GLPK

GNU Linear Programming Kit
Version 1.1
User's Guide
November 2000

Andrew Makhorin

The GLPK package is a part of the GNU project, released under the aegis of GNU.

Copyright © 2000 Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia. All rights reserved.

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

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

1 Introduction 1

1.1 LP Problem 1
1.2 Brief Example 3
1.3 GLPK API Principles 6

2 GLPK API Routines 9

2.1 `glp_check_name` — check symbolic name for correctness . . . 9
2.2 `glp_create_item` — create new row or column 9
2.3 `glp_find_item` — find row or column by name 10
2.4 `glp_first_coef` — find the first constraint coefficient 11
2.5 `glp_first_item` — find the first row or column 12
2.6 `glp_get_activity` — get activity of the current row or column 12
2.7 `glp_get_bounds` — get bounds of the current row or column 13
2.8 `glp_get_coef` — get the current constraint coefficient 14
2.9 `glp_get_cpar` — get value of text control parameter 15
2.10 `glp_get_ipar` — get value of integer control parameter . . 15
2.11 `glp_get_name` — get name of the current row or column . . 15
2.12 `glp_get_rpar` — get value of real control parameter 16
2.13 `glp_get_seqn` — get '`seqn`' attribute of the current row or column 16
2.14 `glp_initialize` — initialize GLPK application program interface 17
2.15 `glp_new_coef` — create new constraint coefficient 17
2.16 `glp_next_coef` — find the next constraint coefficient 18
2.17 `glp_next_item` — find the next row or column 19
2.18 `glp_print_sol` — write problem solution using printble format 19
2.19 `glp_read_mps` — read problem data using MPS format . . 20
2.20 `glp_set_bounds` — set bounds of the current row or column 21
2.21 `glp_set_cpar` — set value of text control parameter 21
2.22 `glp_set_ipar` — set value of integer control parameter . . 22
2.23 `glp_set_name` — change name of the current row or column 22
2.24 `glp_set_rpar` — set value of real control parameter 23
2.25 `glp_set_seqn` — set '`seqn`' attribute of the current row or column 23
2.26 `glp_simplex` — solve problem by means of revised simplex method 24
2.27 `glp_terminate` — terminate GLPK application program interface 25

2.28 `glp_write_mps` — write problem data using MPS format 26

Appendix A Control Parameters 27

Appendix B MPS Format 31

Appendix C Output Format 39

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 library. It is intended for solving large scale linear programming problems by means of the revised simplex method.

1.1 LP Problem

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

minimize (or maximize)

$$Z = x_{obj} \tag{1.1}$$

subject to linear constraints

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

and bounds of variables

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

where $x_1, x_2, \ldots, x_m$ — rows or auxiliary variables; $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ — columns or structural variables; Z — objective function; obj — number of row that defines the objective function; $a_{11}, a_{12}, \ldots, a_{mn}$ — constraints coefficients; $l_1, l_2, \ldots, l_{m+n}$ — lower bounds of variables; $u_1, u_2, \ldots, u_{m+n}$ — upper bounds of variables.

Auxiliary variables are called *rows*, because they correspond to rows of the constraint matrix (i.e. matrix formed by constraint coefficients). Note that the value of an auxiliary variable is the value of the corresponding linear form. Analogously, structural variables are called *columns*, because they correspond to columns of the constraint matrix.

Bounds of variables can be finite as well as infinite. Besides, a lower bound can be equal to an upper bound. Thus, the following types of variables are possible:

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

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

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

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

b) are within their bounds (1.3), and

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

For solving LP problems GLPK uses well known numerical procedure called *the revised simplex method*. The simplex method performs iterations, where on each iteration it transforms the original system of linear constraints (1.3), resolving them through different sets of variables (the original system is resolved through auxiliary variables). Such transformed system is called *simplex table* and has the following form:

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

where $(x_B)_1, (x_B)_2, \ldots, (x_B)_m$ — basis variables; $(x_N)_1, (x_N)_2, \ldots, (x_N)_n$ — non-basis variables; $\alpha_{11}, \alpha_{12}, \ldots, \alpha_{mn}$ — coefficients of the simplex table.

From the linear programming theory it is known that if the optimal solution of LP problem exists, it can be represented in the form of the simplex table (1.4), where all non-basis variables are placed on their bounds, and values of basis variables are defined by equalities of the corresponding simplex table.

The set of values of all basis variables is called *basis solution*. If all basis variables are within their bounds, the solution is called *feasible basis solution*, otherwise the solution is called *infeasible basis solution*. The feasible basis solution, which provides the smallest (in case of minimization) or the largest (in case of maximization) value of the objective function, is called *optimal (basis) solution*. Thus, for solving LP problem the simplex method tries to find its optimal basis solution.

Feasibility of some basis solution may be determined by simple checking whether all basis variables are within their bounds. In order to determine that some basis solution is optimal, the following conditions should be checked. Since the auxiliary variable corresponding to the objective function is usually free (unbounded), it remains basis variable in all basis solutions (it can't become non-basis variable, because it has no bounds). Thus, the corresponding row of the simplex table may be written in the form:

$$Z = d_1(x_N)_1 + d_2(x_N)_2 + \ldots + d_n(x_N)_n, \tag{1.5}$$

where $Z = (x_B)_i$ is the auxiliary variable that defines the objective function; $d_1 = \alpha_{i1}, d_2 = \alpha_{i2}, \ldots, d_n = \alpha_{in}$ are coefficients of the ith row of the simplex table, which in this case are called *reduced costs* or *marginal values*. So, the solution is optimal, if the following *optimality conditions* are satisfied for all non-basis variables:

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

In other words, the solution is optimal, if there is no non-basis variables, 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. decreasing in case of minimization or increase in case of maximization) the objective function.

It may happen that some LP problem has no feasible solutions 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-basis variable can improve the objective function (i.e. optimality conditions (1.6) are violated) and at the same time this variable can be infinitely changed in its feasible direction not meeting any resistance from basis variables.

1.2 Brief Example

In order to understand what GLPK is from the user's point of view, let's consider the following simple LP problem:

minimize

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

subject to

$$x_1 + x_2 + x_3 \leq 100$$

$$10x_1 + 4x_2 + 5x_3 \leq 600$$

$$2x_1 + 2x_2 + 6_x3 \leq 300$$

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). It is easy to do that introducing auxiliary variables, by one for each original linear constraint. Thus, the considered LP problem can be rewritten as the following:

minimize

$$Z$$

subject to linear constraints

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

$$p = x_1 + x_2 + x_3$$

$$q = 10x_1 + 4x_2 + 5x_3$$

$$r = 2x_1 + 2x_2 + 6x_3$$

and bounds of variables

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

where Z, p, q, r are auxiliary variables; x_1, x_2, x_3 are structural (original) variables.

