Pure-GLPK - GLPK interface for the Pure programming language

Author: Jiri Spitz <jiri.spitz@bluetone.cz>
Date: 2010-02-04

This module provides a feature complete GLPK interface for the Pure programming language, which lets you use all capabilities of the GNU Linear Programming Kit (GLPK) directly from Pure.

GLPK (see http://www.gnu.org/software/glpk) contains an efficient simplex LP solver, a simplex LP solver in exact arithmetics, an interior-point solver, a branch-and-cut solver for mixed integer programming and some specialized algorithms for net/grid problems. Using this interface you can build, modify and solve the problem, retrieve the solution, load and save the problem and solution data in standard formats and use any of advanced GLPK features.

The interface uses native Pure data types - lists and tuples - so that you need not perform any data conversions to/from GLPK internal data structures.

To make this module work, you must have a GLPK installation on your system, the version 4.38 is required.

Contents

1 Installation

2 Error Handling

3 Further Information and Examples

4 Interface description

5 Descriptions of interface functions

5.1 Basic API routines

5.1.1 Problem creating and modifying routines

5.1.1.1 Create the GLPK problem object

5.1.1.2 Set the problem name

5.1.1.3 Set objective name

5.1.1.4 Set the objective direction

5.1.1.5 Add new rows to the problem

5.1.1.6 Add new columns to the problem

5.1.1.7 Set the row name

5.1.1.8 Set the column name

5.1.1.9 Set (change) row bounds

5.1.1.10 Set (change) column bounds

5.1.1.11 Set (change) objective coefficient or constant term

5.1.1.12 Load or replace matrix row

5.1.1.13 Load or replace matrix column

5.1.1.14 Load or replace the whole problem matrix
5.1.1.15 Check for duplicate elements in sparse matrix
5.1.1.16 Sort elements of the constraint matrix
5.1.1.17 Delete rows from the matrix
5.1.1.18 Delete columns from the matrix
5.1.1.19 Copy the whole content of the GLPK problem object to another one
5.1.1.20 Erase all data from the GLPK problem object
5.1.1.21 Delete the GLPK problem object
5.1.2 Problem retrieving routines
5.1.2.1 Get the problem name
5.1.2.2 Get the objective name
5.1.2.3 Get the objective direction
5.1.2.4 Get number of rows
5.1.2.5 Get number of columns
5.1.2.6 Get name of a row
5.1.2.7 Get name of a column
5.1.2.8 Get row type
5.1.2.9 Get row lower bound
5.1.2.10 Get row upper bound
5.1.2.11 Get column type
5.1.2.12 Get column lower bound
5.1.2.13 Get column upper bound
5.1.2.14 Get objective coefficient
5.1.2.15 Get number of nonzero coefficients
5.1.2.16 Retrive a row from the problem matrix
5.1.2.17 Retrive a column from the problem matrix
5.1.3 Row and column searching routines
5.1.3.1 Create index for searching rows and columns by their names
5.1.3.2 Find a row number by name
5.1.3.3 Find a column number by name
5.1.3.4 Delete index for searching rows and columns by their names
5.1.4 Problem scaling routines
5.1.4.1 Set the row scale factor
5.1.4.2 Set the column scale factor
5.1.4.3 Retrieve the row scale factor
5.1.4.4 Retrieve the column scale factor
5.1.4.5 Scale the problem data according to supplied flags
5.1.4.6 Unscale the problem data
5.1.5 LP basis constructing routines
5.1.5.1 Set the row status
5.1.5.2 Set the column status
5.1.5.3 Construct standard problem basis
5.1.5.4 Construct advanced problem basis
5.1.5.5 Construct Bixby's problem basis
5.1.6 Simplex method routines
5.1.6.1 Solve the LP problem using simplex method

5.1.6.2 Solve the LP problem using simplex method in exact arithmetics
5.1.6.3 Retrieve generic status of basic solution
5.1.6.4 Retrieve generic status of primal solution
5.1.6.5 Retrieve generic status of dual solution
5.1.6.6 Retrieve value of the objective function
5.1.6.7 Retrieve generic status of a row variable
5.1.6.8 Retrieve row primal value
5.1.6.9 Retrieve row dual value
5.1.6.10 Retrieve generic status of a column variable
5.1.6.11 Retrieve column primal value
5.1.6.12 Retrieve column dual value
5.1.6.13 Determine variable causing unboundedness
5.1.7 Interior-point method routines
5.1.7.1 Solve the LP problem using interior-point method
5.1.7.2 Retrieve status of interior-point solution
5.1.7.3 Retrieve the objective function value of interior-point solution
5.1.7.4 Retrieve row primal value of interior-point solution
5.1.7.5 Retrieve row dual value of interior-point solution
5.1.7.6 Retrieve column primal value of interior-point solution
5.1.7.7 Retrieve column dual value of interior-point solution
5.1.8 Mixed integer programming routines
5.1.8.1 Set column kind
5.1.8.2 Retrieve column kind
5.1.8.3 Retrieve number of integer columns
5.1.8.4 Retrieve number of binary columns
5.1.8.5 Solve the MIP problem using branch-and-cut method
5.1.8.6 Retrieve status of mip solution
5.1.8.7 Retrieve the objective function value of mip solution
5.1.8.8 Retrieve row value of mip solution
5.1.8.9 Retrieve column value of mip solution
5.1.9 Additional routines
5.1.9.1 Check Karush-Kuhn-Tucker conditions
5.2 Utility API routines
5.2.1 Problem data reading/writing routines
5.2.1.1 Read LP problem data from a MPS file
5.2.1.2 Write LP problem data into a MPS file
5.2.1.3 Read LP problem data from a CPLEX file
5.2.1.4 Write LP problem data into a CPLEX file
5.2.1.5 Read LP problem data in GLPK format
5.2.1.6 Write LP problem data in GLPK format
5.2.2 Routines for MathProg models
5.2.2.1 Create the MathProg translator object
5.2.2.2 Read and translate model section
5.2.2.3 Read and translate data section
5.2.2.4 Generate the model

5.2.2.5 Build problem instance from the model
5.2.2.6 Postsolve the model
5.2.2.7 Delete the MathProg translator object
5.2.3 Problem solution reading/writing routines
5.2.3.1 Write basic solution in printable format
5.2.3.2 Read basic solution from a text file
5.2.3.3 Write basic solution into a text file
5.2.3.4 Print sensitivity analysis report
5.2.3.5 Write interior-point solution in printable format
5.2.3.6 Read interior-point solution from a text file
5.2.3.7 Write interior-point solution into a text file
5.2.3.8 Write MIP solution in printable format
5.2.3.9 Read MIP solution from a text file
5.2.3.10 Write MIP solution into a text file
5.3 Advanced API routines
5.3.1 LP basis routines
5.3.1.1 Check whether basis factorization exists
5.3.1.2 Compute the basis factorization
5.3.1.3 Check whether basis factorization has been updated
5.3.1.4 Get basis factorization parameters
5.3.1.5 Change basis factorization parameters
5.3.1.6 Retrieve the basis header information
5.3.1.7 Retrieve row index in the basis header
5.3.1.8 Retrieve column index in the basis header
5.3.1.9 Perform forward transformation
5.3.1.10 Perform backward transformation
5.3.1.11 Warm up LP basis
5.3.2 Simplex tableau routines
5.3.2.1 Compute row of the tableau
5.3.2.2 Compute column of the tableau
5.3.2.3 Transform explicitly specified row
5.3.2.4 Transform explicitly specified column
5.3.2.5 Perform primal ratio test
5.3.2.6 Perform dual ratio test
5.3.2.7 Analyze active bound of non-basic variable
5.3.2.8 Analyze objective coefficient at basic variable
5.4 Branch-and-cut API routines
5.4.1 Basic routines
5.4.1.1 Determine reason for calling the callback routine
5.4.1.2 Access the problem object
5.4.1.3 Determine additional row attributes
5.4.1.4 Compute relative MIP gap
5.4.1.5 Access application-specific data
5.4.1.6 Select subproblem to continue the search
5.4.1.7 Provide solution found by heuristic

5.4.1.8 Check whether can branch upon specified variable
5.4.1.9 Choose variable to branch upon
5.4.1.10 Terminate the solution process
5.4.2 The search tree exploring routines
5.4.2.1 Determine the search tree size
5.4.2.2 Determine current active subproblem
5.4.2.3 Determine next active subproblem
5.4.2.4 Determine previous active subproblem
5.4.2.5 Determine parent active subproblem
5.4.2.6 Determine subproblem level
5.4.2.7 Determine subproblem local bound
5.4.2.8 Find active subproblem with the best local bound
5.4.3 The cut pool routines
5.4.3.1 Determine current size of the cut pool
5.4.3.2 Add constraint to the cut pool
5.4.3.3 Remove constraint from the cut pool
5.4.3.4 Remove all constraints from the cut pool
5.5 Graph and network API routines
5.5.1 Basic graph routines
5.5.1.1 Create the GLPK graph object
5.5.1.2 Set the graph name
5.5.1.3 Add vertices to a graph
5.5.1.4 Add arc to a graph
5.5.1.5 Erase content of the GLPK graph object
5.5.1.6 Delete the GLPK graph object
5.5.1.7 Read graph in a plain text format
5.5.1.8 Write graph in a plain text format
5.5.2 Graph analysis routines
5.5.2.1 Find all weakly connected components of a graph
5.5.2.2 Find all strongly connected components of a graph
5.5.3 Minimum cost flow problem
5.5.3.1 Read minimum cost flow problem data in DIMACS format
5.5.3.2 Write minimum cost flow problem data in DIMACS format
5.5.3.3 Convert minimum cost flow problem to LP
5.5.3.4 Solve minimum cost flow problem with out-of-kilter algorithm
5.5.3.5 Klingman's network problem generator
5.5.3.6 Grid-like network problem generator
5.5.4 Maximum flow problem
5.5.4.1 Read maximum cost flow problem data in DIMACS format
5.5.4.2 Write maximum cost flow problem data in DIMACS format
5.5.4.3 Convert maximum flow problem to LP
5.5.4.4 Solve maximum flow problem with Ford-Fulkerson algorithm
5.5.4.5 Goldfarb's maximum flow problem generator
5.6 Miscellaneous routines
5.6.1 Library environment routines

5.6.1.1 Determine library version
5.6.1.2 Enable/disable terminal output
5.6.1.3 Enable/disable the terminal hook routine
5.6.1.4 Get memory usage information
5.6.1.5 Set memory usage limit
5.6.1.6 Free GLPK library environment

1 Installation

Run `make` to compile the module and `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have Pure installed.

The default make options suppose that GLPK was configured with the following options: `--enable-dl --enable-odbc --enable-mysql --with-gmp --with-zlib`

Using the given options the depndencies are:

- GNU Multiprecision Library (GMP) - serves for the exact simplex solver. When disabled, the exact solver still works but it is much slower.
- ODBC library - serves for reading data directly from database tables within the GNU MathProg language translator through the ODBC interface.
- zlib compression library - enables reading and writing gzip compressed problem and solution files.
- MySQL client library - serves for reading data directly from MySQL tables within the GNU MathProg language translator.
- ltdl dlopen library - must be enabled together with any of ODBC, zlib or MySQL.

