pyglpk Documentation

Release 0.4.8.dev5+g09b7740

Thomas Finley

Oct 10, 2024

CONTENTS:

CHAPTER

ONE

PYGLPK

Fork of T. Finley's PyGLPK module, available through `pip install glpk`.

You can find the documentation on https://pyglpk.readthedocs.io/

1.1 Overview

PyGLPK is a Python module which encapsulates the functionality of the GNU Linear Programming Kit (GLPK). The GLPK allows one to specify linear programs (LPs) and mixed integer programs (MIPs), and to solve them with either simplex, interior-point, or branch-and-bound algorithms. The goal of PyGLPK is to give one access to all documented functionality of GLPK.

1.2 Availability

To get the lastest version, see: https://github.com/bradfordboyle/pyglpk/.

1.3 Documentation

The HTML documentation included with the release in the directory html contains information on building, testing, installation and documentