


GNU Linear Programming Kit

Reference Manual

for GLPK Version 5.0

(December 2020)




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

Copyright © 2000-2020 Free Software Foundation, Inc.

Author: Andrew Makhorin <mao@gnu.org>.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation.

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 the *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

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

and bounds of variables

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

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

1.2 MIP problem

Mixed integer linear programming (MIP) problem is an 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

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

where all variables are non-negative

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

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

maximize

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

subject to

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

and bounds of variables

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

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]

```
/* 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 */
```

[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.

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

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

For more details about configuration options of the GLPK package see Appendix **??**, page **??**.

Chapter 2

Basic API Routines

2.1 General conventions

2.1.1 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>
```

2.1.2 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.

2.1.3 Thread safety

The standard version of GLPK API is *not* thread safe and therefore should not be used in multi-threaded programs.

2.1.4 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.

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.

2.2 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 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 the following segments:

— problem segment,

— basis segment,

— interior-point segment, and

— MIP segment.

2.2.1 Problem segment

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

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 **??**, page **??**). 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.

2.2.2 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 simplex-based solvers 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.

2.2.3 Interior-point segment

The *interior-point segment* contains interior-point solution components, which include the solution status, and primal and dual values of all auxiliary and structural variables.

2.2.4 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, so if the search has been terminated, it cannot be continued.

2.3 Problem creating and modifying routines

2.3.1 glp_create_prob — 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.

2.3.2 glp_set_prob_name — assign (change) problem name

Synopsis

```
void glp_set_prob_name(glp_prob *P, 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.

2.3.3 glp_set_obj_name — assign (change) objective function name

Synopsis

```
void glp_set_obj_name(glp_prob *P, 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.

2.3.4 glp_set_obj_dir — set (change) optimization direction flag

Synopsis

```
void glp_set_obj_dir(glp_prob *P, 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` means minimization;

`GLP_MAX` means maximization.

Note that by default the problem is minimization.

2.3.5 glp_add_rows — add new rows to problem object

Synopsis

```
int glp_add_rows(glp_prob *P, 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.

Each new row becomes a non-active (non-binding) constraint, i.e. the corresponding auxiliary variable is marked as basic.

If the basis factorization exists, adding row(s) invalidates it.

Returns

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

2.3.6 glp_add_cols — add new columns to problem object

Synopsis

```
int glp_add_cols(glp_prob *P, 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.

Each new column is marked as non-basic, i.e. zero value of the corresponding structural variable becomes an active (binding) bound.

If the basis factorization exists, it remains valid.

Returns

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

2.3.7 glp_set_row_name — assign (change) row name

Synopsis

```
void glp_set_row_name(glp_prob *P, 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.

2.3.8 glp_set_col_name — assign (change) column name

Synopsis

```
void glp_set_col_name(glp_prob *P, 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.

2.3.9 glp_set_row_bnds — set (change) row bounds

Synopsis

```
void glp_set_row_bnds(glp_prob *P, 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`	$lb \le x < +\infty$	Variable with lower bound
`GLP_UP`	$-\infty < x \le ub$	Variable with upper bound
`GLP_DB`	$lb \le x \le ub$	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`.

2.3.10 glp_set_col_bnds — set (change) column bounds

Synopsis

```
void glp_set_col_bnds(glp_prob *P, 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`	$lb \le x < +\infty$	Variable with lower bound
`GLP_UP`	$-\infty < x \le ub$	Variable with upper bound
`GLP_DB`	$lb \le x \le ub$	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.

2.3.11 glp_set_obj_coef — set (change) objective coefficient or constant term

Synopsis

```
void glp_set_obj_coef(glp_prob *P, 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.

2.3.12 glp_set_mat_row — set (replace) row of the constraint matrix

Synopsis

```
void glp_set_mat_row(glp_prob *P, 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 should 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`.

Note

If the basis factorization exists and changing the row changes coefficients at basic column(s), the factorization is invalidated.

2.3.13 glp_set_mat_col — set (replace) column of the constraint matrix

Synopsis

```
void glp_set_mat_col(glp_prob *P, 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 should 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`.

Note

If the basis factorization exists, changing the column corresponding to a basic structural variable invalidates it.

2.3.14 glp_load_matrix — load (replace) the whole constraint matrix

Synopsis

```
void glp_load_matrix(glp_prob *P, 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) should 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`.

Note

If the basis factorization exists, this operation invalidates it.

2.3.15 glp_check_dup — check for duplicate elements in sparse matrix

Synopsis

```
int glp_check_dup(int m, int n, int ne, const int ia[], const int ja[]);
```

Description

The routine `glp_check_dup checks` for duplicate elements (that is, elements with identical indices) in a sparse matrix specified in the coordinate format.

The parameters m and n specifies, respectively, the number of rows and columns in the matrix, $m \geq 0$, $n \geq 0$.

The parameter ne specifies the number of (structurally) non-zero elements in the matrix, $ne \geq 0$.

Elements of the matrix are specified as doublets $(ia[k], ja[k])$ for $k = 1, \ldots, ne$, where $ia[k]$ is a row index, $ja[k]$ is a column index.

The routine `glp_check_dup` can be used prior to a call to the routine `glp_load_matrix` to check that the constraint matrix to be loaded has no duplicate elements.

Returns

0	the matrix representation is correct;
$-k$	indices $ia[k]$ or/and $ja[k]$ are out of range;
$+k$	element $(ia[k], ja[k])$ is duplicate.

2.3.16 glp_sort_matrix — sort elements of the constraint matrix

Synopsis

```
void glp_sort_matrix(glp_prob *P);
```

Description

The routine `glp_sort_matrix` sorts elements of the constraint matrix by rebuilding its row and column linked lists.

On exit from the routine the constraint matrix is not changed, however, elements in the row linked lists become ordered by ascending column indices, and the elements in the column linked lists become ordered by ascending row indices.

2.3.17 glp_del_rows — delete rows from problem object

Synopsis

```
void glp_del_rows(glp_prob *P, 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.

If the basis factorization exists, deleting active (binding) rows, i.e. whose auxiliary variables are marked as non-basic, invalidates it.

2.3.18 glp_del_cols — delete columns from problem object

Synopsis

```
void glp_del_cols(glp_prob *P, 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.

If the basis factorization exists, deleting basic columns invalidates it.

2.3.19 glp_copy_prob — copy problem object content

Synopsis

```
void glp_copy_prob(glp_prob *dest, glp_prob *prob, int names);
```

Description

The routine `glp_copy_prob` copies the content of the problem object `prob` to the problem object `dest`.

The parameter `names` is a flag. If it is `GLP_ON`, the routine also copies all symbolic names; otherwise, if it is `GLP_OFF`, no symbolic names are copied.

2.3.20 glp_erase_prob — erase problem object content

Synopsis

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

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 pointer to the problem object remains valid.

2.3.21 glp_delete_prob — delete problem object

Synopsis

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

Description

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

2.4 Problem retrieving routines

2.4.1 glp_get_prob_name — retrieve problem name

Synopsis

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

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`.

2.4.2 glp_get_obj_name — retrieve objective function name

Synopsis

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

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`.

2.4.3 glp_get_obj_dir — retrieve optimization direction flag

Synopsis

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

Returns

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

`GLP_MIN` means minimization;

`GLP_MAX` means maximization.

2.4.4 glp_get_num_rows — retrieve number of rows

Synopsis

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

Returns

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

2.4.5 glp_get_num_cols — retrieve number of columns

Synopsis

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

Returns

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

2.4.6 glp_get_row_name — retrieve row name

Synopsis

```
const char *glp_get_row_name(glp_prob *P, 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`.

2.4.7 glp_get_col_name — retrieve column name

Synopsis

```
const char *glp_get_col_name(glp_prob *P, 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`.

2.4.8 glp_get_row_type — retrieve row type

Synopsis

```
int glp_get_row_type(glp_prob *P, 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.

2.4.9 glp_get_row_lb — retrieve row lower bound

Synopsis

```
double glp_get_row_lb(glp_prob *P, 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`.

2.4.10 glp_get_row_ub — retrieve row upper bound

Synopsis

```
double glp_get_row_ub(glp_prob *P, 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`.

2.4.11 glp_get_col_type — retrieve column type

Synopsis

```
int glp_get_col_type(glp_prob *P, 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.

2.4.12 glp_get_col_lb — retrieve column lower bound

Synopsis

```
double glp_get_col_lb(glp_prob *P, 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`.

2.4.13 glp_get_col_ub — retrieve column upper bound

Synopsis

```
double glp_get_col_ub(glp_prob *P, 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`.

2.4.14 glp_get_obj_coef — retrieve objective coefficient or constant term

Synopsis

```
double glp_get_obj_coef(glp_prob *P, 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.

2.4.15 glp_get_num_nz — retrieve number of constraint coefficients

Synopsis

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

Returns

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

2.4.16 glp_get_mat_row — retrieve row of the constraint matrix

Synopsis

```
int glp_get_mat_row(glp_prob *P, 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.

2.4.17 glp_get_mat_col — retrieve column of the constraint matrix

Synopsis

```
int glp_get_mat_col(glp_prob *P, 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.

2.5 Row and column searching routines

Sometimes it may be necessary to find rows and/or columns by their names (assigned with the routines `glp_set_row_name` and `glp_set_col_name`). Though a particular row/column can be found by its name using simple enumeration of all rows/columns, in case of large instances such a *linear* search may take too long time.

To significantly reduce the search time the application program may create the row/column name index, which is an auxiliary data structure implementing a *binary* search. Even in worst cases the search takes logarithmic time, i.e. the time needed to find a particular row (or column) by its name is $O(\log_2 m)$ (or $O(\log_2 n)$), where m and n are, resp., the number of rows and columns in the problem object.

It is important to note that:

1. On creating the problem object with the routine `glp_create_prob` the name index is *not* created.

2. The name index can be created (destroyed) at any time with the routine `glp_create_index` (`glp_delete_index`). Having been created the name index becomes part of the corresponding problem object.

3. The time taken to create the name index is $O[(m+n)\log_2(m+n)]$, so it is recommended to create the index only once, for example, just after the problem object was created.

4. If the name index exists, it is automatically updated every time the name of a row/column is assigned/changed. The update operation takes logarithmic time.

5. If the name index does not exist, the application should not call the routines `glp_find_row` and `glp_find_col`. Otherwise, an error message will be issued and abnormal program termination will occur.

6. On destroying the problem object with the routine `glp_delete_prob`, the name index, if exists, is automatically destroyed.

2.5.1 glp_create_index — create the name index

Synopsis

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

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.

2.5.2 glp_find_row — find row by its name

Synopsis

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

Returns

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

2.5.3 glp_find_col — find column by its name

Synopsis

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

Returns

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

2.5.4 glp_delete_index — delete the name index

Synopsis

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

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.

2.6 Problem scaling routines

2.6.1 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.

2.6.2 glp_set_rii — set (change) row scale factor

Synopsis

```
void glp_set_rii(glp_prob *P, 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.

2.6.3 glp_set_sjj — set (change) column scale factor

Synopsis

```
void glp_set_sjj(glp_prob *P, 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.

[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.

2.6.4 glp_get_rii — retrieve row scale factor

Synopsis

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

Returns

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

2.6.5 glp_get_sjj — retrieve column scale factor

Synopsis

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

Returns

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

2.6.6 glp_scale_prob — scale problem data

Synopsis

```
void glp_scale_prob(glp_prob *P, 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 also specified as `GLP_SF_AUTO`, in which case the routine chooses the scaling options automatically.

2.6.7 glp_unscale_prob — unscale problem data

Synopsis

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

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.

2.7 LP basis constructing routines

2.7.1 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 $(I \mid -A)$ 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 **??**, page **??**.)

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 re-optimization there is no need to construct an initial basis from scratch.

2.7.2 glp_set_row_stat — set (change) row status

Synopsis

```
void glp_set_row_stat(glp_prob *P, 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);

`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 case of this routine);

`GLP_NF` — the same as `GLP_NL` (only in case of this routine);

`GLP_NS` — the same as `GLP_NL` (only in case of this routine).

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

2.7.3 glp_set_col_stat — set (change) column status

Synopsis

```
void glp_set_col_stat(glp_prob *P, 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 case of this routine);

`GLP_NF` — the same as `GLP_NL` (only in case of this routine);

`GLP_NS` — the same as `GLP_NL` (only in case of this routine).

2.7.4 glp_std_basis — construct standard initial LP basis

Synopsis

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

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).

2.7.5 glp_adv_basis — construct advanced initial LP basis

Synopsis

```
void glp_adv_basis(glp_prob *P, 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.

2.7.6 glp_cpx_basis — construct Bixby's initial LP basis

Synopsis

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

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.

2.8 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:

$$\begin{aligned} z = {} & d_1(x_N)_1 + d_2(x_N)_2 + \ldots + d_n(x_N)_n \\ (x_B)_1 = {} & \xi_{11}(x_N)_1 + \xi_{12}(x_N)_2 + \ldots + \xi_{1n}(x_N)_n \\ (x_B)_2 = {} & \xi_{21}(x_N)_1 + \xi_{22}(x_N)_2 + \ldots + \xi_{2n}(x_N)_n \\ & \ldots\ldots\ldots\ldots\ldots\ldots \\ (x_B)_m = {} & \xi_{m1}(x_N)_1 + \xi_{m2}(x_N)_2 + \ldots + \xi_{mn}(x_N)_n \end{aligned} \tag{2.3}$$

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 (2.3), 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.)

2.8.1 glp_simplex — solve LP problem with the primal or dual simplex method

Synopsis

```
int glp_simplex(glp_prob *P, 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.

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 (num) cnt
```

where: '`nnn`' is the iteration number, '`xxx`' is the current value of the objective function (it 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), '`num`' is the current number of primal or dual infeasibilities (phase I) or non-optimalities (phase II), '`cnt`' is the number of basis factorizations since the last terminal output.

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;

Hash (`#`) 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-9`)

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: `500`)

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.

Example 1

The following example main program reads LP problem instance in fixed MPS format from file `25fv47.mps`,[4] constructs an advanced initial basis, solves the instance with the primal simplex method (by default), and writes the solution to file `25fv47.txt`.

```
/* spxsamp1.c */

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

int main(void)
{     glp_prob *P;
      P = glp_create_prob();
      glp_read_mps(P, GLP_MPS_DECK, NULL, "25fv47.mps");
      glp_adv_basis(P, 0);
      glp_simplex(P, NULL);
      glp_print_sol(P, "25fv47.txt");
      glp_delete_prob(P);
      return 0;
}

/* eof */
```

Below here is shown the terminal output from this example program.

```
Reading problem data from '25fv47.mps'...
Problem: 25FV47
Objective: R0000
822 rows, 1571 columns, 11127 non-zeros
6919 records were read
One free row was removed
Constructing initial basis...
Size of triangular part is 812
GLPK Simplex Optimizer, v4.57
821 rows, 1571 columns, 10400 non-zeros
      0: obj =   7.131703290e+03 inf =   2.145e+05 (204)
    500: obj =   1.886711682e+04 inf =   8.273e+02 (36) 4
    741: obj =   1.846047936e+04 inf =   5.575e-14 (0) 2
*  1000: obj =   9.220063473e+03 inf =   2.423e-14 (432) 2
*  1500: obj =   6.187659664e+03 inf =   1.019e-13 (368) 4
*  2000: obj =   5.503442062e+03 inf =   0.000e+00 (33) 5
*  2052: obj =   5.501845888e+03 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
Writing basic solution to '25fv47.txt'...
```

[4] This instance in fixed MPS format can be found in the Netlib LP collection; see `ftp://ftp.netlib.org/lp/data/`.

Example 2

The following example main program solves the same LP problem instance as in Example 1 above, however, it uses the dual simplex method, which starts from the standard initial basis.