`make` tries to guess your Pure installation directory and platform-specific setup. If it gets this wrong, you can set some variables manually. In particular, `make install prefix=/usr` sets the installation prefix, and `make PIC=-fPIC` or some similar flag might be needed for compilation on 64 bit systems. The variable `ODBCLIB` specifies the ODBC library to be linked with. The default value is `ODBCLIB=-lodbc`. Please see the Makefile for details.

2 Error Handling

When an error condition occurs, the GLPK library itself prints an error mesage and terminates the application. This behaviour is not pleasant when working within an interpreter. Therefore, the Pure - GLPK bindings catches at least the most common errors like indices out of bounds. On such an error an appropriate message is returned to the interpreter. The less common errors are still trapped by the GLPK library.

When one of the most common errors occurs, an error term of the form `glp::error message` will be returned, which specifies what kind of error happend. For instance, an index out of boundsd will cause a report like the following:

```
glp::error "[Pure GLPK error] row index out of bounds"
```

You can check for such return values and take some appropriate action. By redefining `glp::error` accordingly, you can also have it generate exceptions or print an error message. For instance:

```
glp::error message = fprintf stderr "%s\n" message $$ ();
```

NOTE: When redefining `glp::error` in this manner, you should be aware that the return value of `glp::error` is what will be returned by the other operations of this module in case of an error condition. These return values are checked by other functions. Thus the return value should still indicate that an error has happened, and not be something that might be interpreted as a legal return value, such as an

integer or a nonempty tuple. It is usually safe to have `glp::error` return an empty tuple or throw an exception, but other types of return values should be avoided.

IMPORTANT: It is really good to define a `glp::error` function, otherwise the errors might remain unnoticed.

3 Further Information and Examples

For further details about the operations provided by this module please see the GLPK Reference Manual. Sample scripts illustrating the usage of the module can be found in the examples directory.

4 Interface description

Most GLPK functions and symbols live in the namespace `glp`. There are a few functions and symbols in the namespace `lpx`. These functions and symbols are likely to be removed and replaced by new ones in the future.

In general, when you replace the `glp_` prefix from the GLPK Reference Manual with the namespace specification `glp::` then you receive the function name in this module. The same is valid for `lpx_` and `lpx::`. The symbolic constants are converted into lower case in this module, again obeying the same prefix rules.

5 Descriptions of interface functions

5.1 Basic API routines

5.1.1 Problem creating and modifying routines

5.1.1.1 Create the GLPK problem object Synopsis:

`glp::create_prob`

Parameters:

none

Returns:

pointer to the LP problem object

Example:

```
> let lp = glp::create_prob;
> lp;
#<pointer 0x9de7168>
>
```

5.1.1.2 Set the problem name Synopsis:

`glp::set_prob_name lp name`

Parameters:

lp: pointer to the LP problem object
name: problem name

Returns:

```
()
```

Example:

```
> glp::set_prob_name lp "Testing problem";
()
>
```

5.1.1.3 Set objective name Synopsis:

```
glp::set_obj_name lp name
```

Parameters:

lp: pointer to the LP problem object
name: objective name

Returns:

```
()
```

Example:

```
> glp::set_obj_name lp "Total costs";
()
>
```

5.1.1.4 Set the objective direction Synopsis:

```
glp::set_obj_dir lp direction
```

Parameters:

lp: pointer to the LP problem object
direction: one of the following:
glp::min: minimize
glp::max: maximize

Returns:

```
()
```

Example:

```
> glp::set_obj_dir lp glp::min;
()
>
```

5.1.1.5 Add new rows to the problem Synopsis:

```
glp::add_rows lp count
```

Parameters:

lp: pointer to the LP problem object
count: number of rows to add

Returns:

index of the first row added

Example:

```
> let first_added_row = glp_add_rows lp 3;
> first_added_row;
6
>
```

5.1.1.6 Add new columns to the problem Synopsis:

```
glp::add_cols lp count
```

Parameters:

lp: pointer to the LP problem object
count: number of columns to add

Returns:

index of the first column added

Example:

```
> let first_added_col = glp_add_cols lp 3;
> first_added_col;
5
>
```

5.1.1.7 Set the row name Synopsis:

```
glp::set_row_name lp (rowindex, rowname)
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index
rowname: row name

Returns:

```
()
```

Example:

```
> glp::set_row_name lp (3, "The third row");
()
>
```

5.1.1.8 Set the column name Synopsis:

```
glp::set_col_name lp (colindex, colname)
```

Parameters:

lp: pointer to the LP problem object
colindex: column index
colname: column name

Returns:

```
()
```

Example:

```
> glp::set_col_name lp (3, "The third column");
()
>
```

5.1.1.9 Set (change) row bounds Synopsis:

```
glp::set_row_bnds lp (rowindex, rowtype, lowerbound, upperbound)
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index
rowtype: one of the following:
- **glp::fr:** free variable (both bounds are ignored)
- **glp::lo:** variable with lower bound (upper bound is ignored)
- **glp::up:** variable with upper bound (lower bound is ignored)
- **glp::db:** double bounded variable
- **glp::fx:** fixed variable (lower bound applies, upper bound is ignored)

lowerbound: lower row bound
upperbound: upper row bound

Returns: `()`

Example:: `glp::set_row_bnds lp (3, glp::up, 0.0, 150.0);`

5.1.1.10 Set (change) column bounds Synopsis:

```
glp::set_col_bnds lp (colindex, coltype, lowerbound, upperbound)
```

Parameters:

lp: pointer to the LP problem object
colindex: column index
coltype: one of the following:
- **glp::fr:** free variable (both bounds are ignored)
- **glp::lo:** variable with lower bound (upper bound is ignored)
- **glp::up:** variable with upper bound (lower bound is ignored)
- **glp::db:** double bounded variable

glp::fx: fixed variable (lower bound applies, upper bound is ignored)

lowerbound: lower column bound

upperbound: upper column bound

Returns:

```
()
```

Example:

```
> glp::set_col_bnds lp (3, glp::db, 100.0, 150.0);
()
>
```

5.1.1.11 Set (change) objective coefficient or constant term Synopsis:

```
glp::set_obj_coef lp (colindex, coefficient)
```

Parameters:

lp: pointer to the LP problem object

colindex: column index, zero index denotes the constant term (objective shift)

Returns:

```
()
```

Example:

```
> glp::set_obj_coef lp (3, 15.8);
()
>
```

5.1.1.12 Load or replace matrix row Synopsis:

```
glp::set_mat_row lp (rowindex, rowvector)
```

Parameters:

lp: pointer to the LP problem object

rowindex: row index

rowvector: list of tuples (colindex, coefficient); only non-zero coefficients have to be specified, the order of column indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Example:

```
> glp::set_mat_row lp (3, [(1, 3.0), (4, 5.2)]);
()
>
```

5.1.1.13 Load or replace matrix column Synopsis:

```
glp::set_mat_col lp (colindex, colvector)
```

Parameters:

lp: pointer to the LP problem object
colindex: column index
colvector: list of tuples (rowindex, coefficient); only non-zero coefficients have to be specified, the order of row indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Example:

```
> glp::set_mat_col lp (2, [(4, 2.0), (2, 1.5)]);
()
>
```

5.1.1.14 Load or replace the whole problem matrix Synopsis:

```
glp::load_matrix lp matrix
```

Parameters:

lp: pointer to the LP problem object
matrix: list of tuples (rowindex, colindex, coefficient); only non-zero coefficients have to be specified, the order of indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Example:

```
> glp::load_matrix lp [(1, 3, 5.0), (2, 2, 3.5), (3, 1, -
2.0), (3, 2, 1.0)];
()
>
```

5.1.1.15 Check for duplicate elements in sparse matrix Synopsis:

```
glp::check_dup numrows numcols indices
```

Parameters:

numrows: number of rows
numcols: number of columns
indices: list of tuples (rowindex, colindex); indices of only non-zero coefficients have to be specified, the order of indices is not important

Returns:

returns one of the following:

0: the matrix has no duplicate elements
-k: rowindex or colindex of the k-th element in indices is out of range
+k: the k-th element in indices is duplicate

Remark:

Notice, that `k` counts from 1, whereas list members are counted from 0.

Example:

```
> glp::check_dup 3 3  [(1, 3), (2, 2), (3, 1), (2, 2)];
4
>
```

5.1.1.16 Sort elements of the constraint matrix Synopsis:

```
glp::sort_matrix lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::sort_matrix lp;
()
>
```

5.1.1.17 Delete rows from the matrix Synopsis:

```
glp::del_rows lp rows
```

Parameters:

lp: pointer to the LP problem object
rows: list of indices of rows to be deleted; the order of indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Remark:

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.

Example:

```
> glp::del_rows lp [3, 4, 7];
()
>
```

5.1.1.18 Delete columns from the matrix Synopsis:

```
glp::del_cols lp cols
```

Parameters:

lp: pointer to the LP problem object
cols: list of indices of columns to be deleted; the order of indices is not important, duplicates are **not** allowed

Returns:

```
()
```

Remark:

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.

Example:

```
> glp::del_cols lp [6, 4, 5];
()
>
```

5.1.1.19 Copy the whole content of the GLPK problem object to another one Synopsis:

```
glp::copy_prob destination source names
```

Parameters:

destination: pointer to the destination LP problem object (must already exist)
source: pointer to the source LP problem object
names: one of the following:
glp::on: copy all symbolic names as well
glp::off: do not copy the symbolic names

Returns:

```
()
```

Example:

```
> glp::copy_prob lp_dest lp_src glp::on;
()
>
```

5.1.1.20 Erase all data from the GLPK problem object Synopsis:

```
glp::erase_prob lp
```

Parameters:

lp: pointer to the LP problem object, it remains still valid after the function call

Returns:

```
()
```

Example:

```
> glp::erase_prob lp;
()
>
```

5.1.1.21 Delete the GLPK problem object Synopsis:

```
glp::delete_prob lp
```

Parameters:

lp: pointer to the LP problem object, it is not valid any more after the function call

Returns:

```
()
```

Example:

```
> glp::delete_prob lp;
()
>
```

5.1.2 Problem retrieving routines

5.1.2.1 Get the problem name Synopsis:

```
glp::get_prob_name lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

name of the problem

Example:

```
> glp::get_prob_name lp;
"Testing problem"
>
```

5.1.2.2 Get the objective name Synopsis:

```
glp::get_obj_name lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

name of the objective

Example:

```
> glp::get_obj_name lp;
"Total costs"
>
```

5.1.2.3 Get the objective direction Synopsis:

```
glp::get_obj_dir lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

returns one of the following:

glp::min: minimize
glp::max: maximize

Example:

```
> glp::get_obj_dir lp;
glp::min
>
```

5.1.2.4 Get number of rows Synopsis:

```
glp::get_num_rows lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

number of rows (constraints)

Example:

```
> glp::get_num_rows lp;
58
>
```

5.1.2.5 Get number of columns Synopsis:

```
glp::get_num_cols lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

number of columns (structural variables)

Example:

```
> glp::get_num_cols lp;
65
>
```

5.1.2.6 Get name of a row Synopsis:

```
glp::get_row_name lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

name of the given row

Example:

```
> glp::get_row_name lp 3;
"The third row"
>
```

5.1.2.7 Get name of a column Synopsis:

```
glp::get_col_name lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

name of the given column

Example:

```
> glp::get_col_name lp 2;
"The second column"
>
```

5.1.2.8 Get row type Synopsis:

```
glp::get_row_type lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

returns one of the following:

glp::fr: free variable
glp::lo: variable with lower bound
glp::up: variable with upper bound
glp::db: double bounded variable
glp::fx: fixed variable

Example:

```
> glp::get_row_type lp 3;
glp::db
>
```

5.1.2.9 Get row lower bound Synopsis:

```
glp::get_row_lb lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

the row lower bound; if the row has no lower bound then it returns the smallest double number

Example:

```
> glp::get_row_lb lp 3;
50.0
>
```

5.1.2.10 Get row upper bound Synopsis:

```
glp::get_row_ub lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

the row upper bound; if the row has no upper bound then it returns the biggest double number

Example:

```
> glp::get_row_ub lp 3;
150.0
>
```

5.1.2.11 Get column type Synopsis:

```
glp::get_col_type lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

returns one of the following:

glp::fr: free variable
glp::lo: variable with lower bound
glp::up: variable with upper bound
glp::db: double bounded variable
glp::fx: fixed variable

Example:

```
> glp::get_col_type lp 2;
glp::up
>
```

5.1.2.12 Get column lower bound Synopsis:

```
glp::get_col_lb lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

the column lower bound; if the column has no lower bound then it returns the smallest double number

Example:

```
> glp::get_col_lb lp 3;
-1.79769313486232e+308
>
```

5.1.2.13 Get column upper bound Synopsis:

```
glp::get_col_ub lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

the column upper bound; if the column has no upper bound then it returns the biggest double number

Example:

```
> glp::get_col_lb lp 3;
150.0
>
```

5.1.2.14 Get objective coefficient Synopsis:

```
glp::get_obj_coef lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index; zero index denotes the constant term (objective shift)

Returns:

the coefficient of given column in the objective

Example:

```
> glp::get_obj_coef lp 3;
5.8
>
```

5.1.2.15 Get number of nonzero coefficients Synopsis:

```
glp::get_num_nz lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

number of non-zero coefficients in the problem matrix

Example:

```
> glp::get_num_nz lp;
158
>
```

5.1.2.16 Retrive a row from the problem matrix Synopsis:

```
glp::get_mat_row lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

non-zero coefficients of the given row in a list form of tuples (colindex, coefficient)

Example:

```
> get_mat_row lp 3;
[(3,6.0),(2,2.0),(1,2.0)]
>
```

5.1.2.17 Retrive a column from the problem matrix Synopsis:

```
glp::get_mat_col lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

non-zero coefficients of the given column in a list form of tuples (rowindex, coefficient)

Example:

```
> get_mat_col lp 2;
[(3,2.0),(2,4.0),(1,1.0)]
>
```

5.1.3 Row and column searching routines

5.1.3.1 Create index for searching rows and columns by their names Synopsis:

```
glp::create_index lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::create_index lp;
()
>
```

5.1.3.2 Find a row number by name Synopsis:

```
glp::find_row lp rowname
```

Parameters:

lp: pointer to the LP problem object
rowname: row name

Returns:

ordinal number (index) of the row

Remark:

The search index is automatically created if it does not already exists.

Example:

```
> glp::find_row lp "The third row";
3
>
```

5.1.3.3 Find a column number by name Synopsis:

```
glp::find_col lp colname
```

Parameters:

lp: pointer to the LP problem object
colname: column name

Returns:

ordinal number (index) of the column

Remark:

The search index is automatically created if it does not already exists.

Example:

```
> glp::find_col lp "The second row";
2
>
```

5.1.3.4 Delete index for searching rows and columns by their names Synopsis:

```
glp::delete_index lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::delete:index lp;
()
>
```

5.1.4 Problem scaling routines

5.1.4.1 Set the row scale factor Synopsis:

```
glp::set_rii lp (rowindex, coefficient)
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index
coefficient: scaling coefficient

Returns:

```
()
```

Example:

```
> glp::set_rii lp (3, 258.6);
()
>
```

5.1.4.2 Set the column scale factor Synopsis:

```
glp::set_sjj lp (colindex, coefficient)
```

Parameters:

lp: pointer to the LP problem object
colindex: column index
coefficient: scaling coefficient

Returns:

```
()
```

Example:

```
> glp::set_sjj lp (2, 12.8);
()
>
```

5.1.4.3 Retrieve the row scale factor Synopsis:

```
glp::get_rii lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

scaling coefficient of given row

Example:

```
> glp::get_rii lp 3;
258.6
>
```

5.1.4.4 Retrieve the column scale factor Synopsis:

```
glp::get_sjj lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

scaling coefficient of given column

Example:

```
> glp::get_sjj lp 2;
12.8
>
```

5.1.4.5 Scale the problem data according to supplied flags Synopsis:

```
glp::scale_prob lp flags
```

Parameters:

lp: pointer to the LP problem object
flags: symbolic integer constants which can be combined together by arithmetic *or*; the possible constants are:
glp::sf_gm: perform geometric mean scaling
glp::sf_eq: perform equilibration scaling
glp::sf_2n: round scale factors to power of two
glp::sf_skip: skip if problem is well scaled
glp::sf_auto: choose scaling options automatically

Returns:

```
()
```

Example:

```
> glp::scale_prob lp (glp::sf_gm || glp::sf_2n);
()
>
```

5.1.4.6 Unscale the problem data Synopsis:

```
glp::unscale_prob lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::unscale_prob lp;
()
>
```

5.1.5 LP basis constructing routines

5.1.5.1 Set the row status Synopsis:

```
glp::set_row_stat lp (rowindex, status)
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index
status: one of the following:
- **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 the case of this routine)
- **glp::nf:** the same as glp::nl (only in the case of this routine)
- **glp::ns:** the same as glp::nl (only in the case of this routine)

Returns:

```
()
```

Example:

```
> glp::set_row_stat lp (3, glp::nu);
()
>
```

5.1.5.2 Set the column status Synopsis:

```
glp::set_col_stat lp (colindex, status)
```

Parameters:

lp: pointer to the LP problem object
colindex: column index
status: one of the following:
- **glp::bs:** make the column basic

glp::nl: make the column non-basic

glp::nu: make the column non-basic and set it to the upper bound; if the column is not double-bounded, this status is equivalent to glp::nl (only in the case of this routine)

glp::nf: the same as glp::nl (only in the case of this routine)

glp::ns: the same as glp::nl (only in the case of this routine)

Returns:

```
()
```

Example:

```
> glp::set_col_stat lp (2, glp::bs);
()
>
```

5.1.5.3 Construct standard problem basis Synopsis:

```
glp::std_basis lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::std_basis lp;
()
>
```

5.1.5.4 Construct advanced problem basis Synopsis:

```
glp::adv_basis lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::adv_basis lp;
()
>
```

5.1.5.5 Construct Bixby's problem basis Synopsis:

```
glp::cpx_basis lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::cpx_basis lp;
()
>
```

5.1.6 Simplex method routines

5.1.6.1 Solve the LP problem using simplex method Synopsis:

```
glp::simplex lp options
```

Parameters:

lp: pointer to the LP problem object

options: list of solver options in the form of tuples (option_name, value):

- **glp::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)
- **glp::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
- **glp::pricing:** (default: glp::pt_pse) - pricing technique
  - **glp::pt_std:** standard (textbook)
  - **glp::pt_pse:** projected steepest edge
- **glp::r_test:** (default: glp::rt_har) - ratio test technique
  - **glp::rt_std:** standard (textbook)
  - **glp::rt_har:** Harris' two-pass ratio test
- **glp::tol_bnd:** (default: 1e-7) - tolerance used to check if the basic solution is primal feasible
- **glp::tol_dj:** (default: 1e-7) - tolerance used to check if the basic solution is dual feasible
- **glp::tol_piv:** (default: 1e-10) - tolerance used to choose eligble pivotal elements of the simplex table

glp::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

glp::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

glp::it_lim: (default: INT_MAX) - simplex iteration limit

glp::tm lim: (default: INT_MAX) - searching time limit, in milliseconds

glp::out_frq: (default: 200) - output frequency, in iterations - this parameter specifies how frequently the solver sends information about the solution process to the terminal

glp::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

glp::presolve: (default: glp::off) - LP presolver option:

glp::on: enable using the LP presolver

glp::off: disable using the LP presolver

Returns:

one of the following:

glp::ok: 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)

When the list of options contains some bad option(s) then a list of bad options is returned instead.

Remark:

Options not mentioned in the option list are set to their default values.

Example:

```
> glp::simplex lp [(glp::presolve, glp::on), (glp::msg_lev, glp::msg_all)];
glp_simplex: original LP has 3 rows, 3 columns, 9 non-zeros
glp_simplex: presolved LP has 3 rows, 3 columns, 9 non-zeros
Scaling...
 A: min|aij| = 1,000e+000  max|aij| = 1,000e+001  ratio = 1,000e+001
Problem data seem to be well scaled
Crashing...
Size of triangular part = 3
*     0: obj =  0,000000000e+000  infeas = 0,000e+000 (0)
*     2: obj =  7,333333333e+002  infeas = 0,000e+000 (0)
OPTIMAL SOLUTION FOUND
glp::ok
>
```

5.1.6.2 Solve the LP problem using simplex method in exact arithmetics

Synopsis:

```
glp::exact lp options
```

Parameters:

lp: pointer to the LP problem object

options: list of solver options in the form of tuples (option_name, value):

glp::it_lim: (default: INT_MAX) - simplex iteration limit

glp::tm lim: (default: INT_MAX) - searching time limit, in miliseconds

Returns:

one of the following:

glp::ok: 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::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::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

When the list of options contains some bad option(s) then a list of bad options is returned instead.

Remark:

Options not mentioned in the option list are set to their default values.

Example:

```
> glp::exact lp [];
glp_exact: 3 rows, 3 columns, 9 non-zeros
GNU MP bignum library is being used
*     2:   objval =                      0   (0)
*     4:   objval =       733,333333333333   (0)
OPTIMAL SOLUTION FOUND
glp::ok
>
```

5.1.6.3 Retrieve generic status of basic solution Synopsis:

```
glp::get_status lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

one of the following:

glp::undef: solution is undefined

glp::feas: solution is feasible

glp::infeas: solution is infeasible

glp::nofeas: no feasible solution exists

glp::opt: solution is optimal

glp::unbnd: solution is unbounded

Example:

```
> glp::get_status lp;
glp::opt
>
```

5.1.6.4 Retrieve generic status of primal solution Synopsis:

```
glp::get_prim_stat lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

one of the following:

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

Example:

```
> glp::get_prim_stat lp;
glp::feas
>
```

5.1.6.5 Retrieve generic status of dual solution Synopsis:

```
glp::get_dual_stat lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

one of the following:

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

Example:

```
> glp::get_dual_stat lp;
glp::feas
>
```

5.1.6.6 Retrieve value of the objective function Synopsis:

```
glp::get_obj_val lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

value of the objective function

Example:

```
> glp::get_obj_val lp
733.333333333333
>
```

5.1.6.7 Retrieve generic status of a row variable Synopsis:

```
glp::get_row_stat lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

one of the following:

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

Example:

```
> glp::get_row_stat lp 3;
glp::bs
>
```

5.1.6.8 Retrieve row primal value

Synopsis:: glp::get_row_prim lp rowindex

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

primal value of the row (auxiliary) variable

Example:

```
> glp::get_row_prim lp 3;
200.0
>
```

5.1.6.9 Retrieve row dual value Synopsis:

```
glp::get_row_dual lp rowindex
```

Parameters:

lp: pointer to the LP problem object
rowindex: row index

Returns:

dual value of the row (auxiliary) variable

Example:

```
> glp::get_row_dual lp 3;
0.0
>
```

5.1.6.10 Retrieve generic status of a column variable Synopsis:

```
glp::get_col_stat lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

one of the following:

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

Example:

```
> glp::get_col_stat lp 2;
glp::bs
>
```

5.1.6.11 Retrieve column primal value Synopsis:

```
glp::get_col_prim lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

primal value of the column (structural) variable

Example:

```
> glp::get_col_prim lp 2;
66.6666666666667
>
```

5.1.6.12 Retrieve column dual value Synopsis:

```
glp::get_col_dual lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

dual value of the column (structural) variable

Example:

```
> glp::get_col_dual lp 2;
0.0
>
```

5.1.6.13 Determine variable causing unboundedness Synopsis:

```
glp::get_unbnd_ray lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

The routine glp_get_unbnd_ray returns the number k of a variable, which causes primal or dual unboundedness. If 1 <= k <= m, it is k-th auxiliary variable, and if m + 1 <= k <= 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.

Remark:

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).

Example:

```
> glp::get_unbnd_ray lp;
0
>
```

5.1.7 Interior-point method routines

5.1.7.1 Solve the LP problem using interior-point method Synopsis:

```
glp::interior lp options
```

Parameters:

lp: pointer to the LP problem object

options: list of solver options in the form of tuples (option_name, value):

- **glp::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)
- **glp::ord_alg:** (default: glp::ord_amd) - ordering algorithm option
  - **glp::ord_none:** use natural (original) ordering
  - **glp::ord_qmd:** quotient minimum degree (QMD)
  - **glp::ord_amd:** approximate minimum degree (AMD)
  - **glp::ord_sysamd:** approximate minimum degree (SYSAMD)

Returns:

one of the following:

glp::ok: 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

Example:

```
> glp::interior lp [(glp::ord_alg, glp::ord_amd)];
Original LP has 3 row(s), 3 column(s), and 9 non-zero(s)
Working LP has 3 row(s), 6 column(s), and 12 non-zero(s)
Matrix A has 12 non-zeros
Matrix S = A*A' has 6 non-zeros (upper triangle)
Approximate minimum degree ordering (AMD)...
Computing Cholesky factorization S = L*L'...
Matrix L has 6 non-zeros
Guessing initial point...
Optimization begins...
  0: obj = -8,218489503e+002; rpi = 3,6e-001; rdi = 6,8e-001; gap = 2,5e-
001
  1: obj = -6,719060895e+002; rpi = 3,6e-002; rdi = 1,9e-001; gap = 1,4e-
002
  2: obj = -6,917210389e+002; rpi = 3,6e-003; rdi = 9,3e-002; gap = 3,0e-
002
  3: obj = -7,267557732e+002; rpi = 2,1e-003; rdi = 9,3e-003; gap = 4,4e-
002
  4: obj = -7,323038146e+002; rpi = 2,1e-004; rdi = 1,1e-003; gap = 4,8e-
003
  5: obj = -7,332295932e+002; rpi = 2,1e-005; rdi = 1,1e-004; gap = 4,8e-
004
  6: obj = -7,333229585e+002; rpi = 2,1e-006; rdi = 1,1e-005; gap = 4,8e-
005
  7: obj = -7,333322959e+002; rpi = 2,1e-007; rdi = 1,1e-006; gap = 4,8e-
006
  8: obj = -7,333332296e+002; rpi = 2,1e-008; rdi = 1,1e-007; gap = 4,8e-
007
  9: obj = -7,333333230e+002; rpi = 2,1e-009; rdi = 1,1e-008; gap = 4,8e-
008
 10: obj = -7,333333323e+002; rpi = 2,1e-010; rdi = 1,1e-009; gap = 4,8e-
009
OPTIMAL SOLUTION FOUND
glp::ok
>
```

5.1.7.2 Retrieve status of interior-point solution Synopsis:

```
glp::ipt_status lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

one of the following

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

Example:

```
> glp::ipt_status lp;
glp::opt
>
```

5.1.7.3 Retrieve the objective function value of interior-point solution Synopsis:

```
glp::ipt_obj_val lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

objective function value of interior-point solution

Example:

```
> glp::ipt_obj_val lp;
733.333332295849
>
```

5.1.7.4 Retrieve row primal value of interior-point solution Synopsis:

```
glp::ipt_row_prim lp rowindex
```

Parameters:

lp: pointer to the LP problem object

rowindex: row index

Returns:

primal value of the row (auxiliary) variable

Example:

```
> glp::ipt_row_prim lp 3;
200.000000920688
>
```

5.1.7.5 Retrieve row dual value of interior-point solution Synopsis:

```
glp::ipt_row_dual lp rowindex
```

Parameters:

lp: pointer to the LP problem object

rowindex: row index

Returns:

dual value of the row (auxiliary) variable

Example:

```
> glp::ipt_row_dual lp 3;
2.50607466186742e-008
>
```

5.1.7.6 Retrieve column primal value of interior-point solution Synopsis:

```
glp::ipt_col_prim lp colindex
```

Parameters:

lp: pointer to the LP problem object

colindex: column index

Returns:

primal value of the column (structural) variable

Example:

```
> glp::ipt_col_prim lp 2;
66.666666406779
>
```

5.1.7.7 Retrieve column dual value of interior-point solution Synopsis:

```
glp::ipt_col_dual lp colindex
```

Parameters:

lp: pointer to the LP problem object

colindex: column index

Returns:

dual value of the column (structural) variable

Example:

```
> glp::ipt_col_dual lp 2;
2.00019467655466e-009
>
```

5.1.8 Mixed integer programming routines

5.1.8.1 Set column kind Synopsis:

```
glp::set_col_kind lp (colindex, colkind)
```

Parameters:

lp: pointer to the LP problem object
colindex: column index
colkind: column kind - one of the following:
glp::cv: continuous variable
glp::iv: integer variable
glp::bv: binary variable

Returns:

```
()
```

Example:

```
> glp::set_col_kind lp (1, glp::iv);
()
>
```

5.1.8.2 Retrieve column kind Synopsis:

```
glp::get_col_kind lp colindex
```

Parameters:

lp: pointer to the LP problem object
colindex: column index

Returns:

one of the following:

glp::cv: continuous variable
glp::iv: integer variable
glp::bv: binary variable

Example:

```
> glp::get_col_kind lp 1;
glp::iv
>
```

5.1.8.3 Retrieve number of integer columns Synopsis:

```
glp::get_num_int lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

number of integer columns (including binary columns)

Example:

```
> glp_get_num_int lp;
1
>
```

5.1.8.4 Retrieve number of binary columns Synopsis:

```
glp::get_num_bin lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

number of binary columns

Example:

```
> glp::get_num_bin lp
0
>
```

5.1.8.5 Solve the MIP problem using branch-and-cut method Synopsis:

```
glp::intopt lp options
```

Parameters:

lp: pointer to the LP problem object

options: list of solver options in the form of tuples (option_name, value):

glp::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)

glp::br_tech: (default: glp::bt::blb) - branching technique

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 pseudocost heuristic

glp::bt_tech: (default: glp::pt_pse) - backtracking technique

glp::bt_dfs: depth first search;

glp::bt_bfs: breadth first search;

glp::bt_blb: best local bound;

glp::bt_bph: best projection heuristic.

glp::pp_tech: (default: glp::pp_all) - preprocessing technique

glp::pp_none: disable preprocessing;

glp::pp_root: perform preprocessing only on the root level

glp::pp_all: perform preprocessing on all levels

glp::fp_heur: (default: glp::off) - feasibility pump heuristic:

glp::on: enable applying the feasibility pump heuristic

glp::off: disable applying the feasibility pump heuristic

glp::gmi_cuts: (default: glp::off) - Gomory's mixed integer cuts:

glp::on: enable generating Gomory's cuts;

glp::off: disable generating Gomory's cuts.

glp::mir_cuts: (default: glp::off) - mixed integer rounding (MIR) cuts:

glp::on: enable generating MIR cuts;

glp::off: disable generating MIR cuts.

glp::cov_cuts: (default: glp::off) - mixed cover cuts:

glp::on: enable generating mixed cover cuts;

glp::off: disable generating mixed cover cuts.

glp::clq_cuts (default: glp::off) - clique cuts:

glp::on: enable generating clique cuts;

glp::off: disable generating clique cuts.

glp::tol_int: (default: 1e-5) - absolute tolerance used to check if optimal solution to the current LP relaxation is integer feasible

glp::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

glp::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

glp::tm lim: (default: INT_MAX) - searching time limit, in milliseconds

glp::out_frq: (default: 5000) - output frequency, in miliseconds - this parameter specifies how frequently the solver sends information about the solution process to the terminal

glp::out_dly: (default: 10000) - output delay, in milliseconds - this parameter specifies how long the solver should delay sending information about the solution of the current LP relaxation with the simplex method to the terminal

glp::cb_func: (default: glp::off) - specifies whether to use the user-defined callback routine

glp::on: use user-defined callback function - the function `glp::mip_cb tree info` **must** be defined by the user

glp::off: do not use user-defined callback function

glp::cb_info: (default: NULL) - transit pointer passed to the routine `glp::mip_cb tree info` (see above)

glp::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

glp::presolve: (default: glp::off) - LP presolver option:

glp::on: enable using the MIP presolver

glp::off: disable using the MIP presolver

glp::binarize: (default: glp::off) - binarization (used only if the presolver is enabled):

glp::on: replace general integer variables by binary ones

glp::off: do not use binarization

Returns:

one of the following:

glp::ok: 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 (auxiliary or structural) 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

When the list of options contains some bad option(s) then a list of bad options is returned instead.

Remark:

Options not mentioned in the option list are set to their default values.

