GLPK.jl Documentation

Release 0.2

Carlo Baldassi

October 14, 2013

CONTENTS

This module provides a wrapper for the GNU Linear Programming Kit (GLPK), which is a C library, in Julia. It is designed for making it easy to port C code to Julia, while at the same time having the benefits of the higher level language features of Julia, like the automatic management of memory, the possibility of returning tuples/strings/vectors etc.

It's currently based on GLPK version 4.52.

At the moment, only Unix platforms (Linux, OS X, BDS's etc) are fully supported. On Windows, you may be able to use this package if you install manually the GLPK library in your system.

CHAPTER

ONE

INSTALLATION

The module can be installed via Julia's package manager:

```
julia> Pkg.add("GLPK")
```

The package manager will try to find the correct version of the GLPK library in the system; if it doesn't find it, it will install it.

On Linux and BSD, this means that it will download the source files and compile the library.

On OS X, this means that it will need the Homebrew package, and it will download a precompiled binary.

On Windows, no automated installation step is currently available; you'll need to install the library manually before adding the package.

CHAPTER

TWO

PREAMBLE

Almost all GLPK functions can be called in Julia with basically the same syntax as in the original C library, with some simple translation rules (with very *few exceptions*). Some functionality is still missing (see *this list*); most of it will be added in the future.

Let's start with an example. This is an excerpt from the beginning of the `sample.c` example program which ships with GLPK:

```
/* C code */
glp_prob *lp = glp_create_prob();
glp_set_prob_name(lp, "sample");
glp_set_obj_dir(lp, GLP_MAX);
glp_add_rows(lp, 3);
glp_set_row_name(lp, 1, "p");
glp_set_row_bnds(lp, 1, GLP_UP, 0.0, 100.0);
```

This is the Julia translation of the above:

```
# Julia code
lp = GLPK.Prob()
GLPK.set_prob_name(lp, "sample")
GLPK.set_obj_dir(lp, GLPK.MAX)
GLPK.add_rows(lp, 3)
GLPK.set_row_name(lp, 1, "p")
GLPK.set_row_bnds(lp, 1, GLPK.UP, 0.0, 100.0)
```

Apart from the first line, which is different, the translation of subsequent lines follows the very simple rule that function names and constants drop the prefixes `glp_` and `GLP_`, and take the `GLPK` module prefix instead (at the moment, constants are integer values, like in C, but this may change in the future). Note that, as with all Julia modules, the `GLPK` prefix could be omitted by adding a `using GLPK` line in the code, but this is not advised in this case due to the very high number of functions with relatively common names in the library.

Because of the strict adherence of the Julia functions to their C counterparts, and since the GLPK documentation is extremely well written and complete, this manual page is not going to document the whole GLPK library in detail, but rather provide *the rules* needed to translate from C to Julia, detail the *few exceptions* to these rules and then *list all the available functions* with a brief description of their usage.

Please, refer to the original GLPK manual (available at http://www.gnu.org/software/glpk) for a detailed description of the library API.

CHAPTER

THREE

GLPK TRANSLATION RULES FROM C TO JULIA

3.1 1) functions and constants drop their prefix

Almost all functions in the C library start with the prefix `glp_`, and all constants start with the prefix `GLP_`. These prefixes are dropped in Julia, and the module prefix `GLPK.` is used instead. For example, the function `glp_simplex` becomes `GLPK.simplex`, and the constant `GLP_UP` becomes `GLPK.UP`.

3.2 2) from C structs to Julia objects

All structs in the original GLPK are wrapped up in composite types, which initialize and destroy themselves as needed. For example, the `glp_prob` C struct becomes the `GLPK.Prob` Julia type. Whenever in C you would pass a pointer to a struct, in Julia you pass a corresponding composite object. This is the table relating C structs with Julia types:

C	Julia
`glp_prob`	`GLPK.Prob`
`glp_smcp`	`GLPK.SimplexParam`
`glp_iptcp`	`GLPK.InteriorParam`
`glp_iocp`	`GLPK.IntoptParam`
`glp_bfcp`	`GLPK.BasisFactParam`
`glp_tran`	`GLPK.MathProgWorkspace`
`glp_attr`	`GLPK.Attr`

Therefore, the original C GLPK API:

```
int glp_simplex(glp_prob * lp, glp_smpc * param)
```

becomes:

```
GLPK.simplex(lp::GLPK.Prob, param::GLPL.SimplexParam)
```

In the C GLPK API, objects are created by functions, such as:

```
glp_prob * lp = glp_create_prob();
glp_smcp * param = glp_smcp_init();
```

and need to be destroyed when the program is finished:

```
glp_delete_prob(lp);
glp_smcp_delete(smcp);
```

In Julia, objects are created by calling the object constructor (without parameters):

```
lp = GLPK.Prob()
param = GLPK.SimplexParam()
```

and they are automatically destroyed by the garbage collector when no longer needed.

3.3 3) setting the parameters to the solvers

In all GLPK solver functions, like `glp_simplex`, options are passed via structs. As stated before, these become composite object types in Julia, and no special syntax is required to access them. In C:

```
param = glp_smcp_init();
param.msg_lev = GLP_MSG_ERR;
param.presolve = GLP_ON;
```

In Julia:

```
param = GLPK.SimplexParam()
param.msg_lev = GLPK.MSG_ERR
param.presolve = GLPK.ON
```

As a special case, since *type* is a reserved word in Julia, the *type* field of *glp_bfcp* has been renamed to *bftype*:

```
bf_opts = GLPK.BasisFactParam()
bf_opts.bftype = ...
```

Additionally, parameters can be accessed via an array-like referencing syntax:

```
param = GLPK.SimplexParam()
param["msg_lev"]= GLPK.MSG_ERR
param["presolve"] = GLPK.ON
```

Note that the field names are passed as strings, and that all GLPK constants are available in Julia. Also note that no test is currently performed at assignment to check that the provided values are valid, but this may change in the future.