```
/* spxsamp2.c */

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

int main(void)
{     glp_prob *P;
      glp_smcp parm;
      P = glp_create_prob();
      glp_read_mps(P, GLP_MPS_DECK, NULL, "25fv47.mps");
      glp_init_smcp(&parm);
      parm.meth = GLP_DUAL;
      glp_simplex(P, &parm);
      glp_print_sol(P, "25fv47.txt");
      glp_delete_prob(P);
      return 0;
}

/* eof */
```

Below here is shown the terminal output from this example program.

```
Reading problem data from '25fv47.mps'...
Problem: 25FV47
Objective: R0000
822 rows, 1571 columns, 11127 non-zeros
6919 records were read
One free row was removed
GLPK Simplex Optimizer, v4.57
821 rows, 1571 columns, 10400 non-zeros
      0:                          inf =   1.223e+02 (41)
    258:                          inf =   3.091e-16 (0) 2
#   500: obj =  -5.071287080e+03 inf =   2.947e-15 (292) 2
#  1000: obj =  -1.352843873e+03 inf =   8.452e-15 (302) 5
#  1500: obj =   7.985859737e+02 inf =   1.127e-14 (263) 5
#  2000: obj =   3.059023029e+03 inf =   6.290e-11 (197) 4
#  2500: obj =   5.354770966e+03 inf =   7.172e-13 (130) 5
#  2673: obj =   5.501845888e+03 inf =   3.802e-16 (0) 2
OPTIMAL LP SOLUTION FOUND
Writing basic solution to '25fv47.txt'...
```

2.8.2 glp_exact — solve LP problem in exact arithmetic

Synopsis

```
int glp_exact(glp_prob *P, const glp_smcp *parm);
```

Description

The routine `glp_exact` is a tentative 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.

Note that the routine `glp_exact` uses only three control parameters passed in the structure `glp_smcp`, namely, `msg_lev`, `it_lim`, and `tm_lim`.

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 exactly singular.
`GLP_EBOUND`	Unable to start the search, because some double-bounded (auxiliary or structural) variables have incorrect bounds.
`GLP_EFAIL`	The problem instance has no rows/columns.
`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.

Note

Computations in exact arithmetic are very time-consuming, so solving LP problem with the routine `glp_exact` from the very beginning is not a good idea. It is much better at first to find an optimal basis with the routine `glp_simplex` and only then to call `glp_exact`, in which case only a few simplex iterations need to be performed in exact arithmetic.

2.8.3 glp_init_smcp — initialize simplex solver 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.

2.8.4 glp_get_status — determine generic status of basic solution

Synopsis

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

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`.

2.8.5 glp_get_prim_stat — retrieve status of primal basic solution

Synopsis

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

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.

2.8.6 glp_get_dual_stat — retrieve status of dual basic solution

Synopsis

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

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.

2.8.7 glp_get_obj_val — retrieve objective value

Synopsis

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

Returns

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

2.8.8 glp_get_row_stat — retrieve row status

Synopsis

```
int glp_get_row_stat(glp_prob *P, 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.

2.8.9 glp_get_row_prim — retrieve row primal value

Synopsis

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

Returns

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

2.8.10 glp_get_row_dual — retrieve row dual value

Synopsis

```
double glp_get_row_dual(glp_prob *P, 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.

2.8.11 glp_get_col_stat — retrieve column status

Synopsis

```
int glp_get_col_stat(glp_prob *P, 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.

2.8.12 glp_get_col_prim — retrieve column primal value

Synopsis

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

Returns

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

2.8.13 glp_get_col_dual — retrieve column dual value

Synopsis

```
double glp_get_col_dual(glp_prob *P, 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.

2.8.14 glp_get_unbnd_ray — determine variable causing unboundedness

Synopsis

```
int glp_get_unbnd_ray(glp_prob *P);
```

Returns

The routine `glp_get_unbnd_ray` returns the number k of a variable, which causes primal or dual unboundedness. If $1 \leq k \leq m$, it is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, it is $(k - m)$-th structural variable, where m is the number of rows, n is the number of columns in the problem object. If such variable is not defined, the routine returns 0.

Note

If it is not exactly known which version of the simplex solver detected unboundedness, i.e. whether the unboundedness is primal or dual, it is sufficient to check the status of the variable with the routine `glp_get_row_stat` or `glp_get_col_stat`. If the variable is non-basic, the unboundedness is primal, otherwise, if the variable is basic, the unboundedness is dual (the latter case means that the problem has no primal feasible dolution).

2.9 Interior-point method routines

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

In brief, the GLPK interior-point solver works as follows.

At first, the solver transforms the original LP to a *working* LP in the standard format:

minimize

$$z = c_1 x_{m+1} + c_2 x_{m+2} + \ldots + c_n x_{m+n} + c_0 \tag{2.4}$$

subject to linear constraints

$$\begin{aligned} a_{11} x_{m+1} + a_{12} x_{m+2} + \ldots + a_{1n} x_{m+n} &= b_1 \\ a_{21} x_{m+1} + a_{22} x_{m+2} + \ldots + a_{2n} x_{m+n} &= b_2 \\ \ldots \ldots \ldots \ldots \ldots \ldots \ldots & \\ a_{m1} x_{m+1} + a_{m2} x_{m+2} + \ldots + a_{mn} x_{m+n} &= b_m \end{aligned} \tag{2.5}$$

and non-negative variables

$$x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots, \quad x_n \geq 0 \tag{2.6}$$

where: z is the objective function; x_1, $\ldots$, x_n are variables; c_1, $\ldots$, c_n are objective coefficients; c_0 is a constant term of the objective function; a_{11}, $\ldots$, a_{mn} are constraint coefficients; b_1, $\ldots$, b_m are right-hand sides.

Using vector and matrix notations the working LP (2.4)—(2.6) can be written as follows:

$$z = c^T x + c_0 \ \rightarrow \ \min, \tag{2.7}$$

$$Ax = b, \tag{2.8}$$

$$x \geq 0, \tag{2.9}$$

where: $x = (x_j)$ is n-vector of variables, $c = (c_j)$ is n-vector of objective coefficients, $A = (a_{ij})$ is $m \times n$-matrix of constraint coefficients, and $b = (b_i)$ is m-vector of right-hand sides.

Karush–Kuhn–Tucker optimality conditions for LP (2.7)—(2.9) are the following:

$$Ax = b, \tag{2.10}$$

$$A^T \pi + \lambda = c, \tag{2.11}$$

$$\lambda^T x = 0, \tag{2.12}$$

$$x \geq 0, \quad \lambda \geq 0, \tag{2.13}$$

where: π is m-vector of Lagrange multipliers (dual variables) for equality constraints (2.8), λ is n-vector of Lagrange multipliers (dual variables) for non-negativity constraints (2.9), (2.10) is the primal feasibility condition, (2.11) is the dual feasibility condition, (2.12) is the primal-dual complementarity condition, and (2.13) is the non-negativity conditions.

The main idea of the primal-dual interior-point method is based on finding a point in the primal-dual space (i.e. in the space of all primal and dual variables x, π, and λ), which satisfies

to all optimality conditions (2.10)—(2.13). Obviously, x-component of such point then provides an optimal solution to the working LP (2.7)—(2.9).

To find the optimal point (x^*, π^*, λ^*) the interior-point method attempts to solve the system of equations (2.10)—(2.12), which is closed in the sense that the number of variables x_j, π_i, and λ_j and the number equations are the same and equal to $m + 2n$. Due to condition (2.12) this system of equations is non-linear, so it can be solved with a version of *Newton's method* provided with additional rules to keep the current point within the positive orthant as required by the non-negativity conditions (2.13).

Finally, once the optimal point (x^*, π^*, λ^*) has been found, the solver performs inverse transformations to recover corresponding solution to the original LP passed to the solver from the application program.

2.9.1 glp_interior — solve LP problem with the interior-point method

Synopsis

```
int glp_interior(glp_prob *P, const glp_iptcp *parm);
```

Description

The routine `glp_interior` is a driver to the LP solver based on the primal-dual interior-point 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 interior-point solver has a set of control parameters. Values of the control parameters can be passed in the structure `glp_iptcp`, 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_iptcp` by default values of all control parameters using the routine `glp_init_iptcp` (see the next subsection). This is needed for backward compatibility, because in the future there may appear new members in the structure `glp_iptcp`.

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_EFAIL`	The problem has no rows/columns.
`GLP_ENOCVG`	Very slow convergence or divergence.
`GLP_EITLIM`	Iteration limit exceeded.
`GLP_EINSTAB`	Numerical instability on solving Newtonian system.

Comments

The routine `glp_interior` implements an easy version of the primal-dual interior-point method based on Mehrotra's technique.[5]

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 may 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 interior-point solution found.

Terminal output

Solving large LP problems may take a long time, so the solver reports some information about every interior-point iteration,[6] which is sent to the terminal. This information has the following format:

```
nnn: obj = 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 (cf. (2.10)):

$$\frac{\|Ax^{(k)} - b\|}{1 + \|b\|}, \tag{2.14}$$

`ddd` is the current relative dual infeasibility (cf. (2.11)):

$$\frac{\|A^T\pi^{(k)} + \lambda^{(k)} - c\|}{1 + \|c\|}, \tag{2.15}$$

`ggg` is the current primal-dual gap (cf. (2.12)):

$$\frac{|c^Tx^{(k)} - b^T\pi^{(k)}|}{1 + |c^Tx^{(k)}|}, \tag{2.16}$$

and $[x^{(k)}, \pi^{(k)}, \lambda^{(k)}]$ is the current point on k-th iteration, $k = 0, 1, 2, \ldots$. Note that all solution components are internally scaled, so information sent to the terminal is suitable only for visual inspection.

[5] S. Mehrotra. On the implementation of a primal-dual interior point method. SIAM J. on Optim., 2(4), pp. 575-601, 1992.

[6] 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.

Control parameters

This paragraph describes all control parameters currently used in the interior-point solver. Symbolic names of control parameters are names of corresponding members in the structure `glp_iptcp`.

`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 ord_alg` (default: `GLP_ORD_AMD`)

Ordering algorithm used prior to Cholesky factorization:

`GLP_ORD_NONE` —use natural (original) ordering;

`GLP_ORD_QMD` —quotient minimum degree (QMD);

`GLP_ORD_AMD` —approximate minimum degree (AMD);

`GLP_ORD_SYMAMD`—approximate minimum degree (SYMAMD).

Example

The following main program reads LP problem instance in fixed MPS format from file `25fv47.mps`,[7] solves it with the interior-point solver, and writes the solution to file `25fv47.txt`.

```
/* iptsamp.c */

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

int main(void)
{     glp_prob *P;
      P = glp_create_prob();
      glp_read_mps(P, GLP_MPS_DECK, NULL, "25fv47.mps");
      glp_interior(P, NULL);
      glp_print_ipt(P, "25fv47.txt");
      glp_delete_prob(P);
      return 0;
}

/* eof */
```

[7] This instance in fixed MPS format can be found in the Netlib LP collection; see `ftp://ftp.netlib.org/lp/data/`.

Below here is shown the terminal output from this example program.

```
Reading problem data from '25fv47.mps'...
Problem: 25FV47
Objective: R0000
822 rows, 1571 columns, 11127 non-zeros
6919 records were read
Original LP has 822 row(s), 1571 column(s), and 11127 non-zero(s)
Working LP has 821 row(s), 1876 column(s), and 10705 non-zero(s)
Matrix A has 10705 non-zeros
Matrix S = A*A' has 11895 non-zeros (upper triangle)
Minimal degree ordering...
Computing Cholesky factorization S = L'*L...
Matrix L has 35411 non-zeros
Guessing initial point...
Optimization begins...
  0: obj =   1.823377629e+05; rpi =  1.3e+01; rdi =  1.4e+01; gap =  9.3e-01
  1: obj =   9.260045192e+04; rpi =  5.3e+00; rdi =  5.6e+00; gap =  6.8e+00
  2: obj =   3.596999742e+04; rpi =  1.5e+00; rdi =  1.2e+00; gap =  1.8e+01
  3: obj =   1.989627568e+04; rpi =  4.7e-01; rdi =  3.0e-01; gap =  1.9e+01
  4: obj =   1.430215557e+04; rpi =  1.1e-01; rdi =  8.6e-02; gap =  1.4e+01
  5: obj =   1.155716505e+04; rpi =  2.3e-02; rdi =  2.4e-02; gap =  6.8e+00
  6: obj =   9.660273208e+03; rpi =  6.7e-03; rdi =  4.6e-03; gap =  3.9e+00
  7: obj =   8.694348283e+03; rpi =  3.7e-03; rdi =  1.7e-03; gap =  2.0e+00
  8: obj =   8.019543639e+03; rpi =  2.4e-03; rdi =  3.9e-04; gap =  1.0e+00
  9: obj =   7.122676293e+03; rpi =  1.2e-03; rdi =  1.5e-04; gap =  6.6e-01
 10: obj =   6.514534518e+03; rpi =  6.1e-04; rdi =  4.3e-05; gap =  4.1e-01
 11: obj =   6.361572203e+03; rpi =  4.8e-04; rdi =  2.2e-05; gap =  3.0e-01
 12: obj =   6.203355508e+03; rpi =  3.2e-04; rdi =  1.7e-05; gap =  2.6e-01
 13: obj =   6.032943411e+03; rpi =  2.0e-04; rdi =  9.3e-06; gap =  2.1e-01
 14: obj =   5.796553021e+03; rpi =  9.8e-05; rdi =  3.2e-06; gap =  1.0e-01
 15: obj =   5.667032431e+03; rpi =  4.4e-05; rdi =  1.1e-06; gap =  5.6e-02
 16: obj =   5.613911867e+03; rpi =  2.5e-05; rdi =  4.1e-07; gap =  3.5e-02
 17: obj =   5.560572626e+03; rpi =  9.9e-06; rdi =  2.3e-07; gap =  2.1e-02
 18: obj =   5.537276001e+03; rpi =  5.5e-06; rdi =  8.4e-08; gap =  1.1e-02
 19: obj =   5.522746942e+03; rpi =  2.2e-06; rdi =  4.0e-08; gap =  6.7e-03
 20: obj =   5.509956679e+03; rpi =  7.5e-07; rdi =  1.8e-08; gap =  2.9e-03
 21: obj =   5.504571733e+03; rpi =  1.6e-07; rdi =  5.8e-09; gap =  1.1e-03
 22: obj =   5.502576367e+03; rpi =  3.4e-08; rdi =  1.0e-09; gap =  2.5e-04
 23: obj =   5.502057119e+03; rpi =  8.1e-09; rdi =  3.0e-10; gap =  7.7e-05
 24: obj =   5.501885996e+03; rpi =  9.4e-10; rdi =  1.2e-10; gap =  2.4e-05
 25: obj =   5.501852464e+03; rpi =  1.4e-10; rdi =  1.2e-11; gap =  3.0e-06
 26: obj =   5.501846549e+03; rpi =  1.4e-11; rdi =  1.2e-12; gap =  3.0e-07
 27: obj =   5.501845954e+03; rpi =  1.4e-12; rdi =  1.2e-13; gap =  3.0e-08
 28: obj =   5.501845895e+03; rpi =  1.5e-13; rdi =  1.2e-14; gap =  3.0e-09
OPTIMAL SOLUTION FOUND
Writing interior-point solution to '25fv47.txt'...
```

2.9.2 glp_init_iptcp — initialize interior-point solver control parameters

Synopsis

```
int glp_init_iptcp(glp_iptcp *parm);
```

Description

The routine `glp_init_iptcp` initializes control parameters, which are used by the interior-point solver, with default values.

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

2.9.3 glp_ipt_status — determine solution status

Synopsis

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

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;

`GLP_INFEAS` — interior-point solution is infeasible;

`GLP_NOFEAS` — no feasible primal-dual solution exists.

2.9.4 glp_ipt_obj_val — retrieve objective value

Synopsis

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

Returns

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

2.9.5 glp_ipt_row_prim — retrieve row primal value

Synopsis

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

Returns

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

2.9.6 glp_ipt_row_dual — retrieve row dual value

Synopsis

```
double glp_ipt_row_dual(glp_prob *P, 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.

2.9.7 glp_ipt_col_prim — retrieve column primal value

Synopsis

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

Returns

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

2.9.8 glp_ipt_col_dual — retrieve column dual value

Synopsis

```
double glp_ipt_col_dual(glp_prob *P, 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.

2.10 Mixed integer programming routines

2.10.1 glp_set_col_kind — set (change) column kind

Synopsis

```
void glp_set_col_kind(glp_prob *P, 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.

2.10.2 glp_get_col_kind — retrieve column kind

Synopsis

```
int glp_get_col_kind(glp_prob *P, 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.

2.10.3 glp_get_num_int — retrieve number of integer columns

Synopsis

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

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.

2.10.4 glp_get_num_bin — retrieve number of binary columns

Synopsis

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

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.

2.10.5 glp_intopt — solve MIP problem with the branch-and-cut method

Synopsis

```
int glp_intopt(glp_prob *P, 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".

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;

`GLP_BR_PCH` — hybrid pseudo-cost heuristic.

`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 sr_heur` (default: `GLP_ON`)

Simple rounding heuristic option:

`GLP_ON` — enable applying the simple rounding heuristic;

`GLP_OFF` — disable applying the simple rounding heuristic.

`int fp_heur` (default: `GLP_OFF`)

Feasibility pump heuristic option:

`GLP_ON` — enable applying the feasibility pump heuristic;

`GLP_OFF` — disable applying the feasibility pump heuristic.

`int ps_heur` (default: `GLP_OFF`)

Proximity search heuristic[8] option:

`GLP_ON` — enable applying the proximity search heuristic;

`GLP_OFF` — disable applying the proximity search pump heuristic.

`int ps_tm_lim` (default: `60000`)

Time limit, in milliseconds, for the proximity search heuristic (see above).

`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.

[8] The Fischetti–Monaci Proximity Search (a.k.a. Proxy) heuristic. This algorithm is often capable of rapidly improving a feasible solution of a MIP problem with binary variables. It allows to quickly obtain suboptimal solutions in some problems which take too long time to be solved to optimality.

`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.

2.10.6 glp_init_iocp — 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.

2.10.7 glp_mip_status — determine status of MIP solution

Synopsis

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

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).

2.10.8 glp_mip_obj_val — retrieve objective value

Synopsis

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

Returns

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

2.10.9 glp_mip_row_val — retrieve row value

Synopsis

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

Returns

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

2.10.10 glp_mip_col_val — retrieve column value

Synopsis

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

Returns

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

2.11 Additional routines

2.11.1 glp_check_kkt — check feasibility/optimality conditions

Synopsis

```
void glp_check_kkt(glp_prob *P, int sol, int cond, double *ae_max, int *ae_ind,
                   double *re_max, int *re_ind);
```

Description

The routine `glp_check_kkt` allows to check feasibility/optimality conditions for the current solution stored in the specified problem object. (For basic and interior-point solutions these conditions are known as *Karush–Kuhn–Tucker optimality conditions*.)

The parameter `sol` specifies which solution should be checked:

`GLP_SOL` — basic solution;

`GLP_IPT` — interior-point solution;

`GLP_MIP` — mixed integer solution.

The parameter `cond` specifies which condition should be checked:

`GLP_KKT_PE` — check primal equality constraints (KKT.PE);

`GLP_KKT_PB` — check primal bound constraints (KKT.PB);

`GLP_KKT_DE` — check dual equality constraints (KKT.DE). This conditions can be checked only for basic or interior-point solution;

`GLP_KKT_DB` — check dual bound constraints (KKT.DB). This conditions can be checked only for basic or interior-point solution.

Detailed explanations of these conditions are given below in paragraph "Background".

On exit the routine stores the following information to locations specified by parameters `ae_max`, `ae_ind`, `re_max`, and `re_ind` (if some parameter is a null pointer, corresponding information is not stored):

`ae_max` — largest absolute error;

`ae_ind` — number of row (KKT.PE), column (KKT.DE), or variable (KKT.PB, KKT.DB) with the largest absolute error;

`re_max` — largest relative error;

`re_ind` — number of row (KKT.PE), column (KKT.DE), or variable (KKT.PB, KKT.DB) with the largest relative error.

Row (auxiliary variable) numbers are in the range 1 to m, where m is the number of rows in the problem object. Column (structural variable) numbers are in the range 1 to n, where n is the number of columns in the problem object. Variable numbers are in the range 1 to $m + n$, where variables with numbers 1 to m correspond to rows, and variables with numbers $m + 1$ to $m + n$ correspond to columns. If the error reported is exact zero, corresponding row, column or variable number is set to zero.

Background

The first condition checked by the routine is the following:

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

where x_R is the subvector of auxiliary variables (rows), x_S is the subvector of structural variables (columns), A is the constraint matrix. This condition expresses the requirement that all primal variables should 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 solution is, that depends on accuracy of a representation of the basis matrix used by the simplex method, or on accuracy provided by the interior-point method.

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

$$g = x_R - Ax_S,$$

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

$$\texttt{ae_max} = \max_{1 \le i \le m} |g_i|,$$

$$\texttt{re_max} = \max_{1 \le i \le m} \frac{|g_i|}{1 + |(x_R)_i|}.$$

The second condition checked by the routine is the following:

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

where x_k is auxiliary ($1 \le k \le m$) or structural ($m + 1 \le k \le m + n$) 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 shoudl satisfy to bound constraints of the original LP problem. In case of basic solution all non-basic variables are placed on their active bounds, so actually the condition (KKT.PB) needs to be checked for basic variables only. If the primal solution has sufficient accuracy, this condition shows its primal feasibility.

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

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

$$\texttt{re_max} = \max_{1 \le k \le m+n} \frac{|h_k|}{1 + |x_k|}.$$

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 should belong to the orthogonal complement of a linear subspace defined by the equality and active bound constraints, i.e. that the gradient is a linear combination of normals to the constraint hyperplanes, where Lagrange multipliers π and d are coefficients of that linear combination.

To eliminate the vector π rewrite the third condition 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,

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

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

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

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

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

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

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

$$\texttt{ae_max} = \max_{1 \le j \le n} |u_j|,$$

$$\texttt{re_max} = \max_{1 \le j \le n} \frac{|u_j|}{1 + |(d_S)_j - (c_S)_j|}.$$

The fourth condition checked by the routine is the following:

$$\begin{cases} \text{if } -\infty < x_k < +\infty, \text{ then } d_k = 0 \\ \text{if } \quad l_k \le x_k < +\infty, \text{ then } d_k \ge 0 \text{ (minimization)} \\ \qquad\qquad\qquad\qquad\quad d_k \le 0 \text{ (maximization)} \\ \text{if } -\infty < x_k \le u_k, \quad \text{then } d_k \le 0 \text{ (minimization)} \\ \qquad\qquad\qquad\qquad\quad d_k \ge 0 \text{ (maximization)} \\ \text{if } \quad l_k \le x_k \le u_k, \quad \text{then } d_k \text{ is of any sign} \end{cases} \tag{KKT.DB}$$

for all $k = 1, \ldots, m+n$, where d_k is a reduced cost (Lagrange multiplier) of auxiliary ($1 \le k \le m$) or structural ($m+1 \le k \le m+n$) 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. In case of basic solution reduced costs of all basic variables are placed on their active (zero) bounds, so actually the condition (KKT.DB) needs to be checked for non-basic variables only. If the dual solution has sufficient accuracy, this condition shows the dual feasibility of the solution.

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

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

$$\texttt{re_max} = \max_{1 \le k \le m+n} \frac{|v_k|}{1 + |d_k - c_k|}.$$

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 should be on its bound for all $k = 1, \ldots, m + n$. Currently checking this condition is not implemented yet.

Chapter 3

Utility API routines

3.1 Problem data reading/writing routines

3.1.1 glp_read_mps — read problem data in MPS format

Synopsis

```
int glp_read_mps(glp_prob *P, int fmt, const glp_mpscp *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 **??**, page **??**.)

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 should 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 glp_write_mps — write problem data in MPS format

Synopsis

```
int glp_write_mps(glp_prob *P, int fmt, const glp_mpscp *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 **??**, page **??**.)

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 should 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 glp_read_lp — read problem data in CPLEX LP format

Synopsis

```
int glp_read_lp(glp_prob *P, const glp_cpxcp *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 **??**, page **??**.)

The parameter `parm` is reserved for use in the future and should 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 glp_write_lp — write problem data in CPLEX LP format

Synopsis

```
int glp_write_lp(glp_prob *P, const glp_cpxcp *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 **??**, page **??**.)

The parameter `parm` is reserved for use in the future and should 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 glp_read_prob — read problem data in GLPK format

Synopsis

```
int glp_read_prob(glp_prob *P, int flags, const char *fname);
```

Description

The routine `glp_read_prob` reads problem data in the GLPK LP/MIP format from a text file. (For description of the GLPK LP/MIP format see below.)

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

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_prob` 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_prob` returns zero. Otherwise, it prints an error message and returns non-zero.

GLPK LP/MIP format

The GLPK LP/MIP format is a DIMACS-like format.[1] The file in this format is a plain ASCII text file containing lines of several types described below. A line is terminated with the end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

[1] The DIMACS formats were developed by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) to facilitate exchange of problem data. For details see: `<http://dimacs.rutgers.edu/Challenges/>`.

The first line of the data file must be the problem line (except optional comment lines, which may precede the problem line). The last line of the data file must be the end line. Other lines may follow in arbitrary order, however, duplicate lines are not allowed.

Comment lines. Comment lines give human-readable information about the data file and are ignored by GLPK routines. Comment lines can appear anywhere in the data file. Each comment line begins with the lower-case character `c`.

```
c This is an example of comment line
```

Problem line. There must be exactly one problem line in the data file. This line must appear before any other lines except comment lines and has the following format:

```
p CLASS DIR ROWS COLS NONZ
```

The lower-case letter `p` specifies that this is the problem line.

The `CLASS` field defines the problem class and can contain either the keyword `lp` (that means linear programming problem) or `mip` (that means mixed integer programming problem).

The `DIR` field defines the optimization direction (that is, the objective function sense) and can contain either the keyword `min` (that means minimization) or `max` (that means maximization).

The `ROWS`, `COLS`, and `NONZ` fields contain non-negative integer values specifying, respectively, the number of rows (constraints), columns (variables), and non-zero constraint coefficients in the problem instance. Note that `NONZ` value does not account objective coefficients.

Row descriptors. There must be at most one row descriptor line in the data file for each row (constraint). This line has one of the following formats:

```
i ROW f
i ROW l RHS
i ROW u RHS
i ROW d RHS1 RHS2
i ROW s RHS
```

The lower-case letter `i` specifies that this is the row descriptor line.

The `ROW` field specifies the row ordinal number, an integer between 1 and m, where m is the number of rows in the problem instance.

The next lower-case letter specifies the row type as follows:

`f` — free (unbounded) row: $-\infty < \sum a_j x_j < +\infty$;

`l` — inequality constraint of '$\geq$' type: $\sum a_j x_j \geq b$;

`u` — inequality constraint of '$\leq$' type: $\sum a_j x_j \leq b$;

`d` — double-sided inequality constraint: $b_1 \leq \sum a_j x_j \leq b_2$;

`s` — equality constraint: $\sum a_j x_j = b$.

The `RHS` field contains a floaing-point value specifying the row right-hand side. The `RHS1` and `RHS2` fields contain floating-point values specifying, respectively, the lower and upper right-hand sides for the double-sided row.

If for some row its descriptor line does not appear in the data file, by default that row is assumed to be an equality constraint with zero right-hand side.

Column descriptors. There must be at most one column descriptor line in the data file for

each column (variable). This line has one of the following formats depending on the problem class specified in the problem line:

LP class	MIP class
`j COL f`	`j COL KIND f`
`j COL l BND`	`j COL KIND l BND`
`j COL u BND`	`j COL KIND u BND`
`j COL d BND1 BND2`	`j COL KIND d BND1 BND2`
`j COL s BND`	`j COL KIND s BND`

The lower-case letter `j` specifies that this is the column descriptor line.

The `COL` field specifies the column ordinal number, an integer between 1 and n, where n is the number of columns in the problem instance.

The `KIND` field is used only for MIP problems and specifies the column kind as follows:

`c` — continuous column;

`i` — integer column;

`b` — binary column (in this case all remaining fields must be omitted).

The next lower-case letter specifies the column type as follows:

`f` — free (unbounded) column: $-\infty < x < +\infty$;

`l` — column with lower bound: $x \geq l$;

`u` — column with upper bound: $x \leq u$;

`d` — double-bounded column: $l \leq x \leq u$;

`s` — fixed column: $x = s$.

The `BND` field contains a floating-point value that specifies the column bound. The `BND1` and `BND2` fields contain floating-point values specifying, respectively, the lower and upper bounds for the double-bounded column.

If for some column its descriptor line does not appear in the file, by default that column is assumed to be non-negative (in case of LP class) or binary (in case of MIP class).

Coefficient descriptors. There must be exactly one coefficient descriptor line in the data file for each non-zero objective or constraint coefficient. This line has the following format:

```
a ROW COL VAL
```

The lower-case letter `a` specifies that this is the coefficient descriptor line.

For objective coefficients the `ROW` field must contain 0. For constraint coefficients the `ROW` field specifies the row ordinal number, an integer between 1 and m, where m is the number of rows in the problem instance.

The `COL` field specifies the column ordinal number, an integer between 1 and n, where n is the number of columns in the problem instance.

If both the `ROW` and `COL` fields contain 0, the line specifies the constant term ("shift") of the objective function rather than objective coefficient.

The `VAL` field contains a floating-point coefficient value (it is allowed to specify zero value in this field).

The number of constraint coefficient descriptor lines must be exactly the same as specified in the field `NONZ` of the problem line.

Symbolic name descriptors. There must be at most one symbolic name descriptor line for the problem instance, objective function, each row (constraint), and each column (variable). This line has one of the following formats:

```
n p NAME
n z NAME
n i ROW NAME
n j COL NAME
```

The lower-case letter `n` specifies that this is the symbolic name descriptor line.

The next lower-case letter specifies which object should be assigned a symbolic name:

`p` — problem instance;

`z` — objective function;

`i` — row (constraint);

`j` — column (variable).

The `ROW` field specifies the row ordinal number, an integer between 1 and m, where m is the number of rows in the problem instance.

The `COL` field specifies the column ordinal number, an integer between 1 and n, where n is the number of columns in the problem instance.

The `NAME` field contains the symbolic name, a sequence from 1 to 255 arbitrary graphic ASCII characters, assigned to corresponding object.

End line. There must be exactly one end line in the data file. This line must appear last in the file and has the following format:

```
e
```

The lower-case letter `e` specifies that this is the end line. Anything that follows the end line is ignored by GLPK routines.

Example of data file in GLPK LP/MIP format

The following example of a data file in GLPK LP/MIP format specifies the same LP problem as in Subsection "Example of MPS file".

```
p lp min 8 7 48
n p PLAN
n z VALUE
i 1 f
n i 1 VALUE
i 2 s 2000
n i 2 YIELD
i 3 u 60
n i 3 FE
i 4 u 100
n i 4 CU
i 5 u 40
n i 5 MN
i 6 u 30
n i 6 MG
i 7 l 1500
n i 7 AL
i 8 d 250 300
n i 8 SI
j 1 d 0 200
n j 1 BIN1
j 2 d 0 2500
n j 2 BIN2
j 3 d 400 800
n j 3 BIN3
j 4 d 100 700
n j 4 BIN4
j 5 d 0 1500
n j 5 BIN5
n j 6 ALUM
n j 7 SILICON
a 0 1 0.03
a 0 2 0.08
a 0 3 0.17
a 0 4 0.12
a 0 5 0.15
a 0 6 0.21
a 0 7 0.38
a 1 1 0.03
a 1 2 0.08
a 1 3 0.17
a 1 4 0.12
a 1 5 0.15
a 1 6 0.21
a 1 7 0.38
a 2 1 1
a 2 2 1
a 2 3 1
a 2 4 1
a 2 5 1
a 2 6 1
a 2 7 1
a 3 1 0.15
a 3 2 0.04
a 3 3 0.02
a 3 4 0.04
a 3 5 0.02
a 3 6 0.01
a 3 7 0.03
a 4 1 0.03
a 4 2 0.05
a 4 3 0.08
a 4 4 0.02
a 4 5 0.06
a 4 6 0.01
a 5 1 0.02
a 5 2 0.04
a 5 3 0.01
a 5 4 0.02
a 5 5 0.02
a 6 1 0.02
a 6 2 0.03
a 6 5 0.01
a 7 1 0.7
a 7 2 0.75
a 7 3 0.8
a 7 4 0.75
a 7 5 0.8
a 7 6 0.97
a 8 1 0.02
a 8 2 0.06
a 8 3 0.08
a 8 4 0.12
a 8 5 0.02
a 8 6 0.01
a 8 7 0.97
e o f
```

3.1.6 glp_write_prob — write problem data in GLPK format

Synopsis

```
int glp_write_prob(glp_prob *P, int flags, const char *fname);
```

Description

The routine `glp_write_prob` writes problem data in the GLPK LP/MIP format to a text file. (For description of the GLPK LP/MIP format see Subsection "Read problem data in GLPK format".)

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

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_prob` performs automatic compression on writing it.)

Returns

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

3.2 Routines for processing MathProg models

3.2.1 Introduction

GLPK supports the *GNU MathProg modeling language*.[2] As a rule, models written in MathProg are solved with the GLPK LP/MIP stand-alone solver `glpsol` (see Appendix D) and do not need any programming with API routines. However, for various reasons the user may need to process MathProg models directly in his/her application program, in which case he/she may use API routines described in this section. These routines provide an interface to the *MathProg translator*, a component of GLPK, which translates MathProg models into an internal code and then interprets (executes) this code.

The processing of a model written in GNU MathProg includes several steps, which should be performed in the following order:

1. *Allocating the workspace.* The translator allocates the workspace, an internal data structure used on all subsequent steps.

2. *Reading model section.* The translator reads model section and, optionally, data section from a specified text file and translates them into the internal code. If necessary, on this step data section may be ignored.

3. *Reading data section(s).* The translator reads one or more data sections from specified text file(s) and translates them into the internal code.

4. *Generating the model.* The translator executes the internal code to evaluate the content of the model objects such as sets, parameters, variables, constraints, and objectives. On this step the execution is suspended at the solve statement.

5. *Building the problem object.* The translator obtains all necessary information from the workspace and builds the standard problem object (that is, the program object of type `glp_prob`).

6. *Solving the problem.* On this step the problem object built on the previous step is passed to a solver, which solves the problem instance and stores its solution back to the problem object.

7. *Postsolving the model.* The translator copies the solution from the problem object to the workspace and then executes the internal code from the solve statement to the end of the model. (If model has no solve statement, the translator does nothing on this step.)

8. *Freeing the workspace.* The translator frees all the memory allocated to the workspace.

Note that the MathProg translator performs no error correction, so if any of steps 2 to 7 fails (due to errors in the model), the application program should terminate processing and go to step 8.

[2] The GNU MathProg modeling language is a subset of the AMPL language. For its detailed description see the document "Modeling Language GNU MathProg: Language Reference" included in the GLPK distribution.

Example 1

In this example the program reads model and data sections from input file `egypt.mod`[3] and writes the model to output file `egypt.mps` in free MPS format (see Appendix B). No solution is performed.

```
/* mplsamp1.c */

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

int main(void)
{     glp_prob *lp;
      glp_tran *tran;
      int ret;
      lp = glp_create_prob();
      tran = glp_mpl_alloc_wksp();
      ret = glp_mpl_read_model(tran, "egypt.mod", 0);
      if (ret != 0)
      {  fprintf(stderr, "Error on translating model\n");
         goto skip;
      }
      ret = glp_mpl_generate(tran, NULL);
      if (ret != 0)
      {  fprintf(stderr, "Error on generating model\n");
         goto skip;
      }
      glp_mpl_build_prob(tran, lp);
      ret = glp_write_mps(lp, GLP_MPS_FILE, NULL, "egypt.mps");
      if (ret != 0)
         fprintf(stderr, "Error on writing MPS file\n");
skip: glp_mpl_free_wksp(tran);
      glp_delete_prob(lp);
      return 0;
}

/* eof */
```

[3] This is an example model included in the GLPK distribution.

Example 2

In this example the program reads model section from file `sudoku.mod`[4] ignoring data section in this file, reads alternative data section from file `sudoku.dat`, solves the problem instance and passes the solution found back to the model.

```
/* mplsamp2.c */

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

int main(void)
{     glp_prob *mip;
      glp_tran *tran;
      int ret;
      mip = glp_create_prob();
      tran = glp_mpl_alloc_wksp();
      ret = glp_mpl_read_model(tran, "sudoku.mod", 1);
      if (ret != 0)
      {  fprintf(stderr, "Error on translating model\n");
         goto skip;
      }
      ret = glp_mpl_read_data(tran, "sudoku.dat");
      if (ret != 0)
      {  fprintf(stderr, "Error on translating data\n");
         goto skip;
      }
      ret = glp_mpl_generate(tran, NULL);
      if (ret != 0)
      {  fprintf(stderr, "Error on generating model\n");
         goto skip;
      }
      glp_mpl_build_prob(tran, mip);
      glp_simplex(mip, NULL);
      glp_intopt(mip, NULL);
      ret = glp_mpl_postsolve(tran, mip, GLP_MIP);
      if (ret != 0)
         fprintf(stderr, "Error on postsolving model\n");
skip: glp_mpl_free_wksp(tran);
      glp_delete_prob(mip);
      return 0;
}

/* eof */
```

[4] This is an example model which is included in the GLPK distribution along with alternative data file `sudoku.dat`.

3.2.2 glp_mpl_alloc_wksp — allocate the translator workspace

Synopsis

```
glp_tran *glp_mpl_alloc_wksp(void);
```

Description

The routine `glp_mpl_alloc_wksp` allocates the MathProg translator workspace. (Note that multiple instances of the workspace may be allocated, if necessary.)

Returns

The routine returns a pointer to the workspace, which should be used in all subsequent operations.

3.2.3 glp_mpl_init_rand — initialize pseudo-random number generator

Synopsis

```
void glp_mpl_init_rand(glp_tran *tran, int seed);
```

Description

The routine `glp_mpl_init_rand` initializes a pseudo-random number generator used by the MathProg translator, where the parameter `seed` may be any integer number.

A call to the routine `glp_mpl_init_rand` should immediately follow the call to the routine `glp_mpl_alloc_wksp`. However, using of this routine is optional. If it is not called, the effect is the same as if it were called with `seed` equal to 1.

3.2.4 glp_mpl_read_model — read and translate model section

Synopsis

```
int glp_mpl_read_model(glp_tran *tran, const char *fname, int skip);
```

Description

The routine `glp_mpl_read_model` reads model section and, optionally, data section, which may follow the model section, from a text file, whose name is the character string `fname`, performs translation of model statements and data blocks, and stores all the information in the workspace.

The parameter `skip` is a flag. If the input file contains the data section and this flag is non-zero, the data section is not read as if there were no data section and a warning message is printed. This allows reading data section(s) from other file(s).

Returns

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

3.2.5 glp_mpl_read_data — read and translate data section

Synopsis

```
int glp_mpl_read_data(glp_tran *tran, const char *fname);
```

Description

The routine `glp_mpl_read_data` reads data section from a text file, whose name is the character string `fname`, performs translation of data blocks, and stores the data read in the translator workspace. If necessary, this routine may be called more than once.

Returns

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

3.2.6 glp_mpl_generate — generate the model

Synopsis

```
int glp_mpl_generate(glp_tran *tran, const char *fname);
```

Description

The routine `glp_mpl_generate` generates the model using its description stored in the translator workspace. This operation means generating all variables, constraints, and objectives, executing check and display statements, which precede the solve statement (if it is presented).

The character string `fname` specifies the name of an output text file, to which output produced by display statements should be written. If `fname` is `NULL`, the output is sent to the terminal.

Returns

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

3.2.7 glp_mpl_build_prob — build problem instance from the model

Synopsis

```
void glp_mpl_build_prob(glp_tran *tran, glp_prob *P);
```

Description

The routine `glp_mpl_build_prob` obtains all necessary information from the translator workspace and stores it in the specified problem object `P`. Note that before building the current content of the problem object is erased with the routine `glp_erase_prob`.

3.2.8 glp_mpl_postsolve — postsolve the model

Synopsis

```
int glp_mpl_postsolve(glp_tran *tran, glp_prob *P, int sol);
```

Description

The routine `glp_mpl_postsolve` copies the solution from the specified problem object `prob` to the translator workspace and then executes all the remaining model statements, which follow the solve statement.

The parameter `sol` specifies which solution should be copied from the problem object to the workspace as follows:

`GLP_SOL` — basic solution;

`GLP_IPT` — interior-point solution;

`GLP_MIP` — mixed integer solution.

Returns

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

3.2.9 glp_mpl_free_wksp — free the translator workspace

Synopsis

```
void glp_mpl_free_wksp(glp_tran *tran);
```

Description

The routine `glp_mpl_free_wksp` frees all the memory allocated to the translator workspace. It also frees all other resources, which are still used by the translator.

3.3 Problem solution reading/writing routines

3.3.1 glp_print_sol — write basic solution in printable format

Synopsis

```
int glp_print_sol(glp_prob *P, const char *fname);
```

Description

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

Information reported by the routine `glp_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.

3.3.2 glp_read_sol — read basic solution in GLPK format

Synopsis

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

Description

The routine `glp_read_sol` reads basic solution from a text file in the GLPK format. (For description of the format see below.)

The character string `fname` specifies the 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_sol` decompresses it "on the fly".)

Returns

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

GLPK basic solution format

The GLPK basic solution format is a DIMACS-like format.[5] The file in this format is a plain ASCII text file containing lines of several types described below. A line is terminated with the end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

The first line of the solution file must be the solution line (except optional comment lines, which may precede the problem line). The last line of the data file must be the end line. Other lines may follow in arbitrary order, however, duplicate lines are not allowed.

[5] The DIMACS formats were developed by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) to facilitate exchange of problem data. For details see: `<http://dimacs.rutgers.edu/Challenges/>`.

Comment lines. Comment lines give human-readable information about the solution file and are ignored by GLPK routines. Comment lines can appear anywhere in the data file. Each comment line begins with the lower-case character `c`.

```
c This is an example of comment line
```

Solution line. There must be exactly one solution line in the solution file. This line must appear before any other lines except comment lines and has the following format:

```
s bas ROWS COLS PST DST OBJ
```

The lower-case letter `s` specifies that this is the solution line.

The three-character solution designator `bas` identifies the file as containing a basic solution to the LP problem instance.

The `ROWS` and `COLS` fields contain non-negative integer values that specify the number of rows (constraints) and columns (variables), resp., in the LP problem instance.

The `PST` and `DST` fields contain lower-case letters that specify the primal and dual solution status, resp., as follows:

`u` — solution is undefined;

`f` — solution is feasible;

`i` — solution is infeasible;

`n` — no feasible solution exists.

The `OBJ` field contains a floating-point number that specifies the objective function value in the basic solution.

Row solution descriptors. There must be exactly one row solution descriptor line in the solution file for each row (constraint). This line has the following format:

```
i ROW ST PRIM DUAL
```

The lower-case letter `i` specifies that this is the row solution descriptor line.

The `ROW` field specifies the row ordinal number, an integer between 1 and m, where m is the number of rows in the problem instance.

The `ST` field contains one of the following lower-case letters that specifies the row status in the basic solution:[6]

`b` — inactive constraint;

`l` — inequality constraint active on its lower bound;

`u` — inequality constraint active on its upper bound;

`f` — active free (unounded) row;

`s` — active equality constraint.

The `PRIM` field contains a floating-point number that specifies the row primal value (the value of the corresponding linear form).

The `DUAL` field contains a floating-point number that specifies the row dual value (the Lagrangian multiplier for active bound).

[6]The row status is the status of the associated auxiliary variable.

Column solution descriptors. There must be exactly one column solution descriptor line in the solution file for each column (variable). This line has the following format:

```
j COL ST PRIM DUAL
```

The lower-case letter `j` specifies that this is the column solution descriptor line.

The `COL` field specifies the column ordinal number, an integer between 1 and n, where n is the number of columns in the problem instance.

The `ST` field contains one of the following lower-case letters that specifies the column status in the basic solution:

`b` — basic variable;

`l` — non-basic variable having its lower bound active;

`u` — non-basic variable having its upper bound active;

`f` — non-basic free (unounded) variable;

`s` — non-basic fixed variable.

The `PRIM` field contains a floating-point number that specifies the column primal value.

The `DUAL` field contains a floating-point number that specifies the column dual value (the Lagrangian multiplier for active bound).

End line. There must be exactly one end line in the solution file. This line must appear last in the file and has the following format:

```
e
```

The lower-case letter `e` specifies that this is the end line. Anything that follows the end line is ignored by GLPK routines.

Example of solution file in GLPK basic solution format

The following example of a solution file in GLPK basic solution format specifies the optimal basic solution to the LP problem instance from Subsection "Example of MPS file".

```
s bas 7 7 f f 296.216606498195
i 1 s 2000 -0.0135956678700369
i 2 u 60 -2.56823104693141
i 3 b 83.9675090252707 0
i 4 u 40 -0.544404332129962
i 5 b 19.9602888086643 0
i 6 l 1500 0.251985559566788
i 7 l 250 0.48519855595668
j 1 l 0 0.253624548736462
j 2 b 665.342960288809 0
j 3 b 490.252707581226 0
j 4 b 424.187725631769 0
j 5 l 0 0.0145559566787004
j 6 b 299.638989169676 0
j 7 b 120.57761732852 0
e o f
```

3.3.3 glp_write_sol — write basic solution in GLPK format

Synopsis

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

Description

The routine `glp_write_sol` writes the current basic solution to a text file in the GLPK format. (For description of the GLPK basic solution format see Subsection "Read basic solution in GLPK format.")

The character string `fname` specifies the name of the text file to be written. (If the file name ends with suffix '`.gz`', the routine `glp_write_sol` compresses it "on the fly".)

Returns

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

3.3.4 glp_print_ipt — write interior-point solution in printable format

Synopsis

```
int glp_print_ipt(glp_prob *P, const char *fname);
```

Description

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

Information reported by the routine `glp_print_ipt` 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.

3.3.5 glp_read_ipt — read interior-point solution in GLPK format

Synopsis

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

Description

The routine `glp_read_ipt` reads interior-point solution from a text file in the GLPK format. (For description of the format see below.)

The character string `fname` specifies the 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_ipt` decompresses it "on the fly".)

Returns

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

GLPK interior-point solution format

The GLPK interior-point solution format is a DIMACS-like format.[7] The file in this format is a plain ASCII text file containing lines of several types described below. A line is terminated with the end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

The first line of the solution file must be the solution line (except optional comment lines, which may precede the problem line). The last line of the data file must be the end line. Other lines may follow in arbitrary order, however, duplicate lines are not allowed.

Comment lines. Comment lines give human-readable information about the solution file and are ignored by GLPK routines. Comment lines can appear anywhere in the data file. Each comment line begins with the lower-case character `c`.