Example:

```
> glp::intopt lp [(glp::presolve, glp::on)];
ipp_basic_tech:  0 row(s) and 0 column(s) removed
ipp_reduce_bnds: 2 pass(es) made, 3 bound(s) reduced
ipp_basic_tech:  0 row(s) and 0 column(s) removed
ipp_reduce_coef: 1 pass(es) made, 0 coefficient(s) reduced
glp_intopt: presolved MIP has 3 rows, 3 columns, 9 non-zeros
glp_intopt: 3 integer columns, none of which are binary
Scaling...
 A: min|aij| =  1,000e+00  max|aij| =  1,000e+01  ratio =  1,000e+01
Problem data seem to be well scaled
Crashing...
Size of triangular part = 3
```

```
Solving LP relaxation...
*     2: obj =   0,000000000e+00  infeas =  0,000e+00 (0)
*     5: obj =   7,333333333e+02  infeas =  0,000e+00 (0)
OPTIMAL SOLUTION FOUND
Integer optimization begins...
+     5: mip =     not found yet <=              +inf        (1; 0)
+     6: >>>>>   7,320000000e+02 <=   7,320000000e+02   0.0% (2; 0)
+     6: mip =   7,320000000e+02 <=     tree is empty   0.0% (0; 3)
INTEGER OPTIMAL SOLUTION FOUND
glp::ok
>
```

5.1.8.6 Retrieve status of mip solution Synopsis:

```
glp::mip_status lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

one of the following:

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)

Example:

```
> glp::mip_status lp;
glp::opt
>
```

5.1.8.7 Retrieve the objective function value of mip solution Synopsis:

```
glp::mip_obj_val lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

objective function value of mip solution

Example:

```
> glp::mip_obj_val lp;
732.0
>
```

5.1.8.8 Retrieve row value of mip solution Synopsis:

```
glp::mip_row_val lp rowindex
```

Parameters:

lp: pointer to the LP problem object

rowindex: row index

Returns:

row value (value of auxiliary variable)

Example:

```
> glp::mip_row_val lp 3;
200.0
>
```

5.1.8.9 Retrieve column value of mip solution Synopsis:

```
glp::mip_col_val lp colindex
```

Parameters:

lp: pointer to the LP problem object

colindex: column index

Returns:

column value (value of structural variable)

Example:

```
> glp::mip_col_val lp 2;
67.0
>
```

5.1.9 Additional routines

5.1.9.1 Check Karush-Kuhn-Tucker conditions Synopsis:

```
lpx::check_kkt lp scaled
```

Parameters:

lp: pointer to the LP problem object

scaled: one of the following:

true: test the scaled problem

false: test the unscaled problem

Returns:

list of four tuples with five mebers (see GLPK reference manual):

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

where number of variable is (1 <= k <= m) for auxiliary variable and (m+1 <= k <= m+n) for structural variable

Example:

```
> lpx::check_kkt lp true;
[(1.4210854715202e-14,2,3.54385404369127e-17,3,"H"),(0.0,0,0.0,0,"H"),
(4.44089209850063e-16,1,2.11471052309554e-17,1,"H"),(0.0,0,0.0,0,"H")]
>
```

5.2 Utility API routines

5.2.1 Problem data reading/writing routines

5.2.1.1 Read LP problem data from a MPS file Synopsis:

```
glp::read_mps lp format filename
```

Parameters:

lp: pointer to the LP problem object

format: one of the following

glp::mps_deck: fixed (ancient) MPS file format

glp::mps_file: free (modern) MPS file format

filename: file name - 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"

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_mps lp glp::mps_deck "examples/plan.mps";
Reading problem data from 'examples/plan.mps'...
Problem PLAN
Objective R0000000
8 rows, 7 columns, 55 non-zeros
63 records were read
0
>
```

5.2.1.2 Write LP problem data into a MPS file Synopsis:

```
glp::write_mps lp format filename
```

Parameters:

lp: pointer to the LP problem object

format: one of the following

glp::mps_deck: fixed (ancient) MPS file format

glp::mps_file: free (modern) MPS file format

filename: file name - 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:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_mps lp glp::mps_file "examples/plan1.mps";
Writing problem data to 'examples/plan1.mps'...
63 records were written
0
>
```

5.2.1.3 Read LP problem data from a CPLEX file Synopsis:

```
glp::read_lp lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name - 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"

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::read_lp lp "examples/plan.lp";
reading problem data from 'examples/plan.lp'...
8 rows, 7 columns, 48 non-zeros
39 lines were read
0
>
```

5.2.1.4 Write LP problem data into a CPLEX file Synopsis:

```
glp::write_lp lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name - 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:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_lp lp "examples/plan1.lp";
writing problem data to 'examples/plan1.lp'...
29 lines were written
0
>
```

5.2.1.5 Read LP problem data in GLPK format Synopsis:

```
glp::read_prob lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name - 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"

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::read_prob lp "examples/plan.glpk";
reading problem data from 'examples/plan.glpk'...
8 rows, 7 columns, 48 non-zeros
86 lines were read
0
>
```

5.2.1.6 Write LP problem data in GLPK format Synopsis:

```
glp::write_prob lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name - 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:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_prob lp "examples/plan1.glpk";
writing problem data to 'examples/plan1.glpk'...
86 lines were written
0
>
```

5.2.2 Routines for MathProg models

5.2.2.1 Create the MathProg translator object Synopsis:

```
glp::mpl_alloc_wksp
```

Parameters:

none

Returns:

pointer to the MathProg translator object

Example:

```
> let mpt = glp::mpl_alloc_wksp;
> mpt;
#<pointer 0xa0d0180>
>
```

5.2.2.2 Read and translate model section Synopsis:

```
glp::mpl_read_model tranobject filename skip
```

Parameters:

tranobject: pointer to the MathProg translator object

filename: file name

skip: if **0** then the data section from the model file is read; if non-zero, the data section in the data model is skipped

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> mpl_read_model mpt "examples/sudoku.mod" 1;
Reading model section from examples/sudoku.mod...
examples/sudoku.mod:69: warning: data section ignored
69 lines were read
0
>
```

5.2.2.3 Read and translate data section Synopsis:

```
glp::mpl_read_data tranobject filename
```

Parameters:

tranobject: pointer to the MathProg translator object

filename: file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::mpl_read_data mpt "examples/sudoku.dat";
Reading data section from examples/sudoku.dat...
16 lines were read
0
>
```

5.2.2.4 Generate the model Synopsis:

```
glp::mpl_generate tranobject filename
```

Parameters:

tranobject: pointer to the MathProg translator object

filename: file name

Returns:

0 if generating went OK; non-zero in case of an error

Example:

```
> glp::mpl_generate mpt "examples/sudoku.lst";
Generating fa...
Generating fb...
Generating fc...
Generating fd...
Generating fe...
Model has been successfully generated
0
>
```

5.2.2.5 Build problem instance from the model Synopsis:

```
glp::mpl_build_prob tranobject lp
```

Parameters:

tranobject: pointer to the MathProg translator object

lp: pointer to the LP problem object

Returns:

```
()
```

Example:

```
> glp::mpl_build_prob mpt lp;
()
>
```

5.2.2.6 Postsolve the model Synopsis:

```
glp::mpl_postsolve tran lp solution
```

Parameters:

tranobject: pointer to the MathProg translator object

lp: pointer to the LP problem object

solution: one of the following:

glp::sol: use the basic solution

glp::ipt: use the interior-point solution

glp::mip: use mixed integer solution

Returns:

0 if postsolve went OK; non-zero in case of an error

Example:

```
> glp::mpl_postsolve mpt lp glp::sol;
Model has been successfully processed
0
>
```

5.2.2.7 Delete the MathProg translator object Synopsis:

```
glp::mpl_free_wksp tranobject
```

Parameters:

tranobject: pointer to the MathProg translator object

Returns:

```
()
```

Example:

```
> glp::mpl_free_wksp mpt;
()
>
```

5.2.3 Problem solution reading/writing routines

5.2.3.1 Write basic solution in printable format Synopsis:

```
glp::print_sol lp filename
```

Parameters:

lp: pointer to the LP problem object
filename: file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::print_sol lp "examples/test.txt";
Writing basic solution to 'examples/test.txt'...
0
>
```

5.2.3.2 Read basic solution from a text file Synopsis:

```
glp::read_sol lp filename
```

Parameters:

lp: pointer to the LP problem object
filename: file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_sol lp "examples/test.txt";
Reading basic solution from 'examples/test.txt'...
1235 lines were read
0
>
```

5.2.3.3 Write basic solution into a text file Synopsis:

```
glp::write_sol lp filename
```

Parameters:

lp: pointer to the LP problem object
filename: file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_sol lp "examples/test.txt";
Writing basic solution to 'examples/test.txt'...
1235 lines were written
0
>
```

5.2.3.4 Print sensitivity analysis report Synopsis:

```
glp::print_ranges lp indices filename
```

Parameters:

lp: pointer to the LP problem object

indices: list indices k of of rows and columns to be included in the report. If 1 $\leq$ k $\leq$ m, the basic variable is k-th auxiliary variable, and if m + 1 $\leq$ k $\leq$ m + n, the non-basic variable is (k $-$ m)-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object. An empty lists means printing report for all rows and columns.

filename: file name

Returns:

0: if the operation was successful

non-zero: if the operation failed

Example:

```
> glp::print_ranges lp [] "sensitivity.rpt";
Write sensitivity analysis report to 'sensitivity.rpt'...
0
>
```

5.2.3.5 Write interior-point solution in printable format Synopsis:

```
glp::print_ipt lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::print_ipt lp "examples/test.txt";
Writing interior-point solution to 'examples/test.txt'...
0
>
```

5.2.3.6 Read interior-point solution from a text file Synopsis:

```
glp::read_ipt lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_ipt lp "examples/test.txt";
Reading interior-point solution from ‘examples/test.txt’...
1235 lines were read
0
>
```

5.2.3.7 Write interior-point solution into a text file Synopsis:

```
glp::write_ipt lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_ipt lp "examples/test.txt";
Writing interior-point solution to ‘examples/test.txt’...
1235 lines were written
0
>
```

5.2.3.8 Write MIP solution in printable format Synopsis:

```
glp::print_mip lp filename
```

Parameters:

lp: pointer to the LP problem object

filename: file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::print_mip lp "examples/test.txt";
Writing MIP solution to ‘examples/test.txt’...
0
>
```

5.2.3.9 Read MIP solution from a text file Synopsis:

```
glp::read_mip lp filename
```

Parameters:

lp: pointer to the LP problem object
filename: file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_mip lp "examples/test.txt";
Reading MIP solution from ‘examples/test.txt’...
1235 lines were read
0
>
```

5.2.3.10 Write MIP solution into a text file Synopsis:

```
glp::write_mip lp filename
```

Parameters:

lp: pointer to the LP problem object
filename: file name

Returns:

0 if writing went OK; non-zero in case of an error

Example:

```
> glp::write_mip lp "examples/test.txt";
Writing MIP solution to ‘examples/test.txt’...
1235 lines were written
0
>
```

5.3 Advanced API routines

5.3.1 LP basis routines

5.3.1.1 Check whether basis factorization exists Synopsis:

```
glp::bf_exists lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

non-zero: the basis factorization exists and can be used for calculations
0: the basis factorization does not exist

Example:

```
> glp::bf:exists lp;
1
>
```

5.3.1.2 Compute the basis factorization Synopsis:

```
glp::factorize lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

one of the following:

glp::ok: 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::exond: the basis matrix is ill-conditioned, i.e. its condition number is too large

Example:

```
> glp::factorize lp;
glp::ok
>
```

5.3.1.3 Check whether basis factorization has been updated Synopsis:

```
glp::bf_updated lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

0: if the basis factorization has been just computed from "scratch"

non-zero: if the factorization has been updated at least once

Example:

```
> glp::bf_updated lp;
0
>
```

5.3.1.4 Get basis factorization parameters Synopsis:

```
glp::get_bfcp lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

complete list of options in a form of tuples (option_name, value):

glp::type: basis factorization type:

glp::bf_ft: LU + Forrest–Tomlin update

glp::bf_bg: LU + Schur complement + Bartels–Golub update

glp::bf_gr: LU + Schur complement + Givens rotation update

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

glp::piv_tol: threshold pivoting (Markowitz) tolerance, $0 < \text{piv\_tol} < 1$, used on computing LU-factorization of the basis matrix

glp::piv_lim: 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, $\text{piv\_lim} \geq 1$

glp::suhl: this parameter is used on computing LU-factorization of the basis matrix

glp::on: enables applying the heuristic proposed by Uwe Suhl

glp::off: disables this heuristic

glp::eps_tol: epsilon tolerance, $\text{eps\_tol} \geq 0$, used on computing LU-factorization of the basis matrix

glp::max_gro: maximal growth of elements of factor U, $\text{max\_gro} \geq 1$, allowable on computing LU-factorization of the basis matrix

glp::nfs_max: maximal number of additional row-like factors (entries of the eta file), $\text{nfs\_max} \geq 1$, which can be added to LU-factorization of the basis matrix on updating it with the Forrest–Tomlin technique

glp::upd_tol: update tolerance, $0 < \text{upd\_tol} < 1$, used on updating LU-factorization of the basis matrix with the Forrest–Tomlin technique

glp::nrs_max: maximal number of additional rows and columns, $\text{nrs\_max} \geq 1$, which can be added to LU-factorization of the basis matrix on updating it with the Schur complement technique

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

Example:

```
> glp::get_bfcp lp;
[(glp::type,glp::bf_ft),(glp::lu_size,0),(glp::piv_tol,0.1),(glp::piv_lim,4),
(glp::suhl,glp::on),(glp::eps_tol,1e-15),(glp::max_gro,10000000000.0),
(glp::nfs_max,50),(glp::upd_tol,1e-06),(glp::nrs_max,50),(glp::rs_size,0)]
>
```

5.3.1.5 Change basis factorization parameters

Synopsis:

```
glp::set_bfcp lp options
```

Parameters:

lp: pointer to the LP problem object

options: list of options in a form of tuples (option_name, value):

glp::type: (default: glp::bf_ft) - basis factorization type:

glp::bf_ft: LU + Forrest–Tomlin update

glp::bf_bg: LU + Schur complement + Bartels–Golub update

glp::bf_gr: LU + Schur complement + Givens rotation update

glp::lu_size: (default: 0) - the initial size of the Sparse Vector Area, in non-zeros, used on computing LU-factorization of the basis matrix for the first time - if this parameter is set to 0, the initial SVA size is determined automatically

glp::piv_tol: (default: 0.10) - threshold pivoting (Markowitz) tolerance, $0 <$ piv_tol < 1, used on computing LU-factorization of the basis matrix.

glp::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

glp::suhl: (default: glp::on) - this parameter is used on computing LU-factorization of the basis matrix.

glp::on: enables applying the heuristic proposed by Uwe Suhl

glp::off: disables this heuristic

glp::eps_tol: (default: 1e-15) - epsilon tolerance, eps_tol ≥ 0, used on computing LU -factorization of the basis matrix.

glp::max_gro: (default: 1e+10) - maximal growth of elements of factor U, max_gro ≥ 1, allowable on computing LU-factorization of the basis matrix.

glp::nfs_max: (default: 50) - maximal number of additional row-like factors (entries of the eta file), nfs_max ≥ 1, which can be added to LU-factorization of the basis matrix on updating it with the Forrest–Tomlin technique.

glp::upd_tol: (default: 1e-6) - update tolerance, $0 <$ upd_tol < 1, used on updating LU -factorization of the basis matrix with the Forrest–Tomlin technique.

glp::nrs_max: (default: 50) - maximal number of additional rows and columns, nrs_max ≥ 1, which can be added to LU-factorization of the basis matrix on updating it with the Schur complement technique.

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

Remarks:

Options not mentioned in the option list are left unchanged.

All options will be reset to their default values when an empty option list is supplied.

Returns:

`()` if all options are OK, otherwise returns a list of bad options

Example:

```
> glp_set_bfcp lp [(glp::type, glp::bf_ft), (glp::piv_tol, 0.15)];
()
>
```

5.3.1.6 Retrieve the basis header information Synopsis:

```
glp::get_bhead lp k
```

Parameters:

lp: pointer to the LP problem object

k: variable index in the basis matrix

Returns:

If basic variable (xB)k , $1 \leq k \leq m$, is i-th auxiliary variable ($1 \leq i \leq m$), the routine returns i. Otherwise, if (xB)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.

Example:

```
> glp::get_bhead lp 3;
5
>
```

5.3.1.7 Retrieve row index in the basis header Synopsis:

```
glp::get_row_bind lp rowindex
```

Parameters:

lp: pointer to the LP problem object

rowindex: row index

Returns:

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

Example:

```
> glp::get_row_bind lp 3;
1
>
```

5.3.1.8 Retrieve column index in the basis header Synopsis:

```
glp::get_col_bind lp colindex
```

Parameters:

lp: pointer to the LP problem object

colindex: column index

Returns:

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

Example:

```
> glp::get_col_bind lp 2;
3
>
```

5.3.1.9 Perform forward transformation Synopsis:

```
glp::ftran lp vector
```

Parameters:

lp: pointer to the LP problem object

vector: vector to be transformed - a dense vector in a form of a list of double numbers has to be supplied and the number of its members must exactly correspond to the number of LP problem constraints

Returns:

the transformed vector in the same format

Example:

```
> glp::ftran lp [1.5, 3.2, 4.8];
[1.8,0.466666666666667,-1.96666666666667]
>
```

5.3.1.10 Perform backward transformation Synopsis:

```
glp::btran lp vector
```

Parameters:

lp: pointer to the LP problem object

vector: vector to be transformed - a dense vector in a form of a list of double numbers has to be supplied and the number of its members must exactly correspond to the number of LP problem constraints

Returns:

the transformed vector in the same format

Example:

```
> glp::btran lp [1.5, 3.2, 4.8];
[-8.86666666666667,0.266666666666667,1.5]
>
```

5.3.1.11 Warm up LP basis Synopsis:

```
glp::warm_up lp
```

Parameters:

lp: pointer to the LP problem object

Returns:

one of the following:

glp::ok: the LP basis has been successfully "warmed up"

glp::ebadb: the LP basis is invalid, because the number of basic variables is not the same as the number of rows

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

Example:

```
> glp::warm_up lp;
glp::e_ok
>
```

5.3.2 Simplex tableau routines

5.3.2.1 Compute row of the tableau Synopsis:

```
glp::eval_tab_row lp k
```

Parameters:

lp: pointer to the LP problem object

k: variable index such that it corresponds to some basic variable: if $1 \leq k \leq m$, the basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

simplex tableau row in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::eval_tab_row lp 3;
[(1,2.0),(6,4.0)]
>
```

5.3.2.2 Compute column of the tableau Synopsis:

```
glp::eval_tab_col lp k
```

Parameters:

lp: pointer to the LP problem object

k: variable index such that it corresponds to some non-basic variable: if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

simplex tableau column in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::eval_tab_col lp 1;
[(3,2.0),(4,-0.666666666666667),(5,1.66666666666667)]
>
```

5.3.2.3 Transform explicitly specified row Synopsis:

```
glp::transform_row lp rowvector
```

Parameters:

lp: pointer to the LP problem object

rowvector: row vector to be transformed in a sparse form as a list of tuples (k, value): if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

the transformed row in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::transform_row lp [(1, 3.0), (2, 3.5)];
[(1,3.83333333333333),(2,-0.0833333333333333),(6,-3.41666666666667)]
>
```

5.3.2.4 Transform explicitly specified column Synopsis:

```
glp::transform_col lp colvector
```

Parameters:

lp: pointer to the LP problem object

colvector: column vector to be transformed in a sparse form as a list of tuples (k, value): if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist)

Returns:

the transformed column in a sparse form as a list of tuples (index, value), where index has the same meaning as k in parameters

Example:

```
> glp::transform_col lp [(2, 1.0), (3, 2.3)];
[(3,2.3),(4,-0.166666666666667),(5,0.166666666666667)]
>
```

5.3.2.5 Perform primal ratio test Synopsis:

```
glp::prim_rtest lp colvector dir eps
```

Parameters:

lp: pointer to the LP problem object

colvector: simplex tableau column in a sparse form as a list of tuples (k, value): if $1 \leq k \leq m$, the basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the primal solution must be feasible)

dir: specifies in which direction the variable y changes on entering the basis: $+1$ means increasing, -1 means decreasing

eps: relative tolerance (small positive number) used to skip small values in the column

Returns:

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

Example:

```
> glp::prim_rtest lp [(3, 2.5), (5, 7.0)] 1 1.0e-5;
3
>
```

5.3.2.6 Perform dual ratio test Synopsis:

```
glp::dual_rtest lp rowvector dir eps
```

Parameters:

lp: pointer to the LP problem object

rowvector: simplex tableau row in a sparse form as a list of tuples (k, value): if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the dual solution must be feasible)

dir: specifies in which direction the variable y changes on leaving the basis: $+1$ means increasing, -1 means decreasing

eps: relative tolerance (small positive number) used to skip small values in the row

Returns:

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

Example:

```
> glp::dual_rtest lp [(1, 1.5), (6, 4.0)] 1 1.0e-5;
6
>
```

5.3.2.7 Analyze active bound of non-basic variable Synopsis:

```
glp::analyze_bound lp k
```

Parameters:

lp: pointer to the LP problem object

k: if $1 \leq k \leq m$, the non-basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the solution must be optimal)

Returns:

The routine returns a tuple (limit1, var1, limit2 var2) where:

value1: 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: 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: 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: 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.

Example:

```
> analyze_bound lp 2;
1995.06864446899,12,2014.03478832467,4
>
```

5.3.2.8 Analyze objective coefficient at basic variable Synopsis:

```
glp::analyze_coef lp k
```

Parameters:

lp: pointer to the LP problem object

k: if $1 \leq k \leq m$, the basic variable is k-th auxiliary variable, and if $m + 1 \leq k \leq m + n$, the non-basic variable is $(k - m)$-th structural variable, where m is the number of rows and n is the number of columns in the specified problem object (the basis factorization must exist and the solution must be optimal)

Returns:

The routine returns a tuple (coef1, var1, value1, coef2 var2, value2) where:

coef1: 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: value of the basic variable being analyzed in an adjacent basis, which is defined as follows. Let the objective coefficient reaches its minimal value (coef1) and continues 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: 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: 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: 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.