The C program shown below uses GLPK API routines in order to solve this example LP problem. (Note that this program is intended only for demonstrating some main principles of GLPK API.)

```
/* sample.c */

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

int main(void)
{     double Z, x1, x2, x3;

s1:   glp_initialize();

s2:   glp_set_cpar("problem", "SAMPLE");

s3:   glp_create_item(GLP_COL, "x1");
s4:   glp_create_item(GLP_COL, "x2");
s5:   glp_create_item(GLP_COL, "x3");

s6:   glp_create_item(GLP_ROW, "Z");
s7:   glp_set_bounds(GLP_ROW, GLP_FR, 0.0, 0.0);
s8:   glp_find_item(GLP_COL, "x1"), glp_new_coef(10.0);
s9:   glp_find_item(GLP_COL, "x2"), glp_new_coef(6.0);
s10:  glp_find_item(GLP_COL, "x3"), glp_new_coef(4.0);

s11:  glp_create_item(GLP_ROW, "p");
s12:  glp_set_bounds(GLP_ROW, GLP_UP, 0.0, 100.0);
s13:  glp_find_item(GLP_COL, "x1"), glp_new_coef(1.0);
s14:  glp_find_item(GLP_COL, "x2"), glp_new_coef(1.0);
s15:  glp_find_item(GLP_COL, "x3"), glp_new_coef(1.0);

s16:  glp_create_item(GLP_ROW, "q");
s17:  glp_set_bounds(GLP_ROW, GLP_UP, 0.0, 600.0);
s18:  glp_find_item(GLP_COL, "x1"), glp_new_coef(10.0);
s19:  glp_find_item(GLP_COL, "x2"), glp_new_coef(4.0);
s20:  glp_find_item(GLP_COL, "x3"), glp_new_coef(5.0);

s21:  glp_create_item(GLP_ROW, "r");
s22:  glp_set_bounds(GLP_ROW, GLP_UP, 0.0, 300.0);
```

```
s23:  glp_find_item(GLP_COL, "x1"), glp_new_coef(2.0);
s24:  glp_find_item(GLP_COL, "x2"), glp_new_coef(2.0);
s25:  glp_find_item(GLP_COL, "x3"), glp_new_coef(6.0);

s26:  glp_set_ipar("obj_dir", GLP_MAX);
s27:  glp_set_cpar("obj_row", "Z");

s28:  glp_simplex();

s29:  glp_find_item(GLP_ROW, "Z");
s30:  glp_get_activity(GLP_ROW, NULL, &Z, NULL);
s31:  glp_find_item(GLP_COL, "x1");
s32:  glp_get_activity(GLP_COL, NULL, &x1, NULL);
s34:  glp_find_item(GLP_COL, "x2");
s35:  glp_get_activity(GLP_COL, NULL, &x2, NULL);
s36:  glp_find_item(GLP_COL, "x3");
s37:  glp_get_activity(GLP_COL, NULL, &x3, NULL);

s38:  printf("Z = %g; x1 = %g; x2 = %g; x3 = %g", Z, x1, x2, x3);

s39:  glp_terminate();

      return 0;
}

/* eof */
```

The statement `s1` initializes GLPK Application Program Interface (GLPK API) calling the `glp_initialize` routine.

The statement `s2` identifies the problem. It calls the `glp_set_cpar` routine, which assigns the character string `"SAMPLE"` (name of the problem) to the text control parameter '`problem`' (name of the control parameter is specified by the first character string).

Statements `s3`, `s4`, and `s5` create columns named '`x1`', '`x2`', and '`x3`', which correspond to structural variables of the problem. Being created a column initially is non-negative, so in this case no additional actions are needed.

The statement `s6` creates row named '`Z`' (which correspond to the first auxiliary variable of the problem and plays the role of the objective function). Being created the row '`Z`' becomes the current row. The statement `s7` sets type and bounds of the current row (which is the row '`Z`'), where `GLP_FR` means that the current row is free (i.e. the corresponding auxiliary variable is unbounded). The statement `s8` finds the column named '`x1`', making it be the current column, and calls the `glp_new_coef` routine, which creates new constraint coefficient with given value 10.0 and assigns it to the current row and to the current column, i.e. in this case the coefficient 10.0 is placed in the row '`Z`' (because it is the current row) at the structural variable '`x1`' (because it is the current column). Statements `s9` and `s10` act in the same way and create the next two coefficients 6.0 and 4.0 in the same row '`Z`' (because it remains the current row) at the structural variables (columns) '`x2`' and '`x3`' respectively.

Note that the application program may introduce only non-zero coefficients, because if a coefficient is not defined, it is considered as zero.

Statements `s11`—`s15` create the row named '`p`' (the second auxiliary variable), set its type and upper bound (`GLP_UP` means that a row has upper bound, but no lower bound), and create constraint coefficients at the corresponding structural variables (columns).

Statements `s16`—`s20` perform similar actions for creating the row named '`q`' (the third auxiliary variable), and statements `s21`—`s25` do the same for creating the row named '`r`'.

The statement `s26` sets the optimization direction flag. It calls the `glp_set_ipar` routine, which assigns the flag `GLP_MAX` (meaning that the objective function should be maximized) to the integer control parameter named '`obj_dir`'. The statement `s27` defines the objective function, calling the `glp_set_cpar` routine, which assigns the character string `"Z"` (the name of row that is the objective function) to the text control parameter named '`obj_row`'.

At this point all LP problem data have been entered to the workspace, therefore the statement `s28` calls the routine `glp_simplex`, which performs searching for optimal solution of the problem by means of the revised simplex method.

The statement `s29` makes the row named '`Z`' be the current row, and the statement `s30` calls the `glp_get_activity` routine, which obtains the computed value of the current row (i.e. of the corresponding auxiliary variable) from the workspace and stores it to the location `Z`. Since the row named '`Z`' corresponds to the objective function, the location `Z` will contain the computed optimal (maximal) value of the objective function. Analogously, statements `s31`—`s37` obtain computed optimal values of columns (structural variables) '`x1`', '`x2`', and '`x3`', and store them to locations `x1`, `x2`, and `x3` respectively.

The statement `s38` prints the found optimal solution to the standard output. The corresponding printout looks like the following:

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

Finally the statement `s39` calls the `glp_terminate` routine, which terminates GLPK API and frees all memory allocated to the workspace.

1.3 GLPK API Principles

This section explains main GLPK API principles, which are important from the user's point of view.

For keeping all information related to a particular LP problem GLPK uses so called *workspace*. The user can consider the workspace as a database. For preparing problem data the application program calls GLPK API routines, which store these data to the workspace. The *solver*, which is a routine intended for solving LP problems, reads prepared problem data from the workspace, solves the problem, and writes computed solution and other relevant information back to the workspace. Finally, the application program again calls GLPK API routines in order to obtain the solution from the workspace for futher processing. Of course, other schemes are possible. For example, the application program may change some problem data and solve the problem repeatedly. Note that the current version of GLPK allows only one LP problem to be in the workspace.

The workspace is a storehouse, which keeps objects of the following kinds:

- rows (auxiliary variables);
- columns (structural variables);
- constraint coefficients;
- control parameters.

Each object of a particular kind has a set of attributes.

Rows and columns have the same set of attributes. These attributes are the following:

- row/column name (up to 255 arbitrary characters, except control and space characters);
- row/column type (for example, free variable, double-bounded variable, etc.);
- row/column lower and upper bounds;
- row/column constraint coefficient list (i.e. the list of coefficients placed in the corresponding row or column of the constraint matrix);
- row/column status (for example, basis variable, non-basis variable placed on its lower bound, etc.);
- row/column primal activity (i.e. the value of the corresponding auxiliary or structural variable computed by the solver);
- row/column dual activity (i.e. the reduced cost of the corresponding auxiliary or structural variable computed by the solver).