```
c This is an example of comment line
```

Solution line. There must be exactly one solution line in the solution file. This line must appear before any other lines except comment lines and has the following format:

```
s ipt ROWS COLS SST OBJ
```

The lower-case letter `s` specifies that this is the solution line.

The three-character solution designator `ipt` identifies the file as containing an interior-point solution to the LP problem instance.

The `ROWS` and `COLS` fields contain non-negative integer values that specify the number of rows (constraints) and columns (variables), resp., in the LP problem instance.

The `SST` field contains one of the following lower-case letters that specifies the solution status:

`o` — solution is optimal;

`i` — solution is infeasible;

`n` — no feasible solution exists;

`u` — solution is undefined.

The `OBJ` field contains a floating-point number that specifies the objective function value in the interior-point solution.

Row solution descriptors. There must be exactly one row solution descriptor line in the solution file for each row (constraint). This line has the following format:

```
i ROW PRIM DUAL
```

The lower-case letter `i` specifies that this is the row solution descriptor line.

The `ROW` field specifies the row ordinal number, an integer between 1 and m, where m is the number of rows in the problem instance.

The `PRIM` field contains a floating-point number that specifies the row primal value (the value of the corresponding linear form).

The `DUAL` field contains a floating-point number that specifies the row dual value (the Lagrangian multiplier for active bound).

[7] The DIMACS formats were developed by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) to facilitate exchange of problem data. For details see: `<http://dimacs.rutgers.edu/Challenges/>`.

Column solution descriptors. There must be exactly one column solution descriptor line in the solution file for each column (variable). This line has the following format:

```
j COL PRIM DUAL
```

The lower-case letter `j` specifies that this is the column solution descriptor line.

The `COL` field specifies the column ordinal number, an integer between 1 and n, where n is the number of columns in the problem instance.

The `PRIM` field contains a floating-point number that specifies the column primal value.

The `DUAL` field contains a floating-point number that specifies the column dual value (the Lagrangian multiplier for active bound).

End line. There must be exactly one end line in the solution file. This line must appear last in the file and has the following format:

```
e
```

The lower-case letter `e` specifies that this is the end line. Anything that follows the end line is ignored by GLPK routines.

Example of solution file in GLPK interior-point solution format

The following example of a solution file in GLPK interior-point solution format specifies the optimal interior-point solution to the LP problem instance from Subsection "Example of MPS file".

```
s ipt 7 7 o 296.216606851403
i 1 2000.00000290369 -0.0135956757623443
i 2 60.0000001302903 -2.568231024875
i 3 83.9675094251819 -8.85591445202383e-10
i 4 39.9999999985064 -0.544404310443766
i 5 19.9602886941262 -2.24817803513554e-09
i 6 1500.00000199013 0.251985567257828
i 7 250.000000244896 0.48519856304979
j 1 3.3482079213784e-07 0.253624547432525
j 2 665.342955760768 6.04613825351601e-11
j 3 490.25271366802 5.8488360240978e-10
j 4 424.187729774275 -2.54144550490434e-11
j 5 1.46067738492801e-06 0.0145559574770786
j 6 299.638985053112 1.49359074902927e-10
j 7 120.577616852015 3.50297708781545e-10
e o f
```

3.3.6 glp_write_ipt — write interior-point solution in GLPK format

Synopsis

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

Description

The routine `glp_write_ipt` writes the current interior-point solution to a text file in the GLPK format. (For description of the GLPK interior-point solution format see Subsection "Read interior-point solution in GLPK format.")

The character string `fname` specifies the name of the text file to be written. (If the file name ends with suffix '`.gz`', the routine `glp_write_ipt` compresses it "on the fly".)

Returns

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

3.3.7 glp_print_mip — write MIP solution in printable format

Synopsis

```
int glp_print_mip(glp_prob *P, const char *fname);
```

Description

The routine `glp_print_mip` writes the current solution to a MIP problem, which is specified by the pointer `P`, to a text file, whose name is the character string `fname`, in printable format.

Information reported by the routine `glp_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.

3.3.8 glp_read_mip — read MIP solution in GLPK format

Synopsis

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

Description

The routine `glp_read_mip` reads MIP solution from a text file in the GLPK format. (For description of the format see below.)

The character string `fname` specifies the 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_mip` decompresses it "on the fly".)

Returns

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

GLPK MIP solution format

The GLPK MIP solution format is a DIMACS-like format.[8] The file in this format is a plain ASCII text file containing lines of several types described below. A line is terminated with the end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

The first line of the solution file must be the solution line (except optional comment lines, which may precede the problem line). The last line of the data file must be the end line. Other lines may follow in arbitrary order, however, duplicate lines are not allowed.

Comment lines. Comment lines give human-readable information about the solution file and are ignored by GLPK routines. Comment lines can appear anywhere in the data file. Each comment line begins with the lower-case character `c`.

```
c This is an example of comment line
```

Solution line. There must be exactly one solution line in the solution file. This line must appear

[8] The DIMACS formats were developed by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) to facilitate exchange of problem data. For details see: `<http://dimacs.rutgers.edu/Challenges/>`.

before any other lines except comment lines and has the following format:

```
s mip ROWS COLS SST OBJ
```

The lower-case letter `s` specifies that this is the solution line.

The three-character solution designator `mip` identifies the file as containing a solution to the MIP problem instance.

The `ROWS` and `COLS` fields contain non-negative integer values that specify the number of rows (constraints) and columns (variables), resp., in the LP problem instance.

The `SST` field contains one of the following lower-case letters that specifies the solution status:

`o` — solution is integer optimal;

`f` — solution is integer feasible (non-optimal);

`n` — no integer feasible solution exists;

`u` — solution is undefined.

The `OBJ` field contains a floating-point number that specifies the objective function value in the MIP solution.

Row solution descriptors. There must be exactly one row solution descriptor line in the solution file for each row (constraint). This line has the following format:

```
i ROW VAL
```

The lower-case letter `i` specifies that this is the row solution descriptor line.

The `ROW` field specifies the row ordinal number, an integer between 1 and m, where m is the number of rows in the problem instance.

The `VAL` field contains a floating-point number that specifies the row value (the value of the corresponding linear form).

Column solution descriptors. There must be exactly one column solution descriptor line in the solution file for each column (variable). This line has the following format:

```
j COL VAL
```

The lower-case letter `j` specifies that this is the column solution descriptor line.

The `COL` field specifies the column ordinal number, an integer between 1 and n, where n is the number of columns in the problem instance.

The `VAL` field contains a floating-point number that specifies the column value.

End line. There must be exactly one end line in the solution file. This line must appear last in the file and has the following format:

```
e
```

The lower-case letter `e` specifies that this is the end line. Anything that follows the end line is ignored by GLPK routines.

Example of solution file in GLPK MIP solution format

The following example of a solution file in GLPK MIP solution format specifies an optimal solution to a MIP problem instance.

```
s mip 8 8 o 1201500
i 1 60
i 2 8400
i 3 -1200
i 4 0
i 5 9000
i 6 -600
i 7 0
i 8 8000
j 1 60
j 2 6
j 3 0
j 4 60
j 5 6
j 6 600
j 7 60
j 8 16
e o f
```

3.3.9 glp_write_mip — write MIP solution in GLPK format

Synopsis

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

Description

The routine `glp_write_mip` writes the current MIP solution to a text file in the GLPK format. (For description of the GLPK MIP solution format see Subsection "Read MIP solution in GLPK format.")

The character string `fname` specifies the name of the text file to be written. (If the file name ends with suffix '`.gz`', the routine `glp_write_mip` compresses it "on the fly".)

Returns

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

3.4 Post-optimal analysis routines

3.4.1 glp_print_ranges — print sensitivity analysis report

Synopsis

```
int glp_print_ranges(glp_prob *P, int len, const int list[], int flags,
                     const char *fname);
```

Description

The routine `glp_print_ranges` performs sensitivity analysis of current optimal basic solution and writes the analysis report in human-readable format to a text file, whose name is the character string *fname*. (Detailed description of the report structure is given below.)

The parameter *len* specifies the length of the row/column list.

The array *list* specifies ordinal number of rows and columns to be analyzed. The ordinal numbers should be passed in locations *list*[1], *list*[2], ..., *list*[*len*]. Ordinal numbers from 1 to m refer to rows, and ordinal numbers from $m + 1$ to $m + n$ refer to columns, where m and n are, resp., the total number of rows and columns in the problem object. Rows and columns appear in the analysis report in the same order as they follow in the array list.

It is allowed to specify $len = 0$, in which case the array *list* is not used (so it can be specified as `NULL`), and the routine performs analysis for all rows and columns of the problem object.

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

On entry to the routine `glp_print_ranges` the current basic solution must be optimal and the basis factorization must exist. The application program can check that with the routine `glp_bf_exists`, and if the factorization does not exist, compute it with the routine `glp_factorize`. Note that if the LP preprocessor is not used, on normal exit from the simplex solver routine `glp_simplex` the basis factorization always exists.

Returns

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

Analysis report example

An example of the sensitivity analysis report is shown on the next two pages. This example corresponds to the example of LP problem described in Subsection "Example of MPS file".

Structure of the analysis report

For each row and column specified in the array *list* the routine prints two lines containing generic information and analysis information, which depends on the status of corresponding row or column.

Note that analysis of a row is analysis of its auxiliary variable, which is equal to the row linear form $\sum a_j x_j$, and analysis of a column is analysis of corresponding structural variable. Therefore, formally, on performing the sensitivity analysis there is no difference between rows and columns.

Problem: PLAN
Objective: VALUE = 296.2166065 (MINimum)

No.	Row name	St	Activity	Slack / Marginal	Lower bound / Upper bound	Activity range	Obj coef range	Obj value at break point	Limiting variable
1	VALUE	BS	296.21661	-296.21661	-Inf	299.25255	-1.00000	.	MN
				.	+Inf	296.21661	+Inf	+Inf	
2	YIELD	NS	2000.00000	.	2000.00000	1995.06864	-Inf	296.28365	BIN3
				-.01360	2000.00000	2014.03479	+Inf	296.02579	CU
3	FE	NU	60.00000	.	-Inf	55.89016	-Inf	306.77162	BIN4
				-2.56823	60.00000	62.69978	2.56823	289.28294	BIN3
4	CU	BS	83.96751	16.03249	-Inf	93.88467	-.30613	270.51157	MN
				.	100.00000	79.98213	.21474	314.24798	BIN5
5	MN	NU	40.00000	.	-Inf	34.42336	-Inf	299.25255	BIN4
				-.54440	40.00000	41.68691	.54440	295.29825	BIN3
6	MG	BS	19.96029	10.03971	-Inf	24.74427	-1.79618	260.36433	BIN1
				.	30.00000	9.40292	.28757	301.95652	MN
7	AL	NL	1500.00000	.	1500.00000	1485.78425	-.25199	292.63444	CU
				.25199	+Inf	1504.92126	+Inf	297.45669	BIN3
8	SI	NL	250.00000	50.00000	250.00000	235.32871	-.48520	289.09812	CU
				.48520	300.00000	255.06073	+Inf	298.67206	BIN3

Problem: PLAN
Objective: VALUE = 296.2166065 (MINimum)

No.	Column name	St	Activity	Obj coef / Marginal	Lower bound / Upper bound	Activity range	Obj coef range	Obj value at break point	Limiting variable
1	BIN1	NL	.	.03000	.	-28.82475	-.22362	288.90594	BIN4
				.25362	200.00000	33.88040	+Inf	304.80951	BIN4
2	BIN2	BS	665.34296	.08000	.	802.22222	.01722	254.44822	BIN1
				.	2500.00000	313.43066	.08863	301.95652	MN
3	BIN3	BS	490.25271	.17000	400.00000	788.61314	.15982	291.22807	MN
				.	800.00000	-347.42857	.17948	300.86548	BIN5
4	BIN4	BS	424.18773	.12000	100.00000	710.52632	.10899	291.54745	MN
				.	700.00000	-256.15524	.14651	307.46010	BIN1
5	BIN5	NL	.	.15000	.	-201.78739	.13544	293.27940	BIN3
				.01456	1500.00000	58.79586	+Inf	297.07244	BIN3
6	ALUM	BS	299.63899	.21000	.	358.26772	.18885	289.87879	AL
				.	+Inf	112.40876	.22622	301.07527	MN
7	SILICON	BS	120.57762	.38000	.	124.27093	.14828	268.27586	BIN5
				.	+Inf	85.54745	.46667	306.66667	MN

End of report

Generic information

`No.` is the row or column ordinal number in the problem object. Rows are numbered from 1 to m, and columns are numbered from 1 to n, where m and n are, resp., the total number of rows and columns in the problem object.

`Row name` is the symbolic name assigned to the row. If the row has no name assigned, this field contains blanks.

`Column name` is the symbolic name assigned to the column. If the column has no name assigned, this field contains blanks.

`St` is the status of the row or column in the optimal solution:

`BS` — non-active constraint (row), basic column;

`NL` — inequality constraint having its lower right-hand side active (row), non-basic column having its lower bound active;

`NU` — inequality constraint having its upper right-hand side active (row), non-basic column having its upper bound active;

`NS` — active equality constraint (row), non-basic fixed column.

`NF` — active free row, non-basic free (unbounded) column. (This case means that the optimal solution is dual degenerate.)

`Activity` is the (primal) value of the auxiliary variable (row) or structural variable (column) in the optimal solution.

`Slack` is the (primal) value of the row slack variable.

`Obj coef` is the objective coefficient of the column (structural variable).

`Marginal` is the reduced cost (dual activity) of the auxiliary variable (row) or structural variable (column).

`Lower bound` is the lower right-hand side (row) or lower bound (column). If the row or column has no lower bound, this field contains `-Inf`.

`Upper bound` is the upper right-hand side (row) or upper bound (column). If the row or column has no upper bound, this field contains `+Inf`.

Sensitivity analysis of active bounds

The sensitivity analysis of active bounds is performed only for rows, which are active constraints, and only for non-basic columns, because inactive constraints and basic columns have no active bounds.

For every auxiliary (row) or structural (column) non-basic variable the routine starts changing its active bound in both direction. The first of the two lines in the report corresponds to decreasing, and the second line corresponds to increasing of the active bound. Since the variable being analyzed is non-basic, its activity, which is equal to its active bound, also starts changing. This changing leads to changing of basic (auxiliary and structural) variables, which depend on the non-basic variable. The current basis remains primal feasible and therefore optimal while values of all basic variables are primal feasible, i.e. are within their bounds. Therefore, if some basic variable called the *limiting variable* reaches its (lower or upper) bound first, before any other basic variables, it thereby limits further changing of the non-basic variable, because otherwise the current basis would become primal infeasible. The point, at which this happens, is called the *break point*. Note that

there are two break points: the lower break point, which corresponds to decreasing of the non-basic variable, and the upper break point, which corresponds to increasing of the non-basic variable.

In the analysis report values of the non-basic variable (i.e. of its active bound) being analyzed at both lower and upper break points are printed in the field '`Activity range`'. Corresponding values of the objective function are printed in the field '`Obj value at break point`', and symbolic names of corresponding limiting basic variables are printed in the field '`Limiting variable`'. If the active bound can decrease or/and increase unlimitedly, the field '`Activity range`' contains `-Inf` or/and `+Inf`, resp.

For example (see the example report above), row SI is a double-sided constraint, which is active on its lower bound (right-hand side), and its activity in the optimal solution being equal to the lower bound is 250. The activity range for this row is $[235.32871, 255.06073]$. This means that the basis remains optimal while the lower bound is increasing up to 255.06073, and further increasing is limited by (structural) variable BIN3. If the lower bound reaches this upper break point, the objective value becomes equal to 298.67206.

Note that if the basis does not change, the objective function depends on the non-basic variable linearly, and the per-unit change of the objective function is the reduced cost (marginal value) of the non-basic variable.

Sensitivity analysis of objective coefficients at non-basic variables

The sensitivity analysis of the objective coefficient at a non-basic variable is quite simple, because in this case change in the objective coefficient leads to equivalent change in the reduced cost (marginal value).

For every auxiliary (row) or structural (column) non-basic variable the routine starts changing its objective coefficient in both direction. (Note that auxiliary variables are not included in the objective function and therefore always have zero objective coefficients.) The first of the two lines in the report corresponds to decreasing, and the second line corresponds to increasing of the objective coefficient. This changing leads to changing of the reduced cost of the non-basic variable to be analyzed and does affect reduced costs of all other non-basic variables. The current basis remains dual feasible and therefore optimal while the reduced cost keeps its sign. Therefore, if the reduced cost reaches zero, it limits further changing of the objective coefficient (if only the non-basic variable is non-fixed).

In the analysis report minimal and maximal values of the objective coefficient, on which the basis remains optimal, are printed in the field '`Obj coef range`'. If the objective coefficient can decrease or/and increase unlimitedly, this field contains `-Inf` or/and `+Inf`, resp.

For example (see the example report above), column BIN5 is non-basic having its lower bound active. Its objective coefficient is 0.15, and reduced cost in the optimal solution 0.01456. The column lower bound remains active while the column reduced cost remains non-negative, thus, minimal value of the objective coefficient, on which the current basis still remains optimal, is $0.15 - 0.01456 = 0.13644$, that is indicated in the field '`Obj coef range`'.

Sensitivity analysis of objective coefficients at basic variables

To perform sensitivity analysis for every auxiliary (row) or structural (column) variable the routine starts changing its objective coefficient in both direction. (Note that auxiliary variables are not included in the objective function and therefore always have zero objective coefficients.) The first of the two lines in the report corresponds to decreasing, and the second line corresponds to increasing of the objective coefficient. This changing leads to changing of reduced costs of non-basic

variables. The current basis remains dual feasible and therefore optimal while reduced costs of all non-basic variables (except fixed variables) keep their signs. Therefore, if the reduced cost of some non-basic non-fixed variable called the *limiting variable* reaches zero first, before reduced cost of any other non-basic non-fixed variable, it thereby limits further changing of the objective coefficient, because otherwise the current basis would become dual infeasible (non-optimal). The point, at which this happens, is called the *break point*. Note that there are two break points: the lower break point, which corresponds to decreasing of the objective coefficient, and the upper break point, which corresponds to increasing of the objective coefficient. Let the objective coefficient reach its limit value and continue changing a bit further in the same direction that makes the current basis dual infeasible (non-optimal). Then the reduced cost of the non-basic limiting variable becomes "a bit" dual infeasible that forces the limiting variable to enter the basis replacing there some basic variable, which leaves the basis to keep its primal feasibility. It should be understood that if we change the current basis in this way exactly at the break point, both the current and adjacent bases will be optimal with the same objective value, because at the break point the limiting variable has zero reduced cost. On the other hand, in the adjacent basis the value of the limiting variable changes, because there it becomes basic, that leads to changing of the value of the basic variable being analyzed. Note that on determining the adjacent basis the bounds of the analyzed basic variable are ignored as if it were a free (unbounded) variable, so it cannot leave the current basis.

In the analysis report lower and upper limits of the objective coefficient at the basic variable being analyzed, when the basis remains optimal, are printed in the field '`Obj coef range`'. Corresponding values of the objective function at both lower and upper break points are printed in the field '`Obj value at break point`', symbolic names of corresponding non-basic limiting variables are printed in the field '`Limiting variable`', and values of the basic variable, which it would take on in the adjacent bases (as was explained above) are printed in the field '`Activity range`'. If the objective coefficient can increase or/and decrease unlimitedly, the field '`Obj coef range`' contains `-Inf` and/or `+Inf`, resp. It also may happen that no dual feasible adjacent basis exists (i.e. on entering the basis the limiting variable can increase or decrease unlimitedly), in which case the field '`Activity range`' contains `-Inf` and/or `+Inf`.

For example (see the example report above), structural variable (column) BIN3 is basic, its optimal value is 490.25271, and its objective coefficient is 0.17. The objective coefficient range for this column is $[0.15982, 0.17948]$. This means that the basis remains optimal while the objective coefficient is decreasing down to 0.15982, and further decreasing is limited by (auxiliary) variable MN. If we make the objective coefficient a bit less than 0.15982, the limiting variable MN will enter the basis, and in that adjacent basis the structural variable BIN3 will take on new optimal value 788.61314. At the lower break point, where the objective coefficient is exactly 0.15982, the objective function takes on the value 291.22807 in both the current and adjacent bases.

Note that if the basis does not change, the objective function depends on the objective coefficient at the basic variable linearly, and the per-unit change of the objective function is the value of the basic variable.

Chapter 4

Advanced API Routines

4.1 Background

Using vector and matrix notations the LP problem (1.1)—(1.3) (see Section **??**, page **??**) 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

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

where:

$x_R = (x_1, \ldots, x_m)$ is the vector of auxiliary variables;

$x_S = (x_{m+1}, \ldots, x_{m+n})$ is the vector of structural variables;

z is the objective function;

$c = (c_1, \ldots, c_n)$ is the vector of objective coefficients;

c_0 is the constant term ("shift") of the objective function;

$A = (a_{11}, \ldots, a_{mn})$ is the constraint matrix;

$l_R = (l_1, \ldots, l_m)$ is the vector of lower bounds of auxiliary variables;

$u_R = (u_1, \ldots, u_m)$ is the vector of upper bounds of auxiliary variables;

$l_S = (l_{m+1}, \ldots, l_{m+n})$ is the vector of lower bounds of structural variables;

$u_S = (u_{m+1}, \ldots, u_{m+n})$ 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

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

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 hyperplanes 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 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

[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 at a given point it is satisfied as equality, otherwise it is called *inactive*.

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:

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

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]

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

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

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

$$\lambda_l \ge 0, \quad \lambda_u \le 0 \quad \text{(minimization)} \tag{3.17}$$

$$\lambda_l \le 0, \quad \lambda_u \ge 0 \quad \text{(maximization)} \tag{3.18}$$

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

where:

$\pi = (\pi_1, \ldots, \pi_m)$ is a m-vector of Lagrange multipliers for equality constraints (3.5);

$\lambda_l = [(\lambda_l)_1, \ldots, (\lambda_l)_n]$ is a n-vector of Lagrange multipliers for lower bound constraints (3.6);

$\lambda_u = [(\lambda_u)_1, \ldots, (\lambda_u)_n]$ 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.

[4] This definition corresponds to the GLPK implementation.

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

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.

In GLPK two multipliers $(\lambda_l)_k$ and $(\lambda_u)_k$ for each primal variable x_k, $k = 1, \ldots, m + n$, 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 an 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:

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

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)):

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

[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 $l_k = u_k$.

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:

$$\begin{cases} B^T \pi + \lambda_B = c_B \\ N^T \pi + \lambda_N = c_N \end{cases} \tag{3.28}$$

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.

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.[8] Therefore, a basic solution[9] 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:

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

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

[8] Until double-bounded variables appear.

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

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.

4.2 LP basis routines

4.2.1 glp_bf_exists — check if the basis factorization exists

Synopsis

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

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:[10]

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

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.

4.2.2 glp_factorize — compute the basis factorization

Synopsis

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

Description

The routine `glp_factorize` computes the basis factorization for the current basis associated with the specified problem object.[11]

[10] For more details see Subsection **??**, page **??**.

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

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.

4.2.3 glp_bf_updated — check if the basis factorization has been updated

Synopsis

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

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".

4.2.4 glp_get_bfcp — retrieve basis factorization control parameters

Synopsis

```
void glp_get_bfcp(glp_prob *P, 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.

4.2.5 glp_set_bfcp — change basis factorization control parameters

Synopsis

```
void glp_set_bfcp(glp_prob *P, 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_LUF + GLP_BF_FT`)

Basis factorization type:

`GLP_BF_LUF + GLP_BF_FT` — LUF, Forrest–Tomlin update;

`GLP_BF_LUF + GLP_BF_BG` — LUF, Schur complement, Bartels–Golub update;

`GLP_BF_LUF + GLP_BF_GR` — LUF, Schur complement, Givens rotation update;

`GLP_BF_BTF + GLP_BF_BG` — BTF, Schur complement, Bartels–Golub update;

`GLP_BF_BTF + GLP_BF_GR` — BTF, 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.

`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.

`int nfs_max` (default: `100`)

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, $0 < \texttt{upd_tol} < 1$, 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 $\texttt{upd_tol} \cdot \max(|u_{k*}|, |u_{*k}|)$, the factorization is considered as inaccurate.

`int nrs_max` (default: `100`)

Maximal number of additional rows and columns, $\texttt{nrs_max} \geq 1$, 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.

4.2.6 glp_get_bhead — retrieve the basis header information

Synopsis

```
int glp_get_bhead(glp_prob *P, 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:[12]

$$\begin{pmatrix} x_B \\ x_N \end{pmatrix} = \Pi \begin{pmatrix} x_R \\ x_S \end{pmatrix} \quad \Leftrightarrow \quad \begin{pmatrix} x_R \\ x_S \end{pmatrix} = \Pi^T \begin{pmatrix} x_B \\ x_N \end{pmatrix},$$

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,[13] 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.

[12] For more details see Subsection ??, page ??.

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

4.2.7 glp_get_row_bind — retrieve row index in the basis header

Synopsis

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

Returns

The routine `glp_get_row_bind` returns the index k of basic variable $(x_B)_k$, $1 \le k \le m$, which is i-th auxiliary variable (that is, the auxiliary variable corresponding to i-th row), $1 \le i \le m$, 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 inversion of the routine `glp_get_bhead`; that is, if `glp_get_bhead`(P, k) returns i, `glp_get_row_bind`(P, i) returns k, and vice versa.

4.2.8 glp_get_col_bind — retrieve column index in the basis header

Synopsis

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

Returns

The routine `glp_get_col_bind` returns the index k of basic variable $(x_B)_k$, $1 \le k \le m$, which is j-th structural variable (that is, the structural variable corresponding to j-th column), $1 \le j \le n$, 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 inversion of the routine `glp_get_bhead`; that is, if `glp_get_bhead`(P, k) returns $m + j$, `glp_get_col_bind`(P, j) returns k, and vice versa.

4.2.9 glp_ftran — perform forward transformation

Synopsis

```
void glp_ftran(glp_prob *P, 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.

4.2.10 glp_btran — perform backward transformation

Synopsis

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

Description

The routine `glp_btran` performs backward transformation (BTRAN), i.e. it solves the system $B^T x = b$, 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.

4.2.11 glp_warm_up — "warm up" LP basis

Synopsis

```
int glp_warm_up(glp_prob *P);
```

Description

The routine `glp_warm_up` "warms up" the LP basis for the specified problem object using current statuses assigned to rows and columns (that is, to auxiliary and structural variables).

This operation includes computing factorization of the basis matrix (if it does not exist), computing primal and dual components of basic solution, and determining the solution status.

Returns

0	The operation has been successfully performed.
`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.

4.3 Simplex tableau routines

4.3.1 glp_eval_tab_row — compute row of the tableau

Synopsis

```
int glp_eval_tab_row(glp_prob *P, 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 $1 \le k \le m$, the basic variable is k-th auxiliary variable, and if $m+1 \le k \le m+n$, 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 $\xi_{i1}, \xi_{i2}, \ldots, \xi_{in}$ are elements of the simplex table row, $(x_N)_1, (x_N)_2, \ldots, (x_N)_n$ 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 $0 \le$ `len` $\le n$ 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.

4.3.2 glp_eval_tab_col — compute column of the tableau

Synopsis

```
int glp_eval_tab_col(glp_prob *P, 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 $1 \le k \le m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \le k \le m + n$, 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 $x_k = (x_N)_j$:

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

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 $0 \le$ `len` $\le m$ 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.

4.3.3 glp_transform_row — transform explicitly specified row

Synopsis

```
int glp_transform_row(glp_prob *P, int len, int ind[], double val[]);
```

Description

The routine `glp_transform_row` performs the same operation as the routine `glp_eval_tab_row` with exception that the row to be transformed is specified explicitly as a sparse vector.

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

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

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 specified 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 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:

$$x = \xi_1 (x_N)_1 + \xi_2 (x_N)_2 + \ldots + \xi_n (x_N)_n,$$

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

On exit the routine stores indices and numerical values of non-zero coefficients ξ_j of the resultant row in locations `ind[1]`, ..., `ind[len']` and `val[1]`, ..., `val[len']`, where $0 \leq \mathtt{len}' \leq n$ 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 `glp_transform_row` returns `len'`, the number of non-zero coefficients in the resultant row stored in the arrays `ind` and `val`.

4.3.4 glp_transform_col — transform explicitly specified column

Synopsis

```
int glp_transform_col(glp_prob *P, int len, int ind[], double val[]);
```

Description

The routine `glp_transform_col` performs the same operation as the routine `glp_eval_tab_col` with exception that the column to be transformed is specified explicitly as a sparse vector.

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

$$\begin{aligned} x_1 &= a_{11}x_{m+1} + \ldots + a_{1n}x_{m+n} + a_1x \\ x_2 &= a_{21}x_{m+1} + \ldots + a_{2n}x_{m+n} + a_2x \\ &\ldots\ldots\ldots\ldots\ldots\ldots \\ x_m &= a_{m1}x_{m+1} + \ldots + a_{mn}x_{m+n} + a_mx \end{aligned}$$

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 at x.

On entry row indices and numerical values of non-zero coefficients a_i of the specified 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 (as if the transformed column were added to the problem object and the variable x were non-basic):

$$\begin{aligned} (x_B)_1 &= \ldots + \xi_1x \\ (x_B)_2 &= \ldots + \xi_2x \\ &\ldots\ldots\ldots \\ (x_B)_m &= \ldots + \xi_mx \end{aligned}$$

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

On exit the routine stores indices and numerical values of non-zero coefficients ξ_i of the resultant column in locations `ind[1]`, ..., `ind[len']` and `val[1]`, ..., `val[len']`, where $0 \le \mathtt{len}' \le m$ 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 `glp_transform_col` returns `len'`, the number of non-zero coefficients in the resultant column stored in the arrays `ind` and `val`.

4.3.5 glp_prim_rtest — perform primal ratio test

Synopsis

```
int glp_prim_rtest(glp_prob *P, int len, const int ind[], const double val[],
                   int dir, double eps);
```

Description

The routine `glp_prim_rtest` performs the primal ratio test using an explicitly specified column of the simplex table.

The current basic solution associated with the LP problem object must be primal feasible.

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

$$\begin{aligned}(x_B)_1 &= \ldots + \xi_1 x \\ (x_B)_2 &= \ldots + \xi_2 x \\ &\ldots\ldots \\ (x_B)_m &= \ldots + \xi_m x\end{aligned}$$

The column is specifed on entry to the routine in 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 ξ_i should be placed in locations `val[1]`, ..., `val[len]`. The arrays `ind` and `val` are not changed by the routine.

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

The parameter `eps` is an absolute tolerance (small positive number, say, 10^{-9}) used by the routine to skip ξ_i's whose magnitude is less than `eps`.

The routine determines which basic variable (among those specified in `ind[1]`, ..., `ind[len]`) reaches its (lower or upper) bound first before any other basic variables do, and which, therefore, should leave the basis in order to keep primal feasibility.

Returns

The routine `glp_prim_rtest` returns the index, `piv`, in the arrays `ind` and `val` corresponding to the pivot element chosen, $1 \le$ `piv` $\le$ `len`. If the adjacent basic solution is primal unbounded, and therefore the choice cannot be made, the routine returns zero.

Comments

If the non-basic variable x is presented in the LP problem object, the input column can be computed with the routine `glp_eval_tab_col`; otherwise, it can be computed with the routine `glp_transform_col`.

4.3.6 glp_dual_rtest — perform dual ratio test

Synopsis

```
int glp_dual_rtest(glp_prob *P, int len, const int ind[], const double val[],
                   int dir, double eps);
```

Description

The routine `glp_dual_rtest` performs the dual ratio test using an explicitly specified row of the simplex table.

The current basic solution associated with the LP problem object must be dual feasible.

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

$$x = \xi_1(x_N)_1 + \xi_2(x_N)_2 + \ldots + \xi_n(x_N)_n.$$

The row is specified on entry to the routine in 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 ξ_j should be placed in locations `val[1]`, ..., `val[len]`. The arrays `ind` and `val` are not changed by the routine.

The parameter `dir` specifies direction in which the variable x changes on leaving the basis: $+1$ means that x goes on its lower bound, so its reduced cost (dual variable) is increasing (minimization) or decreasing (maximization); -1 means that x goes on its upper bound, so its reduced cost is decreasing (minimization) or increasing (maximization).

The parameter `eps` is an absolute tolerance (small positive number, say, 10^{-9}) used by the routine to skip ξ_j's whose magnitude is less than `eps`.

The routine determines which non-basic variable (among those specified in `ind[1]`, ..., `ind[len]`) should enter the basis in order to keep dual feasibility, because its reduced cost reaches the (zero) bound first before this occurs for any other non-basic variables.

Returns

The routine `glp_dual_rtest` returns the index, `piv`, in the arrays `ind` and `val` corresponding to the pivot element chosen, $1 \le$ `piv` $\le$ `len`. If the adjacent basic solution is dual unbounded, and therefore the choice cannot be made, the routine returns zero.

Comments

If the basic variable x is presented in the LP problem object, the input row can be computed with the routine `glp_eval_tab_row`; otherwise, it can be computed with the routine `glp_transform_row`.

4.4 Post-optimal analysis routines

4.4.1 glp_analyze_bound — analyze active bound of non-basic variable

Synopsis

```
void glp_analyze_bound(glp_prob *P, int k, double *limit1, int *var1,
                       double *limit2, int *var2);
```

Description

The routine `glp_analyze_bound` analyzes the effect of varying the active bound of specified non-basic variable.

The non-basic variable is specified by the parameter k, where $1 \leq k \leq m$ means auxiliary variable of corresponding row, and $m + 1 \leq k \leq m + n$ means structural variable (column).

Note that the current basic solution must be optimal, and the basis factorization must exist.

Results of the analysis have the following meaning.

`value1` is the minimal value of the active bound, at which the basis still remains primal feasible and thus optimal. `-DBL_MAX` means that the active bound has no lower limit.

`var1` is the ordinal number of an auxiliary (1 to m) or structural ($m+1$ to $m+n$) basic variable, which reaches its bound first and thereby limits further decreasing the active bound being analyzed. if `value1` = `-DBL_MAX`, `var1` is set to 0.

`value2` is the maximal value of the active bound, at which the basis still remains primal feasible and thus optimal. `+DBL_MAX` means that the active bound has no upper limit.

`var2` is the ordinal number of an auxiliary (1 to m) or structural ($m+1$ to $m+n$) basic variable, which reaches its bound first and thereby limits further increasing the active bound being analyzed. if `value2` = `+DBL_MAX`, `var2` is set to 0.

The parameters `value1`, `var1`, `value2`, `var2` can be specified as `NULL`, in which case corresponding information is not stored.

4.4.2 glp_analyze_coef — analyze objective coefficient at basic variable

Synopsis

```
void glp_analyze_coef(glp_prob *P, int k,
                      double *coef1, int *var1, double *value1,
                      double *coef2, int *var2, double *value2);
```

Description

The routine `glp_analyze_coef` analyzes the effect of varying the objective coefficient at specified basic variable.

The basic variable is specified by the parameter k, where $1 \leq k \leq m$ means auxiliary variable of corresponding row, and $m + 1 \leq k \leq m + n$ means structural variable (column).

Note that the current basic solution must be optimal, and the basis factorization must exist.

Results of the analysis have the following meaning.

`coef1` is the minimal value of the objective coefficient, at which the basis still remains dual feasible and thus optimal. `-DBL_MAX` means that the objective coefficient has no lower limit.

`var1` is the ordinal number of an auxiliary (1 to m) or structural ($m + 1$ to $m + n$) non-basic variable, whose reduced cost reaches its zero bound first and thereby limits further decreasing the objective coefficient being analyzed. If `coef1` = `-DBL_MAX`, `var1` is set to 0.

`value1` is value of the basic variable being analyzed in an adjacent basis, which is defined as follows. Let the objective coefficient reach its minimal value (`coef1`) and continue decreasing. Then the reduced cost of the limiting non-basic variable (`var1`) becomes dual infeasible and the current basis becomes non-optimal that forces the limiting non-basic variable to enter the basis replacing there some basic variable that leaves the basis to keep primal feasibility. Should note that on determining the adjacent basis current bounds of the basic variable being analyzed are ignored as if it were free (unbounded) variable, so it cannot leave the basis. It may happen that no dual feasible adjacent basis exists, in which case `value1` is set to `-DBL_MAX` or `+DBL_MAX`.

`coef2` is the maximal value of the objective coefficient, at which the basis still remains dual feasible and thus optimal. `+DBL_MAX` means that the objective coefficient has no upper limit.

`var2` is the ordinal number of an auxiliary (1 to m) or structural ($m + 1$ to $m + n$) non-basic variable, whose reduced cost reaches its zero bound first and thereby limits further increasing the objective coefficient being analyzed. If `coef2` = `+DBL_MAX`, `var2` is set to 0.

`value2` is value of the basic variable being analyzed in an adjacent basis, which is defined exactly in the same way as `value1` above with exception that now the objective coefficient is increasing.

The parameters `coef1`, `var1`, `value1`, `coef2`, `var2`, `value2` can be specified as `NULL`, in which case corresponding information is not stored.

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 *T, 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:

```
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.

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

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

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

```
void foo_bar(glp_tree *T, void *info)
{     . . .
      switch (glp_ios_reason(T))
      {  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 $z^{best} := +\infty$ (in case of minimization) or $z^{best} := -\infty$ (in case of maximization), where z^{best} is *incumbent value*, i.e. an upper (minimization) or lower (maximization) global bound for z^{opt}, the optimal objective value for P^0.

2. Subproblem selection

If $L = \varnothing$ 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 $z^{LP} \geq z^{best}$ (minimization) or $z^{LP} \leq z^{\text{best}}$ (), 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 $x_j^{LP} \in x^{LP}$ be its value in the optimal solution to P^{LP}.

If x_j^{LP} are integral for all integer variables, then a better integer feasible solution is found. Store its components, set $z^{best} := z^{LP}$, 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 $x_j^{LP} \in x^{LP}$ 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 $\lfloor x_j^{LP} \rfloor$. (For example, if $x_j^{LP} = 3.14$, 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 $\lceil x_j^{LP} \rceil$. (For example, if $x_j^{LP} = 3.14$, 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 L and add two new subproblems P^D and P^U to it. 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 z^{best}, i.e. set $L := L \backslash \{P\}$ for all P, where $\widetilde{z} \geq z^{best}$ (in case of minimization) or $\widetilde{z} \leq z^{best}$ (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 $z^{best} = +\infty$ (in case of minimization) or $z^{best} = -\infty$ (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 with $z^{opt} = z^{best}$. STOP.

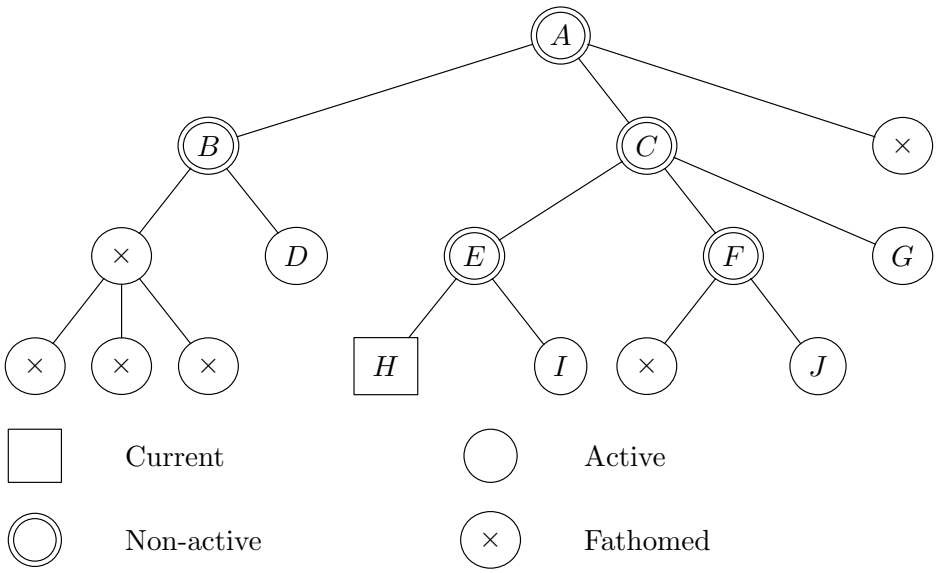


Fig. 1. An example of the search tree.

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.

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

[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.

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.

Note that components of optimal solution to 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 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

[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.

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 glp_ios_reason — determine reason for calling the callback routine

Synopsis

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

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 glp_ios_get_prob — access the problem object

Synopsis

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

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 glp_ios_row_attr — determine additional row attributes

Synopsis

```
void glp_ios_row_attr(glp_tree *T, 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 glp_ios_mip_gap — compute relative MIP gap

Synopsis

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

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 glp_ios_node_data — access application-specific data

Synopsis

```
void *glp_ios_node_data(glp_tree *T, 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 glp_ios_select_node — select subproblem to continue the search

Synopsis

```
void glp_ios_select_node(glp_tree *T, 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 glp_ios_heur_sol — provide solution found by heuristic

Synopsis

```
int glp_ios_heur_sol(glp_tree *T, 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 $x[1], \ldots, x[n]$, 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 glp_ios_can_branch — check if can branch upon specified variable

Synopsis

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

Returns

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

5.2.9 glp_ios_branch_upon — choose variable to branch upon

Synopsis

```
void glp_ios_branch_upon(glp_tree *T, 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:

`GLP_DN_BRNCH` — select down-branch;

`GLP_UP_BRNCH` — select up-branch;

`GLP_NO_BRNCH` — use general selection technique.

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 glp_ios_terminate — terminate the solution process

Synopsis

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

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 glp_ios_tree_size — determine size of the search tree

Synopsis

```
void glp_ios_tree_size(glp_tree *T, 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 glp_ios_curr_node — determine current active subproblem

Synopsis

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

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 glp_ios_next_node — determine next active subproblem

Synopsis

```
int glp_ios_next_node(glp_tree *T, 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 glp_ios_prev_node — determine previous active subproblem

Synopsis

```
int glp_ios_prev_node(glp_tree *T, 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 glp_ios_up_node — determine parent subproblem

Synopsis

```
int glp_ios_up_node(glp_tree *T, 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 glp_ios_node_level — determine subproblem level

Synopsis

```
int glp_ios_node_level(glp_tree *T, 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 glp_ios_node_bound — determine subproblem local bound

Synopsis

```
double glp_ios_node_bound(glp_tree *T, 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 glp_ios_best_node — find active subproblem with best local bound

Synopsis

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

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 glp_ios_pool_size — determine current size of the cut pool

Synopsis

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

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 glp_ios_add_row — add constraint to the cut pool

Synopsis

```
int glp_ios_add_row(glp_tree *T, 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:

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

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.

Ordinal numbers of structural variables (i.e. column indices) $j \in J$ and numerical values of corresponding constraint coefficients a_j should 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 $\Sigma a_j x_j \leq b$;

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

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 glp_ios_del_row — remove constraint from the cut pool

Synopsis

```
void glp_ios_del_row(glp_tree *T, 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.

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

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));
```

5.4.4 glp_ios_clear_pool — remove all constraints from the cut pool

Synopsis

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

Description

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

Chapter 6

Miscellaneous API Routines

6.1 GLPK environment routines

6.1.1 glp_init_env — initialize GLPK environment

Synopsis

```
int glp_init_env(void);
```

Description

The routine `glp_init_env` initializes the GLPK environment. Normally the application program does not need to call this routine, because it is called automatically on the first call to any API routine.

Returns

0	initialization successful;
1	environment is already initialized;
2	initialization failed (insufficient memory);
3	initialization failed (unsupported programming model).

6.1.2 glp_version — 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"`.

6.1.3 glp_free_env — free GLPK environment

Synopsis

```
int 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.

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.

Returns

0	termination successful;
1	environment is inactive (was not initialized).

6.1.4 glp_printf — write formatted output to terminal

Synopsis

```
void glp_printf(const char *fmt, ...);
```

Description

The routine `glp_printf` uses the format control string `fmt` to format its parameters and writes the formatted output to the terminal.

This routine is a replacement of the standard C function `printf` and used by all GLPK routines to perform terminal output. The application program may use `glp_printf` for the same purpose that allows controlling its terminal output with the routines `glp_term_out` and `glp_term_hook`.

6.1.5 glp_vprintf — write formatted output to terminal

Synopsis

```
void glp_vprintf(const char *fmt, va_list arg);
```

Description

The routine `glp_vprintf` uses the format control string `fmt` to format its parameters specified by the list `arg` and writes the formatted output to the terminal.

This routine is a replacement of the standard C function `vprintf` and used by all GLPK routines to perform terminal output. The application program may use `glp_vprintf` for the same purpose that allows controlling its terminal output with the routines `glp_term_out` and `glp_term_hook`.

6.1.6 glp_term_out — enable/disable terminal output

Synopsis

```
int 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.

Returns

The routine `glp_term_out` returns the previous value of the terminal output flag.

6.1.7 glp_term_hook — 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;
      . . .
      glp_term_hook(hook, foo); /* redirect terminal output */
      . . .
      glp_term_hook(NULL, NULL); /* resume terminal output */
      . . .
}
```

6.1.8 glp_open_tee — start copying terminal output

Synopsis

```
int glp_open_tee(const char *fname);
```

Description

The routine `glp_open_tee` starts copying all the terminal output to an output text file, whose name is specified by the character string `fname`.

Returns

0	operation successful;
1	copying terminal output is already active;
2	unable to create output file.

6.1.9 glp_close_tee — stop copying terminal output

Synopsis

```
int glp_close_tee(void);
```

Description

The routine `glp_close_tee` stops copying the terminal output to the output text file previously open by the routine `glp_open_tee` closing that file.

Returns

0	operation successful;
1	copying terminal output was not started.

6.1.10 glp_error — display error message and terminate execution

Synopsis

```
void glp_error(const char *fmt, ...);
```

Description

The routine `glp_error` (implemented as a macro) formats its parameters using the format control string `fmt`, writes the formatted message to the terminal, and then abnormally terminates the program.

6.1.11 glp_at_error — check for error state

Synopsis

```
int glp_at_error(void);
```

Description

The routine `glp_at_error` checks if the GLPK environment is at error state, i. e. if the call to the routine is (indirectly) made from the `glp_error` routine via an user-defined hook routine.

This routine can be used, for example, by a custom output handler (installed with the routine `glp_term_hook`) to determine whether or not the message to be displayed is an error message.

Returns

If the GLPK environment is at error state, the routine returns non-zero, otherwise zero.

6.1.12 glp_assert — check logical condition

Synopsis

```
void glp_assert(int expr);
```

Description

The routine `glp_assert` (implemented as a macro) checks a logical condition specified by the expression `expr`. If the condition is true (non-zero), the routine does nothing; otherwise, if the condition is false (zero), the routine prints an error message and abnormally terminates the program.

This routine is a replacement of the standard C function `assert` and used by all GLPK routines to check program logic. The application program may use `glp_assert` for the same purpose.

6.1.13 glp_error_hook — install hook to intercept abnormal termination

Synopsis

```
void glp_error_hook(void (*func)(void *info), void *info);
```

Description

The routine `glp_error_hook` installs a user-defined hook routine to intercept abnormal termination.

The parameter `func` specifies the user-defined hook routine. It is called from the routine `glp_error` before the latter calls the abort function to abnormally terminate the application program because of fatal error. The parameter `info` is a transit pointer, specified in the corresponding call to the routine `glp_error_hook`; it may be used to pass some information to the hook routine.

