declaring integer columns using three additional types of bounds, which are specified in the field 1 of data cards in the BOUNDS section (see Section B.8, page 144):

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

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

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

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 146:

```
*00000000111111111122222222223333333333444444444455555555556
*23456789012345678901234567890123456789012345678901234567890
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 9).

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.

[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.

Keywords which 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 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`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:

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

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

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:

$$\left\{\begin{matrix}\texttt{minimize}\\ \texttt{maximize}\end{matrix}\right\}\ f : s\ c\ x\ s\ c\ x\ \dots\ s\ c\ x$$

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.

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:

$$\begin{array}{l}\texttt{subject to}\\ \mathit{constraint}_1\\ \mathit{constraint}_2\\ \quad\ldots\\ \mathit{constraint}_m\end{array}$$

where $\mathit{constraint}_i, i = 1, \ldots, m$, 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 $\mathit{definition}_k, k = 1, \ldots, p$, 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

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

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.

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`'.

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

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.

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 146.

```
\* 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

`--mps`	read LP/MIP problem in fixed MPS format
`--freemps`	read LP/MIP problem in free MPS format (default)
`--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)
`--memlim` *nnn*	limit available memory to *nnn* Megabytes
`--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
`--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

LP basis factorization option

`--luf`	LU + Forrest–Tomlin update (faster, less stable; default)
`--cbg`	LU + Schur complement + Bartels–Golub update (slower, more stable)
`--cbg`	LU + Schur complement + Givens rotation update (slower, more stable)

Options specific to simplex method

`--std`	use standard initial basis of all slacks
`--adv`	use advanced initial basis (default)
`--bib`	use Bixby's initial basis

`--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
`--exact`	use simplex method based on exact arithmetic
`--xcheck`	check final basis using exact arithmetic
`--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
`--intopt`	use advanced MIP solver (enables MIP presolving)
`--binarize`	replace general integer variables by binary ones (assumes `--intopt`)
`--cover`	generate mixed cover cuts
`--clique`	generate clique cuts
`--gomory`	generate Gomory's mixed integer cuts
`--mir`	generate MIR (mixed integer rounding) cuts
`--cuts`	generate cuts of all classes above (assumes `--intopt`)

For description of the MPS format see Appendix B, page 139.

For description of the CPLEX LP format see Appendix C, page 153.

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

GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `<http://www.gnu.org/licenses/>`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `<http://www.gnu.org/philosophy/why-not-lgpl.html>`.