GLPK API routines provide so called *current row* and *current column*. The current row/column is just a row/column marked as current. This feature allows implicit referencing to rows and columns without using their symbolic names. For example, the `glp_set_bounds` routine changes type and bounds of the current row or column, therefore the application program needn't pass symbolic names of rows or columns to this routine. Note that the current row or/and the current column may be undefined.

Constraint coefficients have the following attributes:

- row, i.e. constraint, in which the coefficient is placed;
- column, i.e. structural variable, at which the coefficient is placed;
- numerical value of the coefficient.

GLPK API routines also provide so called *current coefficient* similarly to rows and columns, i.e. a coefficient may be marked as currentin order to allow implicit referencing to the coefficient without mentioning the corresponding row and column.

Control parameters are intended for several purposes. Some of them just represent additional information about the problem (for example, name of the problem). Others can affect on the behavior of GLPK API routines. GLPK has a fixed set of control parameters, i.e. the user is not able to introduce its own control parameters.

Control parameters can be of three different types: of integer type, of real (floating point) type, and of text (character string) type. Each control parameter has the following two attributes:

- parameter name;
- parameter value.

All GLPK routines are divided into two categories: API routines and internal routines.

API routines implement so called *application program interface (API)*. These routines are intended for direct using by the application program. They are described in the next chapter of this document.

Internal routines are not intended (at least in this version of GLPK) for direct using by the application program. They are called indirectly by API routines.

2 GLPK API Routines

This chapter describes all GLPK API routines in alphabetical order.

Description of some routines include *time complexity*. This characteristic shows how the time of an operation depends on the size of particular data in the workspace. For example, `glp_create_item` routine, which creates new row or column, has time complexity $O(\log n)$, where n is current number of rows or columns in the workspace. This means that if the operation takes t time units in case of n rows, it will take $10t$ time units in case of $1000n$ rows, i.e. increasing number of rows in 1000 times involves increasing the time of operation approx. in 10 times, because $\log 1000 \approx 10$ (of course, the base of logarithm is not essential). The time complexity $O(1)$ means that an operation takes the constant time, i.e. the time of operation doesn't depend on the current size of data in the workspace. Knowing time complexity of GLPK routines the user can estimate this characteristic for its own routines, which are based on GLPK routines.

2.1 `glp_check_name` — check symbolic name for correctness

Synopsis

```
#include "glpk.h"
int glp_check_name(char *name);
```

Description

The `glp_check_name` routine checks the given symbolic `name` for correctness.

The symbolic name is considered as correct if it contains from 1 up to 255 arbitrary characters, except control and space characters.

Returns

0 the given symbolic `name` is valid;

1 the character string `name` has invalid length;

2 the character string `name` contains control or/and space characters.

2.2 `glp_create_item` — create new row or column

Synopsis

```
#include "glpk.h"
int glp_create_item(int what, char *name);
```

Description

If `what` is `GLP_ROW`, the `glp_create_item` routine creates new row with the given `name`. Initially the created row has `GLP_FX` type (i.e. the auxiliary variable is fixed) with both lower and upper bounds equal to zero (such row corresponds to linear equality constraint with zero right hand side). Being created new row becomes the current row.

If `what` is `GLP_COL`, the `glp_create_item` routine creates new column with the given `name` (i.e. the structural variable has lower bound, but no upper bound). Initially the created column has `GLP_LO` type with lower bound equal to zero (such column corresponds to non-negative variable). Being created new column becomes the current column.

Complexity

This operation has time complexity $O(\log n)$, where n is current number of rows or columns in the workspace.

Returns

0	no errors;
1	row/column with the given `name` already exists;
2	the parameter `what` or the given `name` is invalid.

2.3 `glp_find_item` — find row or column by name

Synopsis

```
#include "glpk.h"
int glp_find_item(int what, char *name);
```

Description

If `what` is `GLP_ROW`, the `glp_find_item` routine finds row with the given `name` and makes it be the current row.

If `what` is `GLP_COL`, the `glp_find_item` routine finds column with the given `name` and makes it be the current column.

Complexity

This operation has time complexity $O(\log n)$, where n is current number of rows or columns in the workspace.

Returns

0	no errors;
1	row/column with the given `name` doesn't exist;
2	the parameter `what` or the given `name` is invalid.

2.4 glp_first_coef — find the first constraint coefficient

Synopsis

```
#include "glpk.h"
int glp_first_coef(int what);
```

Description

If `what` is `GLP_ROW`, the `glp_first_coef` routine finds the first constraint coefficient placed in the current row and makes it be the current coefficient. At the same time the routine also makes the column, which corresponds to the found coefficient, be the current column (the current row remains unchanged).

If `what` is `GLP_COL`, the `glp_first_coef` routine finds the first constraint coefficient placed in the current column and makes it be the current coefficient. At the same time the routine also makes the row, which corresponds to the found coefficient, be the current row (the current column remains unchanged).

The order of constraint coefficients can be changed by GLPK API routines, therefore the application program should not rely on some particular ordering (except cases, when particular ordering is explicitly declared).

In order to enumerate all constraint coefficients of the problem in row wise manner the following fragment may be used:

```
for (r = glp_first_item(GLP_ROW); r == 0; r = glp_next_item(GLP_ROW))
for (s = glp_first_coef(GLP_ROW); s == 0; s = glp_next_coef(GLP_ROW))
{
   /* processing the current coefficient */
}
```

Analogously, in order to enumerate all constraint coefficients in column wise manner the following fragment may be used:

```
for (r = glp_first_item(GLP_COL); r == 0; r = glp_next_item(GLP_COL))
for (s = glp_first_coef(GLP_COL); s == 0; s = glp_next_coef(GLP_COL))
{
   /* processing the current coefficient */
}
```

Complexity

This operation has time complexity $O(1)$.

Returns

0 no errors;

1 the current row is undefined or empty (in case of `GLP_ROW`), or the current column is undefined or empty (in case of `GLP_COL`);

2 the parameter `what` is invalid.

2.5 glp_first_item — find the first row or column

Synopsis

```
#include "glpk.h"
int glp_first_item(int what);
```

Description

If `what` is `GLP_ROW`, the `glp_first_item` routine finds the first row of the problem and makes it be the current row.

If `what` is `GLP_COL`, the `glp_first_item` routine finds the first column of the problem and makes it be the current column.

It is assumed that one row (column) precedes other row (column) if the former was created before the later. Therefore the first row (column) is that one, which was created before any other rows (columns).

In order to enumerate all rows of the problem the following fragment may be used:

```
for (r = glp_first_item(GLP_ROW); r == 0; r = glp_next_item(GLP_ROW))
{
   /* processing the current row */
}
```

All columns may be enumerated in the same way.

Complexity

This operation has time complexity $O(1)$.

Returns

0 no errors;

1 there are no rows or columns in the problem;

2 the parameter `what` is invalid.

2.6 glp_get_activity — get activity of the current row or column

Synopsis

```
#include "glpk.h"
int glp_get_activity(int what, int *tagx, double *valx, double *dx);
```

Description

If `what` is `GLP_ROW`, the `glp_get_activity` routine stores status, primal activity, and dual activity of the current row (i.e. of the corresponding auxiliary variable) to locations, which parameters `tagx`, `valx`, and `dx` point to, respectively.

If `what` is `GLP_COL`, the `glp_get_activity` routine stores the same information for the current column (i.e. for the corresponding structural variable).

If some pointers `tagx`, `valx`, or `dx` are `NULL`, the corresponding values are not stored.

Information reported by the `glp_get_activity` routine is determined by the final simplex table (see [LP Problem], page 1) found by the solver. This information has the meaning only if the problem has been successfully solved.

The status of the variable may be one of the following:

`GLP_BS` variable is basis;

`GLP_NL` variable is non-basis and placed on its lower bound (only for variables of `GLP_LO` and `GLP_DB` types);

`GLP_NU` variable is non-basis and placed on its upper bound (only for variables of `GLP_UP` and `GLP_DB` types);

`GLP_NF` free variable is non-basis (only for variables of `GLP_FR` type);

`GLP_NS` fixed variable is non-basis (only for variables of `GLP_FX` type).

The primal activity is the computed value of variable. (Note that the value of free non-basis variable is always equal to zero.)

The dual activity is the computed reduced cost (marginal value) of variable. The signs of dual activities are always determined by the formula (1.5) (see [LP Problem], page 1) in case of minimization as well in case of maximization.

Returns

0 no errors;

1 the current row/column is undefined;

2 the parameter `what` is invalid.

2.7 glp_get_bounds — get bounds of the current row or column

Synopsis

```
#include "glpk.h"
int glp_get_bounds(int what, int *type, double *lb, double *ub);
```

Description

If `what` is `GLP_ROW`, the `glp_get_bounds` routine stores type, lower bound, and upper bound of the current row (i.e. of the corresponding auxiliary variable) to locations, which parameters `type`, `lb`, and `ub` point to, respectively.

If `what` is `GLP_COL`, the `glp_get_bounds` routine stores the same information for the current column (i.e. for the corresponding structural variable).

If some pointers `type`, `lb`, or `ub` are `NULL`, the corresponding values are not stored.

Types and bounds reported by the routine are the following:

Type	Bounds	Description
`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 auxiliary (in case of `GLP_ROW`) or structural (in case of `GLP_COL`) variable.

If the current row/column has no lower or/and upper bound, the corresponding value of `lb` or/and `ub` is set to zero. If the current row/column is of `GLP_FX` types, both values of `lb` and `ub` are set to the fixed value of variable.

Returns

0 no errors;

1 the current row/column is undefined;

2 the parameter `what` is invalid.

2.8 glp_get_coef — get the current constraint coefficient

Synopsis

```
#include "glpk.h"
int glp_get_coef(double *val);
```

Description

The `glp_get_coef` routine stores value of the current constraint coefficient to the location, which the parameter `val` points to.

Returns

0 no errors;

1 the current coefficient is undefined;

2.9 glp_get_cpar — get value of text control parameter

Synopsis

```
#include "glpk.h"
int glp_get_cpar(char *name, char *val);
```

Description

The `glp_get_cpar` routine copies value of the text control parameter with the given `name` to the character string `val`.

See Appendix A for description of all control parameters.

Returns

0 no errors;

1 the given `name` is invalid.

2.10 glp_get_ipar — get value of integer control parameter

Synopsis

```
#include "glpk.h"
int glp_get_ipar(char *name, int *val);
```

Description

The `glp_get_ipar` routine stores value of the integer control parameter with the given `name` to the location, which the parameter `val` points to.

See Appendix A for description of all control parameters.

Returns

0 no errors;

1 the given `name` is invalid.

2.11 glp_get_name — get name of the current row or column

Synopsis

```
#include "glpk.h"
int glp_get_name(int what, char *name);
```

Description

If `what` is `GLP_ROW`, the `glp_get_name` routine copies symbolic name of the current row to the character string `name`.

If `what` is `GLP_COL`, the `glp_get_name` routine copies symbolic name of the current column to the character string `name`.

Maximal length of symbolic names is 255 characters, therefore the array `name` should contain at least 256 locations.

Returns

0 no errors;

1 the current row/column is undefined;

2 the parameter `what` is invalid.

2.12 glp_get_rpar — get value of real control parameter

Synopsis

```
#include "glpk.h"
int glp_get_rpar(char *name, double *val);
```

Description

The `glp_get_rpar` routine stores value of the real control parameter with the given `name` to the location, which the parameter `val` points to.

See Appendix A for description of all control parameters.

Returns

0 no errors;

1 the given `name` is invalid.

2.13 glp_get_seqn — get 'seqn' attribute of the current row or column

Synopsis

```
#include "glpk.h"
int glp_get_seqn(int what, int *seqn);
```

Description

If `what` is `GLP_ROW`, the `glp_get_seqn` routine stores the '`seqn`' attribute of the current row to the location, which the parameter `seqn` points to.

If `what` is `GLP_COL`, the `glp_get_seqn` routine stores the '`seqn`' attribute of the current column to the location, which the parameter `seqn` points to.

The '`seqn`' attribute is an additional attribute of rows and columns. It is just an integer value, which may be used for any purposes. This attribute is intended mainly for needs of GLPK API routines, however, it may be used by the application program.

Returns

0 no errors;

1 the current row/column is undefined;

2 the parameter `what` is invalid.

2.14 `glp_initialize` — initialize GLPK application program interface

Synopsis

```
#include "glpk.h"
int glp_initialize(void);
```

Description

The `glp_initialize` routine initializes GLPK application program interface (API).

This routine should be called once before calls any other GLPK API routines.

Returns

0 no errors;

1 GLPK API already initialized.

2.15 `glp_new_coef` — create new constraint coefficient

Synopsis

```
#include "glpk.h"
int glp_new_coef(double val);
```

Description

The `glp_new_coef` routine creates new constraint coefficient, which is placed in the current row and in the current column and has the value `val`. Being created new coefficient becomes the current coefficient.

The application program needn't specify zero coefficients, because if a coefficient is not specified, it is considered as zero.

Complexity

This operation has time complexity $O(1)$.

Returns

0	no errors;
1	the current row or/and column is undefined.

2.16 `glp_next_coef` — find the next constraint coefficient

Synopsis

```
#include "glpk.h"
int glp_next_coef(int what);
```

Description

If `what` is `GLP_ROW`, the `glp_next_coef` routine finds the next constraint coefficient placed in the same row as the current coefficient and makes it be the current coefficient. At the same time the routine also makes the column, which corresponds to the found coefficient, be the current column (the current row remains unchanged even if it doesn't correspond to the found coefficient).

If `what` is `GLP_COL`, the `glp_next_coef` routine finds the next constraint coefficient placed in the same column as the current coefficient and makes it be the current coefficient. At the same time the routine also makes the row, which corresponds to the found coefficient, be the current row (the current column remains unchanged even if it doesn't correspond to the found coefficient).

The order of constraint coefficients can be changed by GLPK API routines, therefore the application program should not rely on some particular ordering (except cases, when particular ordering is explicitly declared).

As a rule the `glp_next_coef` routine is used in conjunction with the `glp_first_coef` routine in order to enumerate all constraint coefficients of the problem in row wise or column wise manner.

Complexity

This operation has time complexity $O(1)$.

Returns

0	no errors;
1	the current coefficient is undefined or there is no the next coefficient in the same row/column;
2	the parameter `what` is invalid.

2.17 `glp_next_item` — find the next row or column

Synopsis

```
#include "glpk.h"
int glp_next_item(int what);
```

Description

If `what` is `GLP_ROW`, the `glp_next_item` routine finds the next row of the problem, which follows the current row, and makes it be the current row.

If `what` is `GLP_COL`, the `glp_next_item` routine finds the next column of the problem, which follows the current column, and makes it be the current column.

It is assumed that one row (column) precedes other row (column) if the former was created before the later. Therefore the next row (column) is that one, immediately after the current row (column).

As a rule the `glp_next_item` routine is used in conjunction with the `glp_first_item` routine in order to enumerate all rows or columns of the problem.

Complexity

This operation has time complexity $O(1)$.

Returns

0	no errors;
1	the current row/column is undefined or there is no the next row/column in the problem;
2	the parameter `what` is invalid.

2.18 `glp_print_sol` — write problem solution using printble format

Synopsis

```
#include "glpk.h"
int glp_print_sol(char *fname);
```

Description

The `glp_print_sol` routine writes the problem solution using printable format to the text file, whose name is the character string `fname`.

Information reported by the routine corresponds to the final simplex table (see [LP Problem], page 1) found by the solver. This information is intended mainly for visual analyzing and has the meaning only if the problem has been successfully solved.

The output format used by the `glp_print_sol` routine is described in Appendix C.

Returns

0 no errors;

1 the operation failed because of errors. All diagnostic sent to `stderr`.

2.19 `glp_read_mps` — read problem data using MPS format

Synopsis

```
#include "glpk.h"
int glp_read_mps(char *fname);
```

Description

The `glp_read_mps` routine reads LP problem data using MPS format from the input text file, whose name is the character string `fname`.

The MPS format is described in Appendix B.

As a rule the workspace should be empty before reading LP data, i.e. it should contain no rows and columns.

Control parameters

The behavior of the `glp_read_mps` routine depends on the following control parameters (detailed description of these control parameters can be found in Appendix A):

`mps_obj_name`
: the name of row that specifies the objective function;

`mps_rhs_name`
: the name of right-hand side vector;

`mps_rng_name`
: the name of range vector;

`mps_bnd_name`
: the name of bound vector.

Returns

0 no errors;

1 the operation failed because of errors. All diagnostic sent to `stderr`.

2.20 glp_set_bounds — set bounds of the current row or column

Synopsis

```
#include "glpk.h"
int glp_set_bounds(int what, int type, double lb, double ub);
```

Description

If `what` is `GLP_ROW`, the `glp_set_bounds` routine changes type and bounds of the current row.

If `what` is `GLP_COL`, the `glp_set_bounds` routine changes type and bounds of the current column.

Parameters `type`, `lb`, and `ub` specify respectively type, lower bound and upper bound, which should be set for the current row or column:

Type	Bounds	Description
`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 auxiliary (in case of `GLP_ROW`) or structural (in case of `GLP_COL`) variable.

If the current row/column has no lower or/and upper bound, the corresponding values of `lb` or/and `ub` are ignored. If the current row/column is of `GLP_FX` `type`, the fixed value should be specified by the parameter `lb`, and the parameter `ub` is ignored.

Returns

0 no errors;

1 the current row/column is undefined;

2 the parameter `what` is invalid, or the parameter `type` is invalid, or $lb \ge ub$ (in case of `type` = `GLP_DB` only).

2.21 glp_set_cpar — set value of text control parameter

Synopsis

```
#include "glpk.h"
int glp_set_cpar(char *name, char *val);
```

Description

The `glp_set_cpar` routine assigns the character string `val` to the text control parameter with the given `name`.

See Appendix A for description of all control parameters.

Returns

0 no errors;

1 the given `name` is invalid;

2 the given value (character string `val`) is invalid.

2.22 `glp_set_ipar` — set value of integer control parameter

Synopsis

```
#include "glpk.h"
int glp_set_ipar(char *name, int val);
```

Description

The `glp_set_ipar` routine assigns the given value `val` to the integer control parameter with the given `name`.

See Appendix A for description of all control parameters.

Returns

0 no errors;

1 the given `name` is invalid;

2 the given value (integer value `val`) is invalid.

2.23 `glp_set_name` — change name of the current row or column

Synopsis

```
#include "glpk.h"
int glp_set_name(int what, char *name);
```

Description

If `what` is `GLP_ROW`, the `glp_set_name` routine assigns the new `name` to the current row.

If `what` is `GLP_COL`, the `glp_set_name` routine assigns the new `name` to the current column.

Complexity

This operation has time complexity $O(\log n)$, where n is current number of rows or columns in the workspace.

Returns

0 no errors;

1 the current row/column is undefined or row/column with the given `name` already exists;

2 the parameter `what` or the given `name` is invalid.

2.24 glp_set_rpar — set value of real control parameter

Synopsis

```
#include "glpk.h"
int glp_set_rpar(char *name, double val);
```

Description

The `glp_set_rpar` routine assigns the given value `val` to the real control parameter with the given `name`.

See Appendix A for description of all control parameters.

Returns

0 no errors;

1 the given `name` is invalid;

2 the given value (floating point value `val`) is invalid.

2.25 glp_set_seqn — set 'seqn' attribute of the current row or column

Synopsis

```
#include "glpk.h"
int glp_set_seqn(int what, int seqn);
```

Description

If `what` is `GLP_ROW`, the `glp_set_seqn` routine assigns the given value `seqn` to the '`seqn`' attribute of the current row.

If `what` is `GLP_COL`, the `glp_set_seqn` routine assigns the given value `seqn` to the '`seqn`' attribute of the current column.

The '`seqn`' attribute is an additional attribute of rows and columns. It is just an integer value, which may be used for any purposes. This attribute is intended mainly for needs of GLPK routines, however, it may be used by the application program.

Returns

0	no errors;
1	the current row/column is undefined;
2	the parameter `what` is invalid.

2.26 `glp_simplex` — solve problem by means of revised simplex method

Synopsis

```
#include "glpk.h"
int glp_simplex(void);
```

Description

The `glp_simplex` routine is an LP problem solver based on the two-phase revised simplex method.

This routine obtains problem data from the workspace, solves the problem, and stores the computed solution and other relevant information back to the workspace in order that the application program could use this information for further processing.

Generally, the `glp_simplex` routine performs the following actions:

preparing internal data structures;

searching for feasible basis solution (phase I);

searching for optimal basis solution (phase II);

storing the computed solution to the workspace.

Since large scale problems may take a long time, the `glp_simplex` routine reports some information about the current status of the searching. This information is sent to stdout approximately once per second and has the following format:

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

where '`nnn`' is an iteration count, '`xxx`' is the current value of the objective function (which is unscaled and has correct sign), '`yyy`' is the current value of sum of infeasibilities (which

is scaled, therefore may be used only for visual estimating), 'ddd' is the current number of basis fixed variables. If the asterisk '*' precedes to 'nnn', the solver is searching for optimal solution (i.e. the feasible solution has been found yet), otherwise the solver is searching for some feasible solution.

Note that the `glp_simplex` solver is not perfect. Although it has been successfully tested on a wide set of LP problems, there are hard problems[1], which can't be solved by this solver.

Control parameters

The behavior of the `glp_simplex` routine depends on a number of control parameters, which are described in Appendix A.

If the problem is not very hard, default values of control parameters are fit for most cases, so the user needn't take care of them.

Returns

0 no errors. This code means that the solver has successfully finished solving the problem (note that, for example, if the problem has no feasible solution, the code returned by the solver will be equal to zero);

1 it is not possible to start solving the problem because of incorrect data. All diagnostics sent to `stderr`;

2 the solver is not able to solve the problem. All diagnostics sent to `stderr`.

2.27 `glp_terminate` — terminate GLPK application program interface

Synopsis

```
#include "glpk.h"
int glp_terminate(void);
```

Description

The `glp_terminate` routine terminates the GLPK application program interface (API) and frees all the memory allocated to the workspace. All data in the workspace are destroyed.

As a rule this routine is called before termination of the application program. However, it may be used also for cleaning the workspace — in this case the application program should again call the `glp_initialize` routine for re-initializing API.

[1] Hard problem is not the same as large scale problem. Large scale problem is just a problem, which has many (for example, thousands or seceral tens of thousands) rows and columns. Many large scale problems can be successfully solved by the `glp_simplex` solver, if the computer has enough memory and appropriate processing unit. Hard problem is a "pathological" problem, which has some unusual properties making it very hard to be solved by the simplex method.

Returns

0 no errors;

1 GLPK API already terminated.

2.28 glp_write_mps — write problem data using MPS format

Synopsis

```
#include "glpk.h"
int glp_write_mps(char *fname);
```

Description

The `glp_write_routine` writes LP problem data using MPS format from the workspace to the output text file, whose name is the character string `fname`.

The MPS format is described in Appendix B.

Note that the MPS format allows row and column names up to 8 characters. Therefore, if names of some rows or columns presented in the workspace contain more that 8 characters, the routine will reduce such names. However, the routine make all reduced names be unique.

Control parameters

The behavior of the `glp_write_mps` routine depends on the following control parameters (detailed description of these control parameters can be found in Appendix A):

`mps_use_names`
reduce long names using original names;

`mps_one_entry`
don't use fields 5 and 6;

`mps_pedantic`
always put column and vector names to the field 2;

`mps_skip_empty`
skip empty columns.

Returns

0 no errors;

1 the operation failed because of errors. All diagnostic sent to `stderr`.

Appendix A Control Parameters

This appendix describes control parameters. Some of these parameters just report an additional information about the problem, others allow changing the behavior of some GLPK API routines.

`mps_bnd_name` (type: C, getable: yes, setable: yes, default: empty string)
: Affects on the `glp_read_mps` routine. It specifies the name of bound vector that should be used in the problem. If this parameter is an empty string, the first bound vector is used.

`mps_obj_name` (type: C, getable: yes, setable: yes, default: empty string)
: Affects on the `glp_read_mps` routine. It specifies the name of row that should be used as the objective function. If this parameter is an empty string, the first row of '`N`' type is used as the objective function.

`mps_one_entry` (type: I, getable: yes, setable: yes, default: `GLP_NO`)
: Affects on `glp_write_mps` routine. If this parameter is `GLP_YES`, the routine keeps fields 5 and 6 empty. Otherwise the routine uses all fields.

`mps_pedantic` (type: I, getable: yes, setable: yes, default: `GLP_NO`)
: Affects on `glp_write_mps` routine. If this parameter is `GLP_YES`, the routine never omits column and vector names. Otherwise the routine omits those names every time when it is possible.

`mps_rhs_name` (type: C, getable: yes, setable: yes, default: empty string)
: Affects on the `glp_read_mps` routine. It specifies the name of right hand side (RHS) vector that should be used in the problem. If this parameter is an empty string, the first RHS vector is used.

`mps_rng_name` (type: C, getable: yes, setable: yes, default: empty string)
: Affects on the `glp_read_mps` routine. It specifies the name of range vector that should be used in the problem. If this parameter is an empty string, the first range vector is used.

`mps_skip_empty` (type: I, getable: yes, setable: yes, default: `GLP_NO`)
: Affects on the `glp_write_mps` routine. If this parameter is `GLP_YES`, the routine writes all columns, including empty ones (i.e. those, which have no constraint coefficients). Otherwise the routine writes only non-empty columns.

`mps_use_names` (type: I, getable: yes, setable: yes, default: `GLP_YES`)
: Affects on the `glp_write_mps` routine. If this parameter is `GLP_YES`, the routine tries to generate standard MPS names using original row and column names as templates. Otherwise the routine generates plain names using sequential numbers of rows and columns.

`nc` (type: I, getable: yes, setable: no)
: Reports total number of columns (structural variables) in the problem.

`nr` (type: I, getable: yes, setable: no)
: Reports total number of rows (auxiliary variables) in the problem.

`nz` (type: I, getable: yes, setable: no)
: Reports total number of constraint coefficients in the problem (i.e. total number of explicitly defined elements in the constraint matrix).

`obj_dir` (type: I, getable: yes, setable: yes, default: `GLP_MIN`)
: Is the optimization flag, used by the solver. The following values can be assigned to this parameter:

  `GLP_INI` only the initial solution should by found;

  `GLP_ANY` only any feasible solution should be found;

  `GLP_MIN` the objective function should be minimized;

  `GLP_MAX` the objective function should be maximized.

`obj_row` (type: C, getable: yes, setable: yes, default: empty string)
: Specifies the name of row (i.e. the corresponding auxiliary variable), which defines the objective function. If this parameter is an empty string, the objective function is considered as identically equal to zero.

`problem` (type: C, getable: yes, setable: yes, default: `"UNKNOWN"`)
: Specifies the name of problem. This name is used for informative purposes and should obey the same rules as names of rows and columns.

`round` (type: I, getable: yes, setable: yes, default: `GLP_YES`)
: If this parameter is `GLP_YES`, the solver replaces values of those basis variables, which are near to zero, by exact zeros (the rounding is performed after the final basis solution has been computed). If this parameter is `GLP_NO`, the solver remains values of all basis variables "as is".

`scale` (type: I, getable: yes, setable: yes, default: `GLP_YES`)
: If this parameter is `GLP_YES`, the solver performs automatic scaling the problem before solving (the scaling doesn't change any information in the workspace, so this feature is transparent for the user). Otherwise, if this parameter is `GLP_NO`, the scaling is not used.

`spx_form` (type: I, getable: yes, setable: yes, default: `GLP_RFI_BG`)
: Specifies what form of the basis matrix should be used by the solver:

  `GLP_EFI` EFI (Elimination Form of the Inverse);

  `GLP_RFI_BG`
  : RFI (Reid's Form of the Inverse) with updating technique proposed by Bartels and Golub;

  `GLP_RFI_FT`
  : RFI (Reid's Form of the Inverse) with updating technique proposed by Forrest and Tomlin.

`spx_relax` (type: I, getable: yes, setable: yes, default: `GLP_YES`)
: If this parameter is `GLP_YES`, the solver uses the two-pass ratio test (for both primal and dual simplex) proposed by P.Harris, which allows to prevent numerical instability in case of degenerative problems. If this parameter is GLP_NO, the solver uses the standard "textbook" ratio test.

`spx_steep` (type: I, getable: yes, setable: yes, default: `GLP_YES`)
: If this parameter is `GLP_YES`, the solver uses the steepest edge technique (for both primal and dual simplex) proposed by Goldfarb and Reid, which allows great improving the convergence in most cases. If this parameter is `GLP_NO`, the solver uses the standard "textbook" pricing.

`spx_use_dual` (type: I, getable: yes, setable: yes, default: `GLP_NO`)
: If this parameter is GLP_YES, the solver uses the dual simplex method for searching for feasible solution. Otherwise, the solver uses the primal simplex method.

`status` (type: I, getable: yes, setable: yes, initial: `GLP_UNDEF`)
: This parameter is set by the solver and reports the status of the found basis solution:

  `GLP_UNDEF`
  : the solution status is undefined (this case means that the problem has not been solved yet);

  `GLP_OPT` the solution is optimal;

  `GLP_FEAS` the solution is feasible;

  `GLP_INFEAS`
  : the solution is infeasible;

  `GLP_NOFEAS`
  : the problem has no feasible solution;

  `GLP_UNBND`
  : the problem has unbounded solution.

`sum_aij` (type: I, getable: yes, setable: yes, default: `GLP_NO`)
: If this parameter is `GLP_YES`, the solver replaces multiplets (i.e. elements of the constraint matrix, which have identical row and column indices) by their sum before solving the problem (this operation doesn't change any information in the workspace). Otherwise the solver considers multiplets as incorrect data.

`tol_aij` (type: R, getable: yes, setable: yes, default: `10e-15`)
: Specifies the absolute tolerance, which is used by the solver to drop small elements of the original constraint matrix before solving the problem (this operation doesn't change any information in the workspace). If this tolerance is equal to zero, the solver uses the original matrix without any changes.

`tol_bnd` (type: R, getable: yes, setable: yes, default: `10e-8`)
: Specifies the relative tolerance, which is used by the solver for checking primal feasibility of the current basis solution.

  It is not recommended to change this parameter without detailed understanding its purpose.

`tol_dj` (type: R, getable: yes, setable: yes, default: `10e-7`)
: Specifies the relative tolerance, which is used by the solver for checking dual feasibility of the current basis solution.

It is not recommended to change this parameter without detailed understanding its purpose.

`tol_piv` (type: R, getable: yes, setable: yes, default: `10e-10`)
Specifies the relative tolerance, which is used by the solver for pivoting.

It is not recommended to change this parameter without detailed understanding its purpose.

Appendix B MPS Format

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

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

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

Data cards contain LP 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 for identifying LP rows, LP columns, and some vectors (see below).

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

All names specified in fields 2, 3, and 5 should contain from 1 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, these spaces are ignored.

All numerical values placed in fields 4 and 6 should be coded in the form *sxx*`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, these 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 the card are considered as comment and ignored.

MPS file should contain cards in the following order:

- the NAME indicator card;
- the ROWS indicator card;
- data cards specifying LP rows (constraints or auxiliary variables);
- the COLUMNS indicator card;
- data cards specifying LP columns (structural variables) and constraint coefficients;
- the RHS indicator card;

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

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

- data cards specifying right-hand sides of constraints;
- the RANGES indicator card;
- data cards specifying double-sided constraints;
- the BOUNDS indicator card;
- data cards specifying types and bounds of LP columns;
- the ENDATA indicator card.

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

Sections RHS, RANGES, and BOUNDS are optional and may be omitted.

NAME indicator card

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

ROWS section

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

Each data card of the ROWS section specifies one row (constraint) of the LP problem. These data cards have the following format.

The filed 1 specifies the row type (see (1.2), [LP Problem], page 1):

'`N`' in the filed 1 means that the row is free (unbounded), i.e. the corresponding constraint has the form:

$$-\infty < x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} < +\infty;$$

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

$$-\infty < x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} \leq b_i;$$

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

$$b_i \leq x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} < +\infty;$$

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

$$x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} = b_i.$$

where b_i is the right hand side. Note that each constraint is connected with the corresponding auxiliary variable (x_i above), therefore bounds of the linear form can be understood as bounds of such auxiliary variable.

The field 2 specifies the row name (i.e. the name of the corresponding auxiliary variable).

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. All double-sided constraints should be specified in the RANGES section.

COLUMNS section

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

Each data card of the COLUMNS section specifies one or two constraint coefficients and introduces names of LP columns, i.e. names of structural variables. These data cards have the following format.

The field 1 is not used and should be empty.

The field 2 specifies the column name. If this field is empty, the name from the previous data card is used.

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

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

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

Elements of constraint matrix (i.e. constraint coefficients) should follow in the column wise manner: all elements for the current column should be specified before specifying 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's allowed to specify them.

Bounds of columns should be specified in the BOUNDS section.

RHS section

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

Each data card of the RHS section specifies one or two right-hand sides b_i (see the description of the ROWS section).

The field 1 is not used and should be empty.

The field 2 specifies the name of the right-hand side (RHS) vector[3]. If this field is empty, the name from the previous data card is used.

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

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

Fields 5 and 6 are optional. If they are used, they should contain the second pair "row name—right hand side value" 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 COLUMNS section may

[3] This allows the user to specify several RHS vectors in the same MPS file. However, for solving the problem the user should choose the particular RHS vector.

differ from the order of rows in the ROWS section. Each RHS vector should have unique name.

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

RANGES section

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

Each data card of the RANGES section specifies one or two ranges for double-sided constraints, i.e. for constraints that have 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 x_i is an auxiliary variable, l_i is a lower bound, u_i is an upper bound.

The field 1 is not used and should be empty.

The field 2 specifies the name of the range vector[4]. If this field is empty, the name from the previous data card is used.

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

The field 4 specifies the range value ri for the row, whose name is specified in the field 3.

Fields 5 and 6 are optional. If they are used, they should contain the 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. Each range vector should have unique name.

Lower and upper bounds are determined by the following table:

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

BOUNDS section

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

Each data card of the BOUNDS section specifies one bound of one column (structural variable).

The field 1 specifies the bound type:

[4] This allows the user to specify several range vectors in the same MPS file. However, for solving the problem the user should choose the particular range vector.

'LO' lower bound;

'UP' upper bound;

'FX' fixed variable (lower bound equal to upper bound);

'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 contains the name of the bound vector[5]. If this field is empty, the name from the previous data card is used.

The field 3 contains the column name defined in the COLUMNS section.

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

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. Each bound vector should have unique name. The specification of the lower bound should precede the specification of the upper bound for the same column (if both they are explictly specified).

By default all columns (structural variables) are considered as non-negative, i.e. with lower bound equal to zero and upper bound equal to $+\infty$. Lower (l_j) or/and upper (u_j) bounds of each column (i.e. structural variable x_j) specified in the BOUNDS section are set in the following way (s_j denotes the corresponding bound value):

'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 $+\infty$.

If only zero upper bound is specified, it is the same as if FX type would be specified.

ENDATA indicator card

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

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

[5] This allows the user to specify several bound vectors in the same MPS file. However, for solving the problem the user should choose the particular bound vector.