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

If the hook routine returns, the application program is abnormally terminated. To prevent abnormal termnation the hook routine may perform a global jump using the standard function `longjmp`, in which case the application program *must* call the routine `glp_free_env`.

6.1.14 glp_alloc — allocate memory block

Synopsis

```
void *glp_alloc(int n, int size);
```

Description

The routine `glp_alloc` dynamically allocates a memory block of `n`×`size` bytes long. Note that:

1) the parameters `n` and `size` must be positive;

2) having been allocated the memory block contains arbitrary data, that is, it is *not* initialized by binary zeros;

3) if the block cannot be allocated due to insufficient memory, the routine prints an error message and abnormally terminates the program.

This routine is a replacement of the standard C function `malloc` and used by GLPK routines for dynamic memory allocation. The application program may use `glp_alloc` for the same purpose.

Returns

The routine `glp_alloc` returns a pointer to the memory block allocated. To free this block the routine `glp_free` (not the standard C function `free`!) should be used.

6.1.15 glp_realloc — reallocate memory block

Synopsis

```
void *glp_realloc(void *ptr, int n, int size);
```

Description

The routine `glp_realloc` dynamically reallocates a memory block pointed to by `ptr`, which was previously allocated by the routine `glp_alloc` or reallocated by this routine. Note that the pointer `ptr` must be valid and must not be `NULL`. The new size of the memory block is `n`×`size` bytes long. Note that:

1) both parameters `n` and `size` must be positive;

2) if the block cannot be reallocated due to insufficient memory, the routine prints an error message and abnormally terminates the program.

This routine is a replacement of the standard C function `realloc` and used by GLPK routines for dynamic memory allocation. The application program may use `glp_realloc` for the same purpose.

Returns

The routine `glp_realloc` returns a pointer to the memory block reallocated. To free this block the routine `glp_free` (not the standard C function `free`!) should be used.

6.1.16 glp_free — free memory block

Synopsis

```
void glp_free(void *ptr);
```

Description

The routine `glp_free` deallocates a memory block pointed to by `ptr`, which was previously allocated by the routine `glp_malloc` or reallocated by the routine `glp_realloc`. Note that the pointer `ptr` must be valid and must not be `NULL`.

This routine is a replacement of the standard C function `free` and used by GLPK routines for dynamic memory allocation. The application program may use `glp_free` for the same purpose.

6.1.17 glp_mem_usage — get memory usage information

Synopsis

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

Description

The routine `glp_mem_usage` reports some information about utilization of the memory by the routines `glp_malloc`, `glp_calloc`, and `glp_free`. 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 environment.

Example

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

6.1.18 glp_mem_limit — 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 (with the routines `glp_malloc` and `glp_calloc`) to `limit` megabytes.

Appendix A

Installing GLPK on Your Computer

A.1 Downloading the distribution tarball

The distribution tarball of the most recent version of the GLPK package can be found on http://ftp.gnu.org/gnu/glpk/ [via http] and ftp://ftp.gnu.org/gnu/glpk/ [via FTP]. It can also be found on one of the FTP mirrors; see http://www.gnu.org/prep/ftp.html. Please use a mirror if possible.

To make sure that the GLPK distribution tarball you have downloaded is intact you need to download the corresponding '`.sig`' file and run a command like this:

```
gpg --verify glpk-4.38.tar.gz.sig
```

If that command fails because you do not have the required public key, run the following command to import it:

```
gpg --keyserver keys.gnupg.net --recv-keys 5981E818
```

and then re-run the previous command.

A.2 Unpacking the distribution tarball

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 need to copy the GLPK distribution file to a working subdirectory and then unpack and unarchive the distribution file with the following command:

```
tar zx < glpk-X.Y.tar
```

A.3 Configuring the package

After unpacking and unarchiving the GLPK distribution you should configure the package, i.e. automatically tune it for your platform.

Normally, you should just `cd` to the subdirectory `glpk-X.Y` and run the configure script, e.g.

```
./configure
```

The '`configure`' shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a '`Makefile`' in each directory of the package. It also creates file '`config.h`' containing platform-dependent definitions. Finally, it creates a shell script '`config.status`' that you can run in the future to recreate the current configuration, a file '`config.cache`' that saves the results of its tests to speed up reconfiguring, and a file '`config.log`' containing compiler output (useful mainly for debugging '`configure`').

Running '`configure`' takes about a minute. 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`'.

By default both static and shared versions of the GLPK library will be compiled. Compilation of the shared librariy can be turned off by specifying the '`--disable-shared`' option to '`configure`':

```
./configure --disable-shared
```

If you encounter problems building the library try using the above option, because some platforms do not support shared libraries.

The GLPK package has some optional features listed below. By default all these features are disabled. To enable a feature the corresponding option should be passed to the configure script.

`--with-gmp` Enable using the GNU MP bignum library

This feature allows the exact simplex solver to use the GNU MP bignum library. If it is disabled, the exact simplex solver uses the GLPK bignum module, which provides the same functionality as GNU MP, however, it is much less efficient.

For details about the GNU MP bignum library see its web page at http://gmplib.org/.

`--enable-dl` The same as '`--enable-dl=ltdl`'

`--enable-dl=ltdl` Enable shared library support (GNU)

`--enable-dl=dlfcn` Enable shared library support (POSIX)

Currently this feature is only needed to provide dynamic linking to ODBC and MySQL shared libraries (see below).

For details about the GNU shared library support see the manual at http://www.gnu.org/software/libtool/manual/.

`--enable-odbc` Enable using ODBC table driver (`libiodbc`)

`--enable-odbc=unix` Enable using ODBC table driver (`libodbc`)

This feature allows transmitting data between MathProg model objects and relational databases accessed through ODBC.

For more details about this feature see the supplement "Using Data Tables in the GNU MathProg Modeling Language" (`doc/tables.pdf`).

`--enable-mysql` Enable using MySQL table driver (`libmysql`)

This feature allows transmitting data between MathProg model objects and MySQL relational databases.

For more details about this feature see the supplement "Using Data Tables in the GNU MathProg Modeling Language" (`doc/tables.pdf`).

A.4 Compiling the package

Normally, you can compile (build) the package by typing the command:

```
make
```

It reads '`Makefile`' generated by '`configure`' and performs all necessary jobs.

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.

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

```
CC=c89 CFLAGS=-O2 LIBS=-lposix ./configure
```

Or on systems that have the '`env`' program, you can do it like this:

```
env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./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:

`CC` C compiler program. The default is '`cc`'.

`INSTALL` Program used 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:

`DEFS` Configuration options, in the form '`-Dfoo -Dbar ...`'.

`LIBS` Libraries to link with, in the form '`-lfoo -lbar ...`'.

A.5 Checking the package

To check the package, i.e. to run some tests included in the package, you can use the following command:

```
make check
```

A.6 Installing the package

Normally, to install the GLPK package you should type the following command:

```
make install
```

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

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

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

The file '`configure.ac`' 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.7 Uninstalling the package

To uninstall the GLPK package, i.e. to remove all the package's files from the system places, you can use the following 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 **??**, page **??**).

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

On a particular data card some fields may be optional.

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

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

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

All numerical values in the fields 4 and 6 should be coded in the form $sxx\mathtt{E}syy$, 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.

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

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

MPS file should contain cards in the following order:

— NAME indicator card;

— ROWS indicator card;

— data cards specifying rows (constraints);

— COLUMNS indicator card;

— data cards specifying columns (structural variables) and constraint coefficients;

— RHS indicator card;

— data cards specifying right-hand sides of constraints;

— RANGES indicator card;

— data cards specifying ranges for double-bounded constraints;

— BOUNDS indicator card;

— data cards specifying types and bounds of structural variables;

— ENDATA indicator card.

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

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

B.2 Free MPS Format

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

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

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

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

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

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

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

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

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

B.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:

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

'`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 **??**, page **??**). 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.

[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.

B.7 RANGES section

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

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

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

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

The field 1 is not used and should be empty.

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

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

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;

[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.

`FX` — fixed variable (lower and upper bounds are equal);

`FR` — free variable (no bounds);

`MI` — no lower bound (lower bound is "minus infinity");

`PL` — no upper bound (upper bound is "plus infinity").

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

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

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

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

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

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

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

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

and bounds of (auxiliary and structural) variables

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

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

```
*0000000011111111112222222222333333333344444444445555555566
*2345678901234567890123456789012345678901234567890123456789
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 **??**, page **??**).

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 **??**, page **??**):

`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

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

and bound of variables

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

The corresponding MPS file may look like follows:

```
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
```

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 **??**, page **??**).

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, and blanks.

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.

[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.

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:

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

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 the following:

— 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:

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

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

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

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

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

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

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

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

Here is an example of the objective function definition:

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

C.3 Constraints section

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

The constraint section has the following form:

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

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

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

Constraint definitions have the following form:

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

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 the latter case the default names like '`r.nnn`' are assigned to unnamed constraints.

The linear form $s\ c\ x\ \ \ldots\ \ s\ c\ x$ 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 indicates 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. Another way is to introduce a slack double-bounded variable; for example, the constraint

$$10 \leq x + 2y + 3z \leq 50$$

can be written as follows:

$$x + 2y + 3z + t = 50,$$

where $0 \leq t \leq 40$ is a slack variable.

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$_p$

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

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

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

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

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

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

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:

$$\left\{\begin{matrix}\texttt{general}\\ \texttt{integer}\\ \texttt{binary}\end{matrix}\right\}$$
$$x_1$$
$$\dots$$
$$x_q$$

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

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

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

Here is an example of the integer section:

```
Integer
   z12
   z22
   z35
```

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

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 **??**, page **??**.

```
\* 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 to read LP/MIP problem data in any format supported by GLPK, solve the problem, and write its solution to an output text file.

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)
--lp              read LP/MIP problem in CPLEX LP format
--glp             read LP/MIP problem in GLPK 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 (same as --math)
-d filename, --data filename
                  read data section from filename (for --math only);
                  if model file also has data section, it is ignored
-y filename, --display filename
                  send display output to filename (for --math only);
                  by default the output is sent to terminal
--seed value      initialize pseudo-random number generator used in
                  MathProg model with specified seed (any integer);
                  if seed value is ?, some random seed will be used
--mincost         read min-cost flow problem in DIMACS format
--maxflow         read maximum flow problem in DIMACS format
--cnf             read CNF-SAT problem in DIMACS format
--simplex         use simplex method (default)
--interior        use interior point method (LP only)
-r filename, --read filename
                  read solution from filename rather to find it with
                  the solver
```

```
--min             minimization
--max             maximization
--scale           scale problem (default)
--noscale         do not scale problem
-o filename, --output filename
                  write solution to filename in printable format
-w filename, --write filename
                  write solution to filename in plain text format
--ranges filename
                  write sensitivity analysis report to filename in
                  printable format (simplex only)
--tmlim nnn       limit solution time to nnn seconds
--memlim nnn      limit available memory to nnn megabytes
--check           do not solve problem, check input data only
--name probname   change problem name to probname
--wmps filename   write problem to filename in fixed MPS format
--wfreemps filename
                  write problem to filename in free MPS format
--wlp filename    write problem to filename in CPLEX LP format
--wglp filename   write problem to filename in GLPK format
--wcnf filename   write problem to filename in DIMACS CNF-SAT format
--log filename    write copy of terminal output to filename
-h, --help        display this help information and exit
-v, --version     display program version and exit
```

LP basis factorization options

```
--luf             plain LU factorization (default)
--btf             block triangular LU factorization
--ft              Forrest-Tomlin update (requires --luf; default)
--cbg             Schur complement + Bartels-Golub update
--cgr             Schur complement + Givens rotation update
```

Options specific to the simplex solver

```
--primal          use primal simplex (default)
--dual            use dual simplex
--std             use standard initial basis of all slacks
--adv             use advanced initial basis (default)
--bib             use Bixby's initial basis
--ini filename    use as initial basis previously saved with -w
                  (disables LP presolver)
--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 presolver (default; assumes --scale and --adv)
--nopresol        do not use presolver
--exact           use simplex method based on exact arithmetic
--xcheck          check final basis using exact arithmetic
```

Options specific to the interior-point solver

```
--nord            use natural (original) ordering
--qmd             use quotient minimum degree ordering
--amd             use approximate minimum degree ordering (default)
--symamd          use approximate minimum degree ordering
```

Options specific to the MIP solver

```
--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
--mostf           branch on most fractional variable
--drtom           branch using heuristic by Driebeck and Tomlin
                  (default)
--pcost           branch using hybrid pseudocost heuristic (may be
                  useful for hard instances)
--dfs             backtrack using depth first search
--bfs             backtrack using breadth first search
--bestp           backtrack using the best projection heuristic
--bestb           backtrack using node with best local bound
                  (default)
--intopt          use MIP presolver (default)
--nointopt        do not use MIP presolver
--binarize        replace general integer variables by binary ones
                  (assumes --intopt)
--fpump           apply feasibility pump heuristic
--proxy [nnn]     apply proximity search heuristic (nnn is time limit
                  in seconds; default is 60)
--gomory          generate Gomory's mixed integer cuts
--mir             generate MIR (mixed integer rounding) cuts
--cover           generate mixed cover cuts
--clique          generate clique cuts
--cuts            generate all cuts above
--mipgap tol      set relative mip gap tolerance to tol
--minisat         translate integer feasibility problem to CNF-SAT
                  and solve it with MiniSat solver
--objbnd bound    add inequality obj <= bound (minimization) or
                  obj >= bound (maximization) to integer feasibility
                  problem (assumes --minisat)
```

For description of the MPS format see Appendix **??**, page **??**.

For description of the CPLEX LP format see Appendix **??**, page **??**.

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

For description of the DIMACS min-cost flow problem format and DIMACS maximum flow problem format see the document "GLPK: Graph and Network Routines" included in the GLPK distribution.

Appendix E

External Software Used In GLPK

In the GLPK package there are used some external software listed in this Appendix. Note that these software are *not* part of GLPK, but are used with GLPK and included in the distribution.

E.1 AMD

AMD Version 2.2, Copyright © 2007 by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. All Rights Reserved.

Description

AMD is a set of routines for pre-ordering sparse matrices prior to Cholesky or LU factorization, using the approximate minimum degree ordering algorithm.

License

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library 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 Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.

Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included.

AMD is available under alternate licences; contact T. Davis for details.

Availability

http://www.cise.ufl.edu/research/sparse/amd

E.2 COLAMD/SYMAMD

COLAMD/SYMAMD Version 2.7, Copyright © 1998-2007, Timothy A. Davis, All Rights Reserved.

Description

colamd: an approximate minimum degree column ordering algorithm, for LU factorization of symmetric or unsymmetric matrices, QR factorization, least squares, interior point methods for linear programming problems, and other related problems.

symamd: an approximate minimum degree ordering algorithm for Cholesky factorization of symmetric matrices.

Authors

The authors of the code itself are Stefan I. Larimore and Timothy A. Davis (davis at cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory.

License

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library 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 Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.

Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included.

COLAMD is also available under alternate licenses, contact T. Davis for details.

Availability

http://www.cise.ufl.edu/research/sparse/colamd

E.3 MiniSat

MiniSat-C v1.14.1, Copyright © 2005, Niklas Sorensson.

Description

MiniSat is a minimalistic implementation of a Chaff-like SAT solver based on the two-literal watch scheme for fast BCP and clause learning by conflict analysis.

License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Availability

http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat

E.4 zlib

zlib version 1.2.5, Copyright © 1995–2010 Jean-loup Gailly and Mark Adler.

Description

zlib is a general purpose data compression library. All the code is thread safe. The data format used by the zlib library is described by RFCs (Request for Comments) 1950 to 1952 in the files `rfc1950.txt` (zlib format), `rfc1951.txt` (deflate format) and `rfc1952.txt` (gzip format).

License

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly

Mark Adler

Availability

http://www.zlib.net/

GNU General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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>.