(This part of the API may change in the future.)

3.4 4) scalar and array types translate in a natural way

The following C-to-Julia type conversion rules apply:

C	Julia
`int`	`Cint`
`double`	`Cdouble`
`char[]`	`String`

On output, these rules apply exactly. On input, on the other hand, Julia requirements are more relaxed:

C	Julia
`int`	`Integer`
`double`	`Real`

Whenever the C version expects a pointer to an array, a Julia Array can be passed. In the GLPK API, all indexing starts from 1 even in the C version, so no special care is required on that side (in C, you would leave an unused element at the beginning of each array; in Julia you don't).

The relaxed requirements for inputs are also valid for arrays (e.g. one can pass an `Array{Int64}` when an array of `int` is expected, and it will be converted automatically). The only exception is for functions which return an array of values by filling out an allocated array whose pointer is provided by the user. In that case, the strict version of the rules applies (i.e. you can only pass an `Array{Cint}` if an array of `int` is expected). Those functions almost always have an alternative, more convenient formulation as well, though.

3.5 5) optional arguments

Whenever the C version accepts the value `NULL` to indicate an optional pointer argument, the Julia version accepts the constant `nothing`. In case the optional pointer argument is an array, an empty array is also accepted (it can be of the expected type, e.g. `Cint[]`, or even just `[]`) Most of the time, alternative ways to call the function are also provided.

3.6 6) fatal errors become exceptions

Whenever an invalid condition is detected (e.g. if you pass an invalid parameter, such as a negative length), the Julia GLPK wrapper throws a `GLPK.GLPKError` exception with some message detailing what went wrong. With the default settings, all invalid input combinations should be captured by Julia before being passed over to the library, so that all errors could be catched via a `try ... catch` block; in practice, it is likely that some conditions exist which will leak to the C API: this should be considered as a bug (and reported as such).

This behaviour can be modified, leaving to the C library to do the checking, by calling:

```
GLPK.jl_set_preemptive_check(false)
```

In this case, if an error is catched within the C library, Julia will throw a `GLPK.GLPKFatalError` exception. When this happens, all GLPK-related objects which were created up to that point become invalid and cannot be used any more.

The status of the preemptive check can be obtained by:

```
GLPK.jl_get_preemptive_check()
```

(With the default settings, this returns `true`.) The validity of an object can be checked by:

```
GLPK.jl_obj_is_valid(object)
```

CHAPTER

FOUR

GLPK FUNCTIONS WHICH ARE NOT AVALIABLE YET IN JULIA

There are 2 groups of functions which are not wrapped:

1. All graph and network routines (anything involving `glp_graph` objects); these will be added in the future
2. Some misc functions which either have a variable argument list, or involve callbacks, or are implemented as mcaros (see section 6.1 in the GLPK manual):
   - `glp_printf`
   - `glp_vprintf`
   - `glp_term_hook`
   - `glp_error`
   - `glp_assert`
   - `glp_error_hook`

CHAPTER

FIVE

FUNCTIONS WHICH DIFFER FROM THEIR C COUNTERPARTS

Some library functions return multiple values; as C cannot do this directly, this is obtained via some "pointer gymnastics". In Julia, on the other hand, this is not necessary, and providing an exact counterpart to the C version would be awkward and pointless. There are 5 such functions:

- `GLPK.analyze_bound`
- `GLPK.analyze_coef`
- `GLPK.mem_usage`
- `GLPK.ios_tree_size`
- `GLPK.check_kkt`

For example the C declaration for `glp_analyze_bound` is:

```
void glp_analyze_bound(glp_prob *lp, int k, int *limit1, int *var1, int *limit2, int *var2)
```

In Julia, this becomes:

```
GLPK.analyze_bound(glp_prob::GLPK.Prob, k::Integer)
```

which returns a tuple:

```
julia> (limit1, var1, limit2, var2) = GLPK.analyze_bound(glp_prob, k)
```

The other 4 functions work in the same way, by just returning the values which in C you would pass as pointers.

Some other functions have both a strictly-compatible calling form, for simplifying C code porting, and some more convenient Julia counterparts. See *the list below* for more details.

One function has a different return value: `GLPK.version` returns a tuple of integers with the major and minor version numbers, rather then a string.

CHAPTER

SIX

LIST OF GLPK FUNCTIONS IN JULIA

As stated above, this list only offers a brief explanation of what each function does and presents alternative calling forms when available. Refer to the GLPK manual for a complete description.

set_prob_name(*glp_prob*, *name*)
Assigns a name to the problem object (or deletes it if `name` is empty or `nothing`).

set_obj_name(*glp_prob*, *name*)
Assigns a name to the objective function (or deletes it if `name` is empty or `nothing`).

set_obj_dir(*glp_prob*, *dir*)
Sets the optimization direction, `GLPK.MIN` (minimization) or `GLPK.MAX` (maximization).

add_rows(*glp_prob*, *rows*)
Adds the given number of rows (constraints) to the problem object; returns the number of the first new row added.

add_cols(*glp_prob*, *cols*)
Adds the given number of columns (structural variables) to the problem object; returns the number of the first new column added.

set_row_name(*glp_prob*, *row*, *name*)
Assigns a name to the specified row (or deletes it if `name` is empty or `nothing`).

set_col_name(*glp_prob*, *col*, *name*)
Assigns a name to the specified column (or deletes it if `name` is empty or `nothing`).

set_row_bnds(*glp_prob*, *row*, *bounds_type*, *lb*, *ub*)
Sets the type and bounds on a row. `type` must be one of `GLPK.FR` (free), `GLPK.LO` (lower bounded), `GLPK.UP` (upper bounded), `GLPK.DB` (double bounded), `GLPK.FX` (fixed).

At initialization, each row is free.

set_col_bnds(*glp_prob*, *col*, *bounds_type*, *lb*, *ub*)
Sets the type and bounds on a column. `type` must be one of `GLPK.FR` (free), `GLPK.LO` (lower bounded), `GLPK.UP` (upper bounded), `GLPK.DB` (double bounded), `GLPK.FX` (fixed).

At initialization, each column is fixed at 0.

set_obj_coef(*glp_prob*, *col*, *coef*)
Sets the objective coefficient to a column (`col` can be 0 to indicate the constant term of the objective function).

set_mat_row(*glp_prob*, *row*[, *len*], *ind*, *val*)
Sets (replaces) the content of a row. The content is specified in sparse format: `ind` is a vector of indices, `val` is the vector of corresponding values. `len` is the number of vector elements which will be considered, and must be less or equal to the length of both `ind` and `val`. If `len` is 0, `ind` and/or `val` can be `nothing`.

In Julia, `len` can be omitted, and then it is inferred from `ind` and `val` (which need to have the same length in such case).

set_mat_col (*glp_prob*, *col*[, *len*], *ind*, *val*)
Sets (replaces) the content of a column. Everything else is like `set_mat_row`.

load_matrix (*glp_prob*[, *numel*], *ia*, *ja*, *ar*)
load_matrix (*glp_prob*, *A*)
Sets (replaces) the content matrix (i.e. sets all rows/coluns at once). The matrix is passed in sparse format.

In the first form (original C API), it's passed via 3 vectors: `ia` and `ja` are for rows/columns indices, `ar` is for values. `numel` is the number of elements which will be read and must be less or equal to the length of any of the 3 vectors. If `numel` is 0, any of the vectors can be passed as `nothing`.

In Julia, `numel` can be omitted, and then it is inferred from `ia`, `ja` and `ar` (which need to have the same length in such case).

Also, in Julia there's a second, simpler calling form, in which the matrix is passed as a `SparseMatrixCSC` object.

check_dup (*rows*, *cols*[, *numel*], *ia*, *ja*)
Check for duplicates in the indices vectors `ia` and `ja`. `numel` has the same meaning and (optional) use as in `load_matrix`. Returns 0 if no duplicates/out-of-range indices are found, or a positive number indicating where a duplicate occurs, or a negative number indicating an out-of-bounds index.

sort_matrix (*glp_prob*)
Sorts the elements of the problem object's matrix.

del_rows (*glp_prob*[, *num_rows*], *rows_ids*)
Deletes rows from the problem object. Rows are specified in the `rows_ids` vector. `num_rows` is the number of elements of `rows_ids` which will be considered, and must be less or equal to the length id `rows_ids`. If `num_rows` is 0, `rows_ids` can be `nothing`. In Julia, `num_rows` is optional (it's inferred from `rows_ids` if not given).

del_cols (*glp_prob*, *cols_ids*)
Deletes columns from the problem object. See `del_rows`.

copy_prob (*glp_prob_dest*, *glp_prob*, *copy_names*)
Makes a copy of the problem object. The flag `copy_names` determines if names are copied, and must be either `GLPK.ON` or `GLPK.OFF`.

erase_prob (*glp_prob*)
Resets the problem object.

get_prob_name (*glp_prob*)
Returns the problem object's name. Unlike the C version, if the problem has no assigned name, returns an empty string.

get_obj_name (*glp_prob*)
Returns the objective function's name. Unlike the C version, if the objective has no assigned name, returns an empty string.

get_obj_dir (*glp_prob*)
Returns the optimization direction, `GLPK.MIN` (minimization) or `GLPK.MAX` (maximization).

get_num_rows (*glp_prob*)
Returns the current number of rows.

get_num_cols (*glp_prob*)
Returns the current number of columns.

get_row_name(*glp_prob*, *row*)
Returns the name of the specified row. Unlike the C version, if the row has no assigned name, returns an empty string.

get_col_name(*glp_prob*, *col*)
Returns the name of the specified column. Unlike the C version, if the column has no assigned name, returns an empty string.

get_row_type(*glp_prob*, *row*)
Returns the type of the specified row: `GLPK.FR` (free), `GLPK.LO` (lower bounded), `GLPK.UP` (upper bounded), `GLPK.DB` (double bounded), `GLPK.FX` (fixed).

get_row_lb(*glp_prob*, *row*)
Returns the lower bound of the specified row, `-DBL_MAX` if unbounded.

get_row_ub(*glp_prob*, *row*)
Returns the upper bound of the specified row, `+DBL_MAX` if unbounded.

get_col_type(*glp_prob*, *col*)
Returns the type of the specified column: `GLPK.FR` (free), `GLPK.LO` (lower bounded), `GLPK.UP` (upper bounded), `GLPK.DB` (double bounded), `GLPK.FX` (fixed).

get_col_lb(*glp_prob*, *col*)
Returns the lower bound of the specified column, `-DBL_MAX` if unbounded.

get_col_ub(*glp_prob*, *col*)
Returns the upper bound of the specified column, `+DBL_MAX` if unbounded.

get_obj_coef(*glp_prob*, *col*)
Return the objective coefficient to a column (`col` can be 0 to indicate the constant term of the objective function).

get_num_nz(*glp_prob*)
Return the number of non-zero elements in the constraint matrix.

get_mat_row(*glp_prob*, *row*, *ind*, *val*)
get_mat_row(*glp_prob*, *row*)
Returns the contents of a row. In the first form (original C API), it fills the `ind` and `val` vectors provided, which must be of type `Vector{Int32}` and `Vector{Float64}` respectively, and have a sufficient length to hold the result (or they can be empty or `nothing`, and then they're not filled). It returns the length of the result.

In Julia, there's a second, simpler calling form which allocates and returns the two vectors as `(ind, val)`.

get_mat_col(*glp_prob*, *col*, *ind*, *val*)
get_mat_col(*glp_prob*, *col*)
Returns the contents of a column. See `get_mat_row`.

create_index(*glp_prob*)
Creates the name index (used by `find_row`, `find_col`) for the problem object.

find_row(*glp_prob*, *name*)
Finds the numeric id of a row by name. Returns 0 if no row with the given name is found.

find_col(*glp_prob*, *name*)
Finds the numeric id of a column by name. Returns 0 if no column with the given name is found.

delete_index(*glp_prob*)
Deletes the name index for the problem object.

set_rii(*glp_prob*, *row*, *rii*)
Sets the rii scale factor for the specified row.

set_sjj(*glp_prob*, *col*, *sjj*)
Sets the sjj scale factor for the specified column.

get_rii(*glp_prob*, *row*)
Returns the rii scale factor for the specified row.

get_sjj(*glp_prob*, *col*)
Returns the sjj scale factor for the specified column.

scale_prob(*glp_prob*, *flags*)
Performs automatic scaling of problem data for the problem object. The parameter `flags` can be `GLPK.SF_AUTO` (automatic) or a bitwise OR of the forllowing: `GLPK.SF_GM` (geometric mean), `GLPK.SF_EQ` (equilibration), `GLPK.SF_2N` (nearest power of 2), `GLPK.SF_SKIP` (skip if well scaled).

unscale_prob(*glp_prob*)
Unscale the problem data (cancels the scaling effect).

set_row_stat(*glp_prob*, *row*, *stat*)
Sets the status of the specified row. `stat` must be one of: `GLPK.BS` (basic), `GLPK.NL` (non-basic lower bounded), `GLPK.NU` (non-basic upper-bounded), `GLPK.NF` (non-basic free), `GLPK.NS` (non-basic fixed).

set_col_stat(*glp_prob*, *col*, *stat*)
Sets the status of the specified column. `stat` must be one of: `GLPK.BS` (basic), `GLPK.NL` (non-basic lower bounded), `GLPK.NU` (non-basic upper-bounded), `GLPK.NF` (non-basic free), `GLPK.NS` (non-basic fixed).

std_basis(*glp_prob*)
Constructs the standard (trivial) initial LP basis for the problem object.

adv_basis(*glp_prob*[, *flags*])
Constructs an advanced initial LP basis for the problem object. The flag `flags` is optional; it must be 0 if given.

cpx_basis(*glp_prob*)
Constructs an initial LP basis for the problem object with the algorithm proposed by R. Bixby.

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

The parameters are specified via the optional `glp_param` argument, which is of type `GLPK.SimplexParam` (or `nothing` to use the default settings).

Returns 0 in case of success, or a non-zero flag specifying the reason for failure: `GLPK.EBADB` (invalid base), `GLPK.ESING` (singular matrix), `GLPK.ECOND` (ill-conditioned matrix), `GLPK.EBOUND` (incorrect bounds), `GLPK.EFAIL` (solver failure), `GLPK.EOBJLL` (lower limit reached), `GLPK.EOBJUL` (upper limit reached), `GLPK.ITLIM` (iterations limit exceeded), `GLPK.ETLIM` (time limit exceeded), `GLPK.ENOPFS` (no primal feasible solution), `GLPK.ENODFS` (no dual feasible solution).

exact(*glp_prob*[, *glp_param*])
A tentative implementation of the primal two-phase simplex method based on exact (rational) arithmetic. Similar to `simplex`. The optional `glp_param` is of type `GLPK.SimplexParam`.

The possible return values are `0` (success) or `GLPK.EBADB`, `GLPK.ESING`, `GLPK.EBOUND`, `GLPK.EFAIL`, `GLPK.ITLIM`, `GLPK.ETLIM` (see `simplex()`).

init_smcp(*glp_param*)
Initializes a `GLPK.SimplexParam` object with the default values. In Julia, this is done at object creation time; this function can be used to reset the object.

get_status(*glp_prob*)
Returns the generic status of the current basic solution: `GLPK.OPT` (optimal), `GLPK.FEAS` (feasible),

GLPK.INFEAS (infeasible), GLPK.NOFEAS (no feasible solution), GLPK.UNBND (unbounded solution), GLPK.UNDEF (undefined).

get_prim_stat(*glp_prob*)
Returns the status of the primal basic solution: GLPK.FEAS, GLPK.INFEAS, GLPK.NOFEAS, GLPK.UNDEF (see get_status()).

get_dual_stat(*glp_prob*)
Returns the status of the dual basic solution: GLPK.FEAS, GLPK.INFEAS, GLPK.NOFEAS, GLPK.UNDEF (see get_status()).

get_obj_val(*glp_prob*)
Returns the current value of the objective function.

get_row_stat(*glp_prob*, *row*)
Returns the status of the specified row: GLPK.BS, GLPK.NL, GLPK.NU, GLPK.NF, GLPK.NS (see set_row_stat()).

get_row_prim(*glp_prob*, *row*)
Returns the primal value of the specified row.

get_row_dual(*glp_prob*, *row*)
Returns the dual value (reduced cost) of the specified row.

get_col_stat(*glp_prob*, *col*)
Returns the status of the specified column: GLPK.BS, GLPK.NL, GLPK.NU, GLPK.NF, GLPK.NS (see set_row_stat()).

get_col_prim(*glp_prob*, *col*)
Returns the primal value of the specified column.

get_col_dual(*glp_prob*, *col*)
Returns the dual value (reduced cost) of the specified column.

get_unbnd_ray(*glp_prob*)
Returns the number k of a variable, which causes primal or dual unboundedness (if 1 <= k <= rows it's row k; if rows+1 <= k <= rows+cols it's column k-rows, if k=0 such variable is not defined).

interior(*glp_prob*[, *glp_param*])
The routine interior is a driver to the LP solver based on the primal-dual interior-point method. This routine retrieves problem data from the specified problem object, calls the solver to solve the problem instance, and stores results of computations back into the problem object.

The parameters are specified via the optional glp_param argument, which is of type GLPK.InteriorParam (or nothing to use the default settings).

Returns 0 in case of success, or a non-zero flag specifying the reason for failure: GLPK.EFAIL (solver failure), GLPK.ENOCVG (very slow convergence, or divergence), GLPK.ITLIM (iterations limit exceeded), GLPK.EINSTAB (numerical instability).

init_iptcp(*glp_param*)
Initializes a GLPK.InteriorParam object with the default values. In Julia, this is done at object creation time; this function can be used to reset the object.

ipt_status(*glp_prob*)
Returns the status of the interior-point solution: GLPK.OPT (optimal), GLPK.INFEAS (infeasible), GLPK.NOFEAS (no feasible solution), GLPK.UNDEF (undefined).

ipt_obj_val(*glp_prob*)
Returns the current value of the objective function for the interior-point solution.

ipt_row_prim(*glp_prob*, *row*)
Returns the primal value of the specified row for the interior-point solution.

ipt_row_dual(*glp_prob*, *row*)
Returns the dual value (reduced cost) of the specified row for the interior-point solution.

ipt_col_prim(*glp_prob*, *col*)
Returns the primal value of the specified column for the interior-point solution.

ipt_col_dual(*glp_prob*, *col*)
Returns the dual value (reduced cost) of the specified column for the interior-point solution.

set_col_kind(*glp_prob*, *col*, *kind*)
Sets the kind for the specified column (for mixed-integer programming). `kind` must be one of: `GLPK.CV` (continuous), `GLPK.IV` (integer), `GLPK.BV` (binary, 0/1).

get_col_kind(*glp_prob*, *col*)
Returns the kind for the specified column (see `set_col_kind()`).

get_num_int(*glp_prob*)
Returns the number of columns marked as integer (including binary).

get_num_bin(*glp_prob*)
Returns the number of columns marked binary.

intopt(*glp_prob*[, *glp_param*])
The routine `intopt` is a driver to the mixed-integer-programming (MIP) solver based on the branch- and-cut method, which is a hybrid of branch-and-bound and cutting plane methods.

The parameters are specified via the optional `glp_param` argument, which is of type `GLPK.IntoptParam` (or `nothing` to use the default settings).

Returns 0 in case of success, or a non-zero flag specifying the reason for failure: `GLPK.EBOUND` (incorrect bounds), `GLPK.EROOT` (no optimal LP basis given), `GLPK.ENOPFS` (no primal feasible LP solution), `GLPK.ENODFS` (no dual feasible LP solution), `GLPK.EFAIL` (solver failure), `GLPK.EMIPGAP` (mip gap tolearance reached), `GLPK.ETLIM` (time limit exceeded), `GLPK.ESTOP` (terminated by application).

init_iocp(*glp_param*)
Initializes a `GLPK.IntoptParam` object with the default values. In Julia, this is done at object creation time; this function can be used to reset the object.

mip_status(*glp_prob*)
Returns the generic status of the MIP solution: `GLPK.OPT` (optimal), `GLPK.FEAS` (feasible), `GLPK.NOFEAS` (no feasible solution), `GLPK.UNDEF` (undefined).

mip_obj_val(*glp_prob*)
Returns the current value of the objective function for the MIP solution.

mip_row_val(*glp_prob*, *row*)
Returns the value of the specified row for the MIP solution.

mip_col_val(*glp_prob*, *col*)
Returns the value of the specified column for the MIP solution.

check_kkt(*glp_prob*, *sol*, *cond*)
Checks feasibility/optimality conditions for the current solution stored in the given problem. `sol` specifies what solution should be checked: either `GLPK.SOL` (basic), `GLPK.IPT` (interior point) or `GLPK.MIP` (mixed integer). `cond` specifies which condition should be checked: either `GLPK.KKT_PE` (primal equality), `GLPK.KKT_PB` (primal bound), `GLPK.KKT_DE` (dual equality, interior point only) or `GLPK.KKT_DB` (dual bound, interior point only).

In Julia, this function has a different API then C. It returns `(ae_max, ae_ind, re_max, re_ind)` rather then taking them as pointers in the argument list.

The meaning of the returned parameters is as follows: `ae_max` (largest absolute error), `ae_ind` (index of the above), `re_max` (largest relative error) and `re_ind` (index of the above). The indices refer to a row, column or variable depending on the value of `cond` (`KKT_PE`, `KKT_DE` or `KKT_*B`, respectively).

read_mps(*glp_prob*, *format*[, *param*], *filename*)
Reads problem data in MPS format from a text file. `format` must be one of `GLPK.MPS_DECK` (fixed, old) or `GLPK.MPS_FILE` (free, modern). `param` is optional; if given it must be `nothing`.

Returns 0 upon success; throws an error in case of failure.

write_mps(*glp_prob*, *format*[, *param*], *filename*)
Writes problem data in MPS format from a text file. See `read_mps`.

Returns 0 upon success; throws an error in case of failure.

read_lp(*glp_prob*[, *param*], *filename*)
Reads problem data in CPLEX LP format from a text file. `param` is optional; if given it must be `nothing`.

Returns 0 upon success; throws an error in case of failure.

write_lp(*glp_prob*[, *param*], *filename*)
Writes problem data in CPLEX LP format from a text file. See `read_lp`.

Returns 0 upon success; throws an error in case of failure.

read_prob(*glp_prob*[, *flags*], *filename*)
Reads problem data in GLPK LP/MIP format from a text file. `flags` is optional; if given it must be 0.

Returns 0 upon success; throws an error in case of failure.

write_prob(*glp_prob*[, *flags*], *filename*)
Writes problem data in GLPK LP/MIP format from a text file. See `read_prob`.

Returns 0 upon success; throws an error in case of failure.

mpl_read_model(*glp_tran*, *filename*, *skip*)
Reads the model section and, optionally, the data section, from a text file in MathProg format, and stores it in `glp_tran`, which is a `GLPK.MathProgWorkspace` object. If `skip` is nonzero, the data section is skipped if present.