Example of MPS file

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

```
   minimize

      Z = COST

   subject to linear constraints

COST = 0.03 MIX1 + 0.08 MIX2 + 0.17 MIX3 + 0.12 MIX4 + 0.15 MIX5 + 0.21 ALUM
     + 0.38 SILC
SUM  =      MIX1 +      MIX2 +      MIX3 +      MIX4 +      MIX5 +      ALUM
     + SILC
FE   = 0.15 MIX1 + 0.04 MIX2 + 0.02 MIX3 + 0.04 MIX4 + 0.02 MIX5 + 0.01 ALUM
     + 0.03 SILC
CU   = 0.03 MIX1 + 0.05 MIX2 + 0.08 MIX3 + 0.02 MIX4 + 0.06 MIX5 + 0.01 ALUM
MN   = 0.02 MIX1 + 0.04 MIX2 + 0.01 MIX3 + 0.02 MIX4 + 0.02 MIX5
MG   = 0.02 MIX1 + 0.03 MIX2 + 0.01 MIX5
AL   = 0.70 MIX1 + 0.75 MIX2 + 0.80 MIX3 + 0.75 MIX4 + 0.80 MIX5 + 0.97 ALUM
SI   = 0.02 MIX1 + 0.06 MIX2 + 0.08 MIX3 + 0.12 MIX4 + 0.02 MIX5 + 0.01 ALUM
     + 0.97 SILC

   and bounds of (auxiliary and structural) variables

      -inf <  COST < +inf        0 <= MIX1 <=  200
              SUM  =  200        0 <= MIX2 <= 2500
      -inf <  FE   <=  60      400 <= MIX3 <=  800
      -inf <  CU   <= 100      100 <= MIX4 <=  700
      -inf <  MN   <=  40        0 <= MIX3 <= 1500
      1500 <= AL   < +inf        0 <= ALUM <  +inf
       250 <= SI   <= 300        0 <= SILC <  +inf
```

The MPS file, which specifies data for this example LP problem, is shown below (two top comment lines show card positions).

```
*000000001111111111222222222233333333334444444445555555566
*234567890123456789012345678901234567890123456789012345678901
NAME          PLAN
ROWS
 N  COST
 E  SUM
 L  FE
 L  CU
 L  MN
 L  MG
 G  AL
 L  SI
```

```
COLUMNS
    MIX1      COST            .03000   SUM            1.00000
              FE              .15000   CU              .03000
              MN              .02000   MG              .02000
              AL              .70000   SI              .02000
    MIX2      COST            .08000   SUM            1.00000
              FE              .04000   CU              .05000
              MN              .04000   MG              .03000
              AL              .75000   SI              .06000
    MIX3      COST            .17000   SUM            1.00000
              FE              .02000   CU              .08000
              MN              .01000   AL              .80000
              SI              .08000
    MIX4      COST            .12000   SUM            1.00000
              FE              .04000   CU              .02000
              MN              .02000   AL              .75000
              SI              .12000
    MIX5      COST            .15000   SUM            1.00000
              FE              .02000   CU              .06000
              MN              .02000   MG              .01000
              AL              .80000   SI              .02000
    ALUM      COST            .21000   SUM            1.00000
              FE              .01000   CU              .01000
              AL              .97000   SI              .01000
    SILC      COST            .38000   SUM            1.00000
              FE              .03000   SI              .97000
RHS
    RHS1      SUM         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      MIX1         200.00000
 UP           MIX2        2500.00000
 LO           MIX3         400.00000
 UP           MIX3         800.00000
 LO           MIX4         100.00000
 UP           MIX4         700.00000
 UP           MIX5        1500.00000
ENDATA
```

Appendix C Output Format

This appendix describes the output format, which is used by the `glp_print_sol` routine in order to report the basis solution (see [LP Problem], page 1) found by the solver.

The typical output from the `glp_print_sol` routine looks like the following (this output corresponds to the optimal solution of the example problem from Appendix B):

```
Problem:    PLAN
Rows:       8
Columns:    7
Non-zeros:  48
Objective:  COST = 296.217 (MINimization)
Status:     OPTIMAL

Number  Row name  St   Activity     Lower bound   Upper bound    Marginal
------ ---------- -- ------------- ------------- ------------- -------------
     1 COST       B        296.217
     2 SUM        NS          2000          2000             =    -0.0135957
     3 FE         NU            60                         60      -2.56823
     4 CU         B        83.9675                        100
     5 MN         NU            40                         40     -0.544404
     6 MG         B        19.9603                         30
     7 AL         NL          1500          1500                    0.251986
     8 SI         NL           250           250           300      0.485199

Number  Col name  St   Activity     Lower bound   Upper bound    Marginal
------ ---------- -- ------------- ------------- ------------- -------------
     1 MIX1       NL             0             0           200      0.253625
     2 MIX2       B        665.343             0          2500
     3 MIX3       B        490.253           400           800
     4 MIX4       B        424.188           100           700
     5 MIX5       NL             0             0          1500      0.014556
     6 ALUM       B        299.639             0
     7 SILC       B        120.578             0

End of output
```

The output starts from the header that includes the following lines.

'`Problem:`'
: shows the problem name.

'`Rows:`'
: shows total number of rows (auxiliary variables) in the problem.

'`Columns:`'
: shows total number of columns (structural variables) in the problem.

'`Non-zeros:`'
shows total number of constraint coefficients (i.e. elements of the constraint matrix) in the problem[1].

'`Objective:`'
shows the name of the objective function, its value in the computed solution, and the optimization flag ('`Minimization`' or '`Maximization`').

'`Status:`' shows the status of the solution (for example, '`OPTIMAL`' means that the solution is optimal).

The header is followed by two tables. The first table reports rows, the second table reports columns. Both tables have identical format. (In order to obtain information included to these two tables the `glp_print_sol` routine calls some GLPK API routines. The routine doesn't access directly to the workspace.)

'`Number`' is the sequential number of the row/column.

'`Row/Column name`'
shows the name of row/column.

'`St`' shows the status of the corresponding row/column in the found basis solution. This field may contain one of the following values:

'`B`' basis variable. The asterisk '`*`', which may follow this status, means that the value of row/column is out of its bounds. This may happen if the computed solution is infeasible, or if the problem is degenerative (in the latter case the value of row/column is near to its bound);

'`NL`' non-basis variable placed on its lower bound;

'`NU`' non-basis variable placed on its upper bound;

'`NF`' non-basis free variable;

'`NS`' non-basis fixed variable.

'`Activity`'
shows the computed value of row/column in the found basis solution.

'`Lower bound`'
shows the lower bound of row/column or its fixed value (in case of fixed variable). If row/column has no lower bound, this field is empty.

'`Upper bound`'
shows the upper bound of row/column or contains '`=`', that means that the upper bound is equal to the lower bound (in case of fixed variable). If row/column has no upper bound, this field is empty.

[1] "Non-zero" means a constraint coefficient, which has been explicitly introduced to the problem. As a rule there is no need to introduce zero coefficients. However, if the user explicitly introduced zero coefficients, all they are considered as non-zeros.

'`Marginal`'

shows the computed marginal value of row/column. If the corresponding variable is basis, this field is empty, because marginal values of basis variables are always equal to zero. In case of non-basis variable this field may contain '`< eps`', which means that the marginal value is near to zero.