Example:

```
> analyze_coef lp 1;
-1.0,3,306.771624713959,1.79769313486232e+308,0,296.216606498195
>
```

5.4 Branch-and-cut API routines

All branch-and-cut API routines are supposed to be called from the callback routine. They cannot be called directly.

5.4.1 Basic routines

5.4.1.1 Determine reason for calling the callback routine Synopsis:

```
glp::ios_reason tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

one of the following:

glp::irowgen: request for row generation
glp::ibingo: better integer solution found
glp::iheur: request for heuristic solution
glp::icutgen: request for cut generation
glp::ibranch: request for branching
glp::iselect: request for subproblem selection
glp::iprepro: request for preprocessing

Example:

```
glp::ios:reason tree;
```

5.4.1.2 Access the problem object Synopsis:

```
glp::ios_get_prob tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

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

Example:

```
glp::ios_get_prob tree;
```

5.4.1.3 Determine additional row attributes Synopsis:

```
glp::ios_row_attr tree rowindex
```

Parameters:

tree: pointer to the branch-and-cut search tree
rowindex: row index

Returns:

The routine returns a tuple consisting of three values (level, origin, klass):

level: subproblem level at which the row was created
origin: the row origin flag - one of the following:
glp::rf_reg: regular constraint
glp::rf_lazy: "lazy" constraint
glp::rf_cut: cutting plane constraint
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

Example:

```
glp::ios_row_attr tree 3;
```

5.4.1.4 Compute relative MIP gap Synopsis:

```
glp::ios_mip_gap tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

The routine returns the relative MIP gap.

Example:

```
> glp::ios_mip_gap tree;
```

5.4.1.5 Access application-specific data Synopsis:

```
glp::ios_node_data tree node
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

The routine glp_ios_node_data returns a pointer to the memory block for the specified subproblem. Note that if cb_size = 0 was specified in the call of the *intopt* function, the routine returns a null pointer.

Example:

```
> glp::ios_node_data tree 23;
```

5.4.1.6 Select subproblem to continue the search Synopsis:

```
glp::ios_select_node tree node
```

Parameters:

tree: pointer to the branch-and-cut search tree

node: reference number of the subproblem from which the search will continue

Returns:

()

Example:

```
> glp::ios_select_node tree 23;
```

5.4.1.7 Provide solution found by heuristic Synopsis:

```
glp::ios_heur_sol tree colvector
```

Parameters:

tree: pointer to the branch-and-cut search tree

colvector: solution found by a primal heuristic. Primal values of all variables (columns) found by the heuristic should be placed in the list, i. e. the list must contain n numbers where n is the number of columns in the original problem object. Note that the routine does not check primal feasibility of the solution provided.

Returns:

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

Example:

```
> glp::ios_heur_sol tree [15.7, (-3.1), 2.2];
```

5.4.1.8 Check whether can branch upon specified variable Synopsis:

```
glp::ios_can_branch tree j
```

Parameters:

tree: pointer to the branch-and-cut search tree

j: variable (column) index

Returns:

The function returns non-zero if j-th variable can be used for branching. Otherwise, it returns zero.

Example:

```
> glp::ios_can_branch tree 23;
```

5.4.1.9 Choose variable to branch upon Synopsis:

```
glp::ios_branch_upon tree j selection
```

Parameters:

tree: pointer to the branch-and-cut search tree

j: ordinal number of the selected branching variable

selection: one of the following:

glp::dn_brnch: select down-branch
glp::up_brnch: select up-branch
glp::no_brnch: use general selection technique

Returns:

()

Example:

```
> glp::ios_branch_upon tree 23 glp::up_brnch;
```

5.4.1.10 Terminate the solution process Synopsis:

```
glp::ios_terminate tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

()

Example:

```
> glp::ios_terminate tree;
```

5.4.2 The search tree exploring routines

5.4.2.1 Determine the search tree size Synopsis:

```
glp::ios_tree_size tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

The routine returns a tuple (a_cnt, n_cnt, t_cnt), where

a_cnt: the current number of active nodes

n_cnt: the current number of all (active and inactive) nodes

t_cnt: 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.

Example:

```
> glp::ios_tree_size tree;
```

5.4.2.2 Determine current active subproblem Synopsis:

```
glp::ios_curr_node tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

The routine returns the reference number of the current active subproblem. If the current subproblem does not exist, the routine returns zero.

Example:

```
> glp::ios_curr_node tree;
```

5.4.2.3 Determine next active subproblem Synopsis:

```
glp::ios_next_node tree node
```

Parameters:

tree: pointer to the branch-and-cut search tree

node: reference number of an active subproblem or zero

Returns:

If the parameter p is zero, the routine returns the reference number of the first active subproblem. 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. 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.

Example:

```
> glp::ios_next_node tree 23;
```

5.4.2.4 Determine previous active subproblem Synopsis:

```
glp::ios_prev_node tree node
```

Parameters:

tree: pointer to the branch-and-cut search tree

node: reference number of an active subproblem or zero

Returns:

If the parameter p is zero, the routine returns the reference number of the last active subproblem. 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. 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.

Example:

```
> glp::ios_prev_node tree 23;
```

5.4.2.5 Determine parent active subproblem Synopsis:

```
glp::ios_up_node tree node
```

Parameters:

tree: pointer to the branch-and-cut search tree

node: reference number of an active or inactive subproblem

Returns:

The routine returns the reference number of its parent subproblem. If the specified subproblem is the root of the tree, the routine returns zero.

Example:

```
> glp::ios_up_node tree 23;
```

5.4.2.6 Determine subproblem level Synopsis:

```
glp::ios_node_level tree node
```

Parameters:

tree: pointer to the branch-and-cut search tree

node: reference number of an active or inactive subproblem

Returns:

The routine returns the level of the given subproblem in the branch-and-bound tree. (The root subproblem has level 0.)

Example:

```
> glp::ios_node_level tree 23;
```

5.4.2.7 Determine subproblem local bound Synopsis:

```
glp::ios_node_bound tree node
```

Parameters:

tree: pointer to the branch-and-cut search tree

node: reference number of an active or inactive subproblem

Returns:

The routine returns the local bound for the given subproblem.

Example:

```
> glp::ios_node_bound tree 23;
```

5.4.2.8 Find active subproblem with the best local bound Synopsis:

```
glp::ios_best_node tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

The routine 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). If the tree is empty, the routine returns zero.

Example:

```
> glp::ios_best_node tree;
```

5.4.3 The cut pool routines

5.4.3.1 Determine current size of the cut pool Synopsis:

```
glp::ios_pool_size tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

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

Example:

```
> glp::ios_pool_size tree;
```

5.4.3.2 Add constraint to the cut pool Synopsis:

```
glp::ios_add_row tree (name, klass, flags, row, type, rhs)
```

Parameters:

tree: pointer to the branch-and-cut search tree

name: symbolic name of the constraint

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.

flags: currently is not used and must be zero

row: list of pairs (colindex, coefficient)

type: one of the following:

glp::lo: $\sum$(aj.xj) $\geq$ RHS constraint
glp::up: $\sum$(aj.xj) $\leq$ RHS constraint

rhs: right hand side of the constraint

Returns:

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

Example:

```
> glp::ios_add_row tree ("new_constraint", 101, 0,
                         [(3, 15.0), (4, 6.7), (8, 1.25)], glp::up, 152.7);
```

5.4.3.3 Remove constraint from the cut pool Synopsis:

```
glp::ios_del_row tree rowindex
```

Parameters:

tree: pointer to the branch-and-cut search tree

rowindex: index of row to be deleted from the cut pool

Returns:

()

Remark:

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.

Example:

```
> glp::ios_del_row tree 5;
```

5.4.3.4 Remove all constraints from the cut pool Synopsis:

```
glp::ios_clear_pool tree
```

Parameters:

tree: pointer to the branch-and-cut search tree

Returns:

()

Example:

```
> glp::ios_clear_pool tree;
```

5.5 Graph and network API routines

5.5.1 Basic graph routines

5.5.1.1 Create the GLPK graph object Synopsis:

```
glp::create_graph v_size a_size
```

Parameters:

v_size: size of vertex data blocks, in bytes, $0 \leq$ v size ≤ 256

a_size: size of arc data blocks, in bytes, $0 \leq$ a size ≤ 256.

Returns:

The routine returns a pointer to the graph created.

Example:

```
> let g = glp::create_graph 32 64;
> g;
#<pointer 0x9de7168>
>
```

5.5.1.2 Set the graph name Synopsis:

```
glp::set_graph_name graph name
```

Parameters:

graph: pointer to the graph object

name: the graph name, an empty string erases the current name

Returns:

()

Example:

```
> glp::set_graph_name graph "MyGraph";
()
>
```

5.5.1.3 Add vertices to a graph Synopsis:

```
glp::add_vertices graph count
```

Parameters:

graph: pointer to the graph object

count: number of vertices to add

Returns:

The routine returns the ordinal number of the first new vertex added to the graph.

Example:

```
> glp::add_vertices graph 5;
18
>
```

5.5.1.4 Add arc to a graph Synopsis:

```
glp::add_arc graph i j
```

Parameters:

graph: pointer to the graph object

i: index of the tail vertex

j: index of the head vertex

Returns:

()

Example:

```
> glp::add_arc graph 7 12;
()
>
```

5.5.1.5 Erase content of the GLPK graph object Synopsis:

```
glp::erase_graph graph v_size a_size
```

Parameters:

graph: pointer to the graph object
v_size: size of vertex data blocks, in bytes, $0 \leq$ v size ≤ 256
a_size: size of arc data blocks, in bytes, $0 \leq$ a size ≤ 256.

Returns:

()

Remark:

The routine reinitialises the graph object. Its efect is equivalent to calling delete_graph followed by a call to create_graph.

Example:

```
> glp::erase_graph graph 16 34;
()
>
```

5.5.1.6 Delete the GLPK graph object Synopsis:

```
glp::delete_graph graph
```

Parameters:

graph: pointer to the graph object

Returns:

()

Remark:

The routine destroys the graph object and invalidates the pointer. This is done automatically when the graph is not needed anymore, the routine need not be usually called.

Example:

```
> glp::delete_graph graph
()
>
```

5.5.1.7 Read graph in a plain text format Synopsis:

```
glp::read_graph graph filename
```

Parameters:

graph: pointer to the graph object
filename: file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_graph graph "graph_data.txt";
0
>
```

5.5.1.8 Write graph in a plain text format Synopsis:

```
glp::write_graph graph filename
```

Parameters:

graph: pointer to the graph object

filename: file name

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::write_graph graph "graph_data.txt";
0
>
```

5.5.2 Graph analysis routines

5.5.2.1 Find all weakly connected components of a graph Synopsis:

```
glp::weak_comp graph v_num
```

Parameters:

graph: pointer to the graph object

v_num: offset of the field of type int in the vertex data block, to which the routine stores the number of a weakly connected component containing that vertex - if v_num < 0, no component numbers are stored

Returns:

The routine returns the total number of components found.

Example:

```
> glp::weak_comp graph 16;
3
>
```

5.5.2.2 Find all strongly connected components of a graph Synopsis:

```
glp::strong_comp graph v_num
```

Parameters:

graph: pointer to the graph object

v_num: offset of the field of type int in the vertex data block, to which the routine stores the number of a strongly connected component containing that vertex - if v_num < 0, no component numbers are stored

Returns:
The routine returns the total number of components found.
Example:

```
> glp::strong_comp graph 16;
4
>
```

5.5.3 Minimum cost flow problem

5.5.3.1 Read minimum cost flow problem data in DIMACS format Synopsis:

```
glp::read_mincost graph v_rhs a_low a_cap a_cost filename
```

Parameters:

graph: pointer to the graph object

v_rhs: offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, the value is not stored

a_low: offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, the lower bound is not stored

a_cap: offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0, the upper bound is not stored

a_cost: offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, the cost is not stored

fname: the name of a text file to be read in - if the file name name ends with the suffix '.gz', the file is assumed to be compressed, in which case the routine decompresses it "on the fly"

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::read_mincost graph 0 8 16 24 "graphdata.txt";
0
>
```

5.5.3.2 Write minimum cost flow problem data in DIMACS format Synopsis:

```
glp::write_mincost graph v_rhs a_low a_cap a_cost fname
```

Parameters:

graph: pointer to the graph object

v_rhs: offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, the value is not stored

a_low: offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, the lower bound is not stored

a_cap: offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0, the upper bound is not stored

a_cost: offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, the cost is not stored

fname: the name of a text file to be written out - if the file name name ends with the suffix '.gz', the file is assumed to be compressed, in which case the routine compresses it "on the fly"

Returns:

0 if reading went OK; non-zero in case of an error

Example:

```
> glp::write_mincost graph 0 8 16 24 "graphdata.txt";
0
>
```

5.5.3.3 Convert minimum cost flow problem to LP Synopsis:

```
glp::mincost_lp lp graph names v_rhs a_low a_cap a_cost
```

Parameters:

lp: pointer to the LP problem object

graph: pointer to the graph object

names: one of the following:

glp::on: assign symbolic names of the graph object components to symbolic names of the LP problem object components

glp::off: no symbolic names are assigned

v_rhs: offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi $= 0$ for all nodes

a_low: offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, it is assumed lij $=$ 0 for all arcs

a_cap: offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij $= 1$ for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost: offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij $= 0$ for all arcs

Returns:

()

Example:

```
> glp::mincost_lp lp graph glp::on 0 8 16 24;
()
>
```

5.5.3.4 Solve minimum cost flow problem with out-of-kilter algorithm **Synopsis**:

```
glp::mincost_okalg graph v_rhs a_low a_cap a_cost a_x v_pi
```

Parameters:

graph: pointer to the graph object

v_rhs: offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi $= 0$ for all nodes

a_low: offset of the field of type double in the arc data block, to which the routine stores lij, the lower bound to the arc flow - if a_low < 0, it is assumed lij $=$ 0 for all arcs

a_cap: offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij $= 1$ for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost: offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij $= 0$ for all arcs

a_x: offset of the field of type double in the arc data block, to which the routine stores xij, the arc flow found - if a_x < 0, the arc flow value is not stored

v_pi: specifies an offset of the field of type double in the vertex data block, to which the routine stores pi, the node potential, which is the Lagrange multiplier for the corresponding flow conservation equality constraint

Remark:

Note that all solution components (the objective value, arc flows, and node potentials) computed by the routine are always integer-valued.

Returns:

The function returns a tuple in the form **(code, obj)**, where **code** is one of the following

glp::ok: optimal solution found

glp::enopfs: no (primal) feasible solution exists

glp::edata: unable to start the search, because some problem data are either not integer-valued or out of range; this code is also returned if the total supply, which is the sum of bi over all source nodes (nodes with bi > 0), exceeds INT_MAX

glp::erange: the search was prematurely terminated because of integer overflow

glp::efail: an error has been detected in the program logic - if this code is returned for your problem instance, please report to <bug-glpk@gnu.org>

and **obj** is value of the objective function.

Example:

```
> glp::mincost_okalg graph 0 8 16 24 32 40;
(glp::ok, 15)
>
```

5.5.3.5 Klingman's network problem generator Synopsis:

```
glp::netgen graph v_rhs a_cap a_cost parameters
```

Parameters:

graph: pointer to the graph object

v_rhs: offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi $= 0$ for all nodes

a_cap: offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij $= 1$ for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost: offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij $= 0$ for all arcs

parameters: tuple of exactly 15 integer numbers with the following meaning:

parm[1]: iseed 8-digit positive random number seed

parm[2]: nprob 8-digit problem id number

parm[3]: nodes total number of nodes

parm[4]: nsorc total number of source nodes (including transshipment nodes)

parm[5]: nsink total number of sink nodes (including transshipment nodes)

parm[6]: iarcs number of arc

parm[7]: mincst minimum cost for arcs

parm[8]: maxcst maximum cost for arcs

parm[9]: itsup total supply

parm[10]: ntsorc number of transshipment source nodes

parm[11]: ntsink number of transshipment sink nodes

parm[12]: iphic percentage of skeleton arcs to be given the maximum cost

parm[13]: ipcap percentage of arcs to be capacitated

parm[14]: mincap minimum upper bound for capacitated arcs

parm[15]: maxcap maximum upper bound for capacitated arcs

Returns:

0 if the instance was successfully generated, nonzero otherwise

Example:

```
> glp::netgen graph 0 8 16 (12345678, 87654321, 20, 12, 8,
                            25, 5, 20, 300, 6, 5, 15, 100, 1, 30);
0
>
```

5.5.3.6 Grid-like network problem generator Synopsis:

```
glp::gridgen graph v_rhs a_cap a_cost parameters
```

Parameters:

graph: pointer to the graph object

v_rhs: offset of the field of type double in the vertex data block, to which the routine stores bi, the supply/demand value - if v_rhs < 0, it is assumed bi $= 0$ for all nodes

a_cap: offset of the field of type double in the arc data block, to which the routine stores uij, the upper bound to the arc flow (the arc capacity) - if a_cap < 0,it is assumed uij $= 1$ for all arcs, value of DBL_MAX means an uncapacitated arc

a_cost: offset of the field of type double in the arc data block, to which the routine stores cij, the per-unit cost of the arc flow - if a_cost < 0, it is assumed cij $= 0$ for all arcs

parameters: tuple of exactly 14 integer numbers with the following meaning:

parm[1]: two-ways arcs indicator:
- **1:** if links in both direction should be generated
- **0:** otherwise

parm[2]: random number seed (a positive integer)

parm[3]: number of nodes (the number of nodes generated might be slightly different to make the network a grid)

parm[4]: grid width

parm[5]: number of sources

parm[6]: number of sinks

parm[7]: average degree

parm[8]: total flow

parm[9]: distribution of arc costs:
- **1:** uniform
- **2:** exponential

parm[10]: lower bound for arc cost (uniform), 100 lambda¸ (exponential)

parm[11]: upper bound for arc cost (uniform), not used (exponential)

parm[12]: distribution of arc capacities:
- **1:** uniform
- **2:** exponential

parm[13]: lower bound for arc capacity (uniform), 100 lambda (exponential)

parm[14]: upper bound for arc capacity (uniform), not used (exponential)

Returns:

0 if the instance was successfully generated, nonzero otherwise

Example:

```
> glp::gridgen graph 0 8 16 (1, 123, 20, 4, 7, 5, 3, 300, 1, 1, 5, 1, 5, 30);
0
>
```

5.5.4 Maximum flow problem

5.5.4.1 Read maximum cost flow problem data in DIMACS format Synopsis:

```
glp::read_maxflow graph a_cap filename
```

Parameters:

graph: pointer to the graph object

Returns:
Example:

```
>
```

5.5.4.2 Write maximum cost flow problem data in DIMACS format Synopsis:

```
glp::write_maxflow graph s t a_cap filename
```

Parameters:

graph: pointer to the graph object

Returns:
Example:

```
>
```

5.5.4.3 Convert maximum flow problem to LP Synopsis:

```
glp::maxflow_lp lp graph names s t a_cap
```

Parameters:

graph: pointer to the graph object

Returns:
Example:

```
>
```

5.5.4.4 Solve maximum flow problem with Ford-Fulkerson algorithm Synopsis:

```
glp::maxflow_ffalg graph s t a_cap a_x v_cut
```

Parameters:

graph: pointer to the graph object

Returns:
Example:

```
>
```

5.5.4.5 Goldfarb's maximum flow problem generator Synopsis:

```
glp::rmfgen graph a_cap parameters
```

Parameters:

graph: pointer to the graph object

Returns:
Example:

```
>
```

5.6 Miscellaneous routines

5.6.1 Library environment routines

5.6.1.1 Determine library version Synopsis:

```
‘glp::version
```

Parameters:

none

Returns:

GLPK library version

Example:

```
> glp::version;
"4.38"
>
```

5.6.1.2 Enable/disable terminal output Synopsis:

```
glp::term_out switch
```

Parameters:

switch: one of the following:

glp::on: enable terminal output from GLPK routines
glp::off: disable terminal output from GLPK routines

Returns:

```
()
```

Example:

```
> glp::term_out glp:off;
()
>
```

5.6.1.3 Enable/disable the terminal hook routine Synopsis:

```
glp::term_hook switch info
```

Parameters:

switch: one of the following:

glp::on: use the terminal callback function

glp::off: don't use the terminal callback function

info: pointer to a memory block which can be used for passing additional information to the terminal callback function

Returns:

```
()
```

Example:

```
> glp::term_hook glp::on NULL;
()
>
```

5.6.1.4 Get memory usage information Synopsis:

```
glp::mem_usage
```

Parameters:

none

Returns:

tuple consisting of four numbers:

- `count` (int) - the number of currently allocated memory blocks
- `cpeak` (int) - the peak value of `count` reached since the initialization of the GLPK library environment
- `total` (bigint) - the total amount, in bytes, of currently allocated memory blocks
- `tpeak` (bigint) - the peak value of `total` reached since the initialization of the GLPK library envirionment

Example:

```
> glp::mem_usage;
7,84,10172L,45304L
>
```

5.6.1.5 Set memory usage limit Synopsis:

```
glp::mem_limit limit
```

Parameters:

limit: memory limit in megabytes

Returns:

```
()
```

Example:

```
> glp::mem_limit 200;
()
>
```

5.6.1.6 Free GLPK library environment Synopsis:

`glp::free_env`

Parameters:

none

Returns:

`()`

Example:

```
> glp_free_env;
()
>
```