Returns 0 upon success; throws an error in case of failure.

mpl_read_data(*glp_tran*, *filename*)
Reads data section from a text file in MathProg format and stores it in `glp_tran`, which is a `GLPK.MathProgWorkspace` object. May be called more than once.

Returns 0 upon success; throws an error in case of failure.

mpl_generate(*glp_tran*[, *filename*])
Generates the model using its description stored in the `GLPK.MathProgWorkspace` translator workspace `glp_tran`. The optional `filename` specifies an output file; if not given or `nothing`, the terminal is used.

Returns 0 upon success; throws an error in case of failure.

mpl_build_prob(*glp_tran*, *glp_prob*)
Transfer information from the `GLPK.MathProgWorkspace` translator workspace `glp_tran` to the `GLPK.Prob` problem object `glp_prob`.

mpl_postsolve(*glp_tran*, *glp_prob*, *sol*)
Copies the solution from the `GLPK.Prob` problem object `glp_prob` to the `GLPK.MathProgWorkspace`

translator workspace `glp_tran` and then executes all the remaining model statements, which follow the solve statement.

The parameter `sol` specifies which solution should be copied from the problem object to the workspace: `GLPK.SOL` (basic), `GLPK.IPT` (interior-point), `GLPK.MIP` (MIP).

Returns 0 upon success; throws an error in case of failure.

print_sol(*glp_prob*, *filename*)
Writes the current basic solution to a text file, in printable format.

Returns 0 upon success; throws an error in case of failure.

read_sol(*glp_prob*, *filename*)
Reads the current basic solution from a text file, in the format used by `write_sol`.

Returns 0 upon success; throws an error in case of failure.

write_sol(*glp_prob*, *filename*)
Writes the current basic solution from a text file, in a format which can be read by `read_sol`.

Returns 0 upon success; throws an error in case of failure.

print_ipt(*glp_prob*, *filename*)
Writes the current interior-point solution to a text file, in printable format.

Returns 0 upon success; throws an error in case of failure.

read_ipt(*glp_prob*, *filename*)
Reads the current interior-point solution from a text file, in the format used by `write_ipt`.

Returns 0 upon success; throws an error in case of failure.

write_ipt(*glp_prob*, *filename*)
Writes the current interior-point solution from a text file, in a format which can be read by `read_ipt`.

Returns 0 upon success; throws an error in case of failure.

print_mip(*glp_prob*, *filename*)
Writes the current MIP solution to a text file, in printable format.

Returns 0 upon success; throws an error in case of failure.

read_mip(*glp_prob*, *filename*)
Reads the current MIP solution from a text file, in the format used by `write_mip`.

Returns 0 upon success; throws an error in case of failure.

write_mip(*glp_prob*, *filename*)
Writes the current MIP solution from a text file, in a format which can be read by `read_mip`.

Returns 0 upon success; throws an error in case of failure.

print_ranges(*glp_prob, [[len,] list,] [flags,] filename*)
Performs sensitivity analysis of current optimal basic solution and writes the analysis report in human-readable format to a text file. `list` is a vector specifying the rows/columns to analyze (if 1 <= list[i] <= rows, analyzes row list[i]; if rows+1 <= list[i] <= rows+cols, analyzes column list[i]-rows). `len` is the number of elements of `list` which will be consideres, and must be smaller or equal to the length of the list. In Julia, `len` is optional (it's inferred from `len` if not given). `list` can be empty of `nothing` or not given at all, implying all indices will be analyzed. `flags` is optional, and must be 0 if given.

To call this function, the current basic solution must be optimal, and the basis factorization must exist.

Returns 0 upon success, non-zero otherwise.

bf_exists(*glp_prob*)
Returns non-zero if the basis fatorization for the current basis exists, 0 otherwise.

factorize(*glp_prob*)
Computes the basis factorization for the current basis.

Returns 0 if successful, otherwise: `GLPK.EBADB` (invalid matrix), `GLPK.ESING` (singluar matrix), `GLPK.ECOND` (ill-conditioned matrix).

bf_updated(*glp_prob*)
Returns 0 if the basis factorization was computed from scratch, non-zero otherwise.

get_bfcp(*glp_prob*, *glp_param*)
Retrieves control parameters, which are used on computing and updating the basis factorization associated with the problem object, and stores them in the `GLPK.BasisFactParam` object `glp_param`.

set_bfcp(*glp_prob*[, *glp_param*])
Sets the control parameters stored in the `GLPK.BasisFactParam` object `glp_param` into the problem object. If `glp_param` is `nothing` or is omitted, resets the parameters to their defaults.

The `glp_param` should always be retreived via `get_bfcp` before changing its values and calling this function.

get_bhead(*glp_prob*, *k*)
Returns the basis header information for the current basis. `k` is a row index.

Returns either i such that 1 <= i <= rows, if `k` corresponds to i-th auxiliary variable, or rows+j such that 1 <= j <= columns, if `k` corresponds to the j-th structural variable.

get_row_bind(*glp_prob*, *row*)
Returns the index of the basic variable `k` which is associated with the specified row, or `0` if the variable is non-basic. If `GLPK.get_bhead(glp_prob, k) == row`, then `GLPK.get_bind(glp_prob, row) = k`.

get_col_bind(*glp_prob*, *col*)
Returns the index of the basic variable `k` which is associated with the specified column, or `0` if the variable is non-basic. If `GLPK.get_bhead(glp_prob, k) == rows+col`, then `GLPK.get_bind(glp_prob, col) = k`.

ftran(*glp_prob*, *v*)
Performs forward transformation (FTRAN), i.e. it solves the system Bx = b, where B is the basis matrix, x is the vector of unknowns to be computed, b is the vector of right-hand sides. At input, `v` represents the vector b; at output, it contains the vector x. `v` must be a `Vector{Float64}` whose length is the number of rows.

btran(*glp_prob*, *v*)
Performs backward transformation (BTRAN), i.e. it solves the system `B'x = b`, where `B` is the transposed of the basis matrix, `x` is the vector of unknowns to be computed, `b` is the vector of right-hand sides. At input, `v` represents the vector `b`; at output, it contains the vector `x`. `v` must be a `Vector{Float64}` whose length is the number of rows.

warm_up(*glp_prob*)
"Warms up" the LP basis using current statuses assigned to rows and columns, i.e. computes factorization of the basis matrix (if it does not exist), computes primal and dual components of basic solution, and determines the solution status.

Returns 0 if successful, otherwise: `GLPK.EBADB` (invalid matrix), `GLPK.ESING` (singluar matrix), `GLPK.ECOND` (ill-conditioned matrix).

eval_tab_row(*glp_prob*, *k*, *ind*, *val*)
eval_tab_row(*glp_prob*, *k*)
Computes a row of the current simplex tableau which corresponds to some basic variable specified by the

parameter k. If 1 <= k <= rows, uses k-th auxiliary variable; if rows+1 <= k <= rows+cols, uses (k-rows)-th structural variable. The basis factorization must exist.

In the first form, stores the result in the provided vectors ind and val, which must be of type Vector{Int32} and Vector{Float64}, respectively, and returns the length of the outcome; in Julia, the vectors will be resized as needed to hold the result.

In the second, simpler form, ind and val are returned in a tuple as the output of the function.

eval_tab_col(*glp_prob*, *k*, *ind*, *val*)
eval_tab_col(*glp_prob*, *k*)

Computes a column of the current simplex tableau which corresponds to some non-basic variable specified by the parameter k. See eval_tab_row.

transform_row(*glp_prob*[, *len*], *ind*, *val*)

Performs the same operation as eval_tab_row with the exception that the row to be transformed is specified explicitly as a sparse vector. The parameter len is the number of elements of ind and val which will be used, and must be smaller or equal to the length of both vectors; in Julia it is optional (and the ind and val must have the same length). The vectors int and val must be of type Vector{Int32} and Vector{Float64}, respectively, since they will also hold the result; in Julia, they will be resized to the resulting required length.

Returns the length if the resulting vectors ind and val.

transform_col(*glp_prob*[, *len*], *ind*, *val*)

Performs the same operation as eval_tab_col with the exception that the row to be transformed is specified explicitly as a sparse vector. See transform_row.

prim_rtest(*glp_prob*[, *len*], *ind*, *val*, *dir*, *eps*)

Performs the primal ratio test using an explicitly specified column of the simplex table. The current basic solution must be primal feasible. The column is specified in sparse format by len (length of the vector), ind and val (indices and values of the vector). len is the number of elements which will be considered and must be smaller or equal to the length of both ind and val; in Julia, it can be omitted (and then ind and val must have the same length). The indices in ind must be between 1 and rows+cols; they must correspond to basic variables. dir is a direction parameter which must be either +1 (increasing) or -1 (decreasing). eps is a tolerance parameter and must be positive. See the GLPK manual for a detailed explanation.

Returns the position in ind and val which corresponds to the pivot element, or 0 if the choice cannot be made.

dual_rtest(*glp_prob*[, *len*], *ind*, *val*, *dir*, *eps*)

Performs the dual ratio test using an explicitly specified row of the simplex table. The current basic solution must be dual feasible. The indices in ind must correspond to non-basic variables. Everything else is like in prim_rtest.

analyze_bound(*glp_prob*, *k*)

Analyzes the effect of varying the active bound of specified non-basic variable. See the GLPK manual for a detailed explanation. In Julia, this function has a different API then C. It returns (limit1, var1, limit2, var2) rather then taking them as pointers in the argument list.

analyze_coef(*glp_prob*, *k*)

Analyzes the effect of varying the objective coefficient at specified basic variable. See the GLPK manual for a detailed explanation. In Julia, this function has a different API then C. It returns (coef1, var1, value1, coef2, var2, value2) rather then taking them as pointers in the argument list.

ios_reason(*tree*)

(To be used from inside a callback passed via the cb_func field of a GLPK.IntoptParam object. tree is a Ptr{Void} which must be the same obtained by the callback.)

Returns a code which indicates why the callback is being called. Possible return values are: GLPK.ISELECT, GLPK.IPREPRO, GLPK.IROWGEN, GLPK.IHEUR, GLPK.ICUTGEN, GLPK.IBRANCH and GLPK.BINGO.

ios_get_prob(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns a *GLPK.Prob* object used by the MIP solver. It is not the same object as the original, although it will represent the same problem (i.e. wrap the same C structure) if the presolver was not used.

ios_row_attr(*tree*, *row*[, *attr*])
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Retrieves additional attributes of the given `row` for the current subproblem, storing it in a `GLPK.Attr` object (the object will be created and returned if not passed).

ios_mip_gap(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Computes the relative MIP gap (also called duality gap).

ios_node_data(*tree*, *p*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Retuns the memory block allocated for the subproblem whose reference number is `p`.

ios_select_node(*tree*, *p*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Used to select an active subproblem with reference number `p` in response to the reason `GLPK.ISELECT`.

ios_heur_sol(*tree*, *x*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Used to provide an integer feasible solution `x` in response to the reason `GLPK.IHEUR`.

ios_can_branch(*tree*, *col*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns non-zero if the given column can be branched upon, zero otherwise.

ios_branch_upon(*tree*, *col*, *sel*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Used to choose a branching variable (`col`) in response to the reason `GLPK.IBRANCH`. `sel` is a flag which must take a value from `GLPK.DN_BRANCH`, `GLPK.UP_BRANCH`, `GLPK.NO_BRANCH`.

ios_terminate(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Terminates the search.

ios_tree_size(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns counts which characterize the size of the search tree. In Julia, this function has a different API then C. It returns `(a_cnt, n_cnt, t_cnt)` rather then taking them as pointers in the argument list.

ios_curr_node(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the reference number of the current subproblem, or zero if the current subproblem does not exist.

ios_next_node(*tree*, *p*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the reference number of the active subproblem next to `p`, or the first one if `p` is zero, or zero if no such subproblem exists.

ios_prev_node(*tree*, *p*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the reference number of the active subproblem previous to `p`, or the last one if `p` is zero, or zero if no such subproblem exists.

ios_up_node(*tree*, *p*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the reference number of the parent subproblem of `p`, or zero if `p` is the root.

ios_node_level(*tree*, *p*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the level of the subproblem `p`.

ios_node_bound(*tree*, *p*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the local bound for the subproblem `p`.

ios_best_node(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the reference number of the node with the best local bound, or zero if the tree is empty.

ios_pool_size(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Returns the current size of the cut pool.

ios_add_row(*tree*[, *name*], *klass*[, *flags*[, *len*]], *ind*, *val*, *constr_type*, *rhs*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Adds a row (cutting plane constant) to the cut pool. `name` is a string which can be assigned to the constraint and can be also be `nothing` (meaning the empty string). `klass` specifies the constraint class and can be either `0` or an integer between `101` and `200`. `flags` must be `0`.

The constraint is specified from the left hand side (`len`, `ind` and `val`), the constraint type (`constr_type`) and the right hand side (`rhs`). The left hand side is a vector whose content is specified in sparse format: `ind` is a vector of indices, `val` is the vector of corresponding values. `len` is the number of vector elements which will be considered, and must be less or equal to the length of both `ind` and `val`. `constr_type` must be either `GLPK.LO` or `GLPK.UP`. `rhs` is a scalar real number.

In Julia, some arguments are optional: `len`, which if omitted is inferred from `ind` and `val` (which need to have the same length in such case); `flags` which defaults to `0`; `name` which defaults to `nothing`.

ios_del_row(*tree*, *row*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Delete the given row (cutting plane constraint) from the cut pool.

ios_clear_pool(*tree*)
(To be used from inside a callback passed via the `cb_func` field of a `GLPK.IntoptParam` object. `tree` is a `Ptr{Void}` which must be the same obtained by the callback.)

Makes the cut pool empty deleting all existing rows (cutting plane constraints) from it.

init_env()
Initializes the GLPK environment. Not normally needed.

Returns 0 (initilization successful), 1 (environment already initialized), 2 (failed, insufficient memory) or 3 (failed, unsupported programming model).

version()
Returns the GLPK version number. In Julia, instead of returning a string as in C, it returns a tuple of integer values, containing the major and the minor number.

free_env()
Frees all resources used by GLPK routines (memory blocks, etc.) which are currently still in use. Not normally needed.

Returns 0 if successful, 1 if envirnoment is inactive.

term_out(*flag*)
Enables/disables the terminal output of glpk routines. `flag` is either `GLPK.ON` (output enabled) or `GLPK.OFF` (output disabled).

Returns the previous status of the terminal output.

open_tee(*filename*)
Starts copying all the terminal output to an output text file.

Returns 0 if successful, 1 if already active, 2 if it fails creating the output file.

close_tee()
Stops copying the terminal output to the output text file previously open by the `open_tee`.

Return 0 if successful, 1 if copying terminal output was not started.

malloc(*size*)
Replacement of standard C `malloc`. Allocates uninitialized memeory which must freed with `free`.

Returns a pointer to the allocated memory.

calloc(*n*, *size*)
Replacement of standard C `calloc`, but does not initialize the memeory. Allocates uninitialized memeory which must freed with `free`.

Returns a pointer to the allocated memory.

free(*ptr*)
Deallocates a memory block previously allocated by `malloc` or `calloc`.

mem_usage()
Reports some information about utilization of the memory by the routines `malloc`, `calloc`, and `free`. In

Julia, this function has a different API then C. It returns `(count, cpeak, total, tpeak)` rather then taking them as pointers in the argument list.

mem_limit (*limit*)
Limits the amount of memory avaliable for dynamic allocation to a value in megabyes given by the integer parameter `limit`.

read_cnfsat (*glp_prob*, *filename*)
Reads the CNF-SAT problem data in DIMACS format from a text file.

Returns 0 upon success; throws an error in case of failure.

check_cnfsat (*glp_prob*)
Checks if the problem object encodes a CNF-SAT problem instance, in which case it returns 0, otherwise returns non-zero.

write_cnfsat (*glp_prob*, *filename*)
Writes the CNF-SAT problem data in DIMACS format into a text file.

Returns 0 upon success; throws an error in case of failure.

minisat1 (*glp_prob*)
The routine `minisat1` is a driver to MiniSat, a CNF-SAT solver developed by Niklas Eén and Niklas Sörensson, Chalmers University of Technology, Sweden.

Returns 0 in case of success, or a non-zero flag specifying the reason for failure: `GLPK.EDATA` (problem is not CNF-SAT), `GLPK.EFAIL` (solver failure).

intfeas1 (*glp_prob*, *use_bound*, *obj_bound*)
The routine `glp_intfeas1` is a tentative implementation of an integer feasibility solver based on a CNF-SAT solver (currently MiniSat). `use_bound` is a flag: if zero, any feasible solution is seeked, otherwise seraches for an integer feasible solution. `obj_bound` is used only if `use_bound` is non-zero, and specifies an upper/lower bound (for maximization/minimazion respectively) to the objective function.

All variables (columns) must either be binary or fixed. All constraint and objective coeffient must be integer.

Returns 0 in case of success, or a non-zero flag specifying the reason for failure: `GLPK.EDATA` (problem data is not valid), `GLPK.ERANGE` (integer overflow occurred), `GLPK.EFAIL` (solver failure).

PYTHON MODULE INDEX

g

PYTHON MODULE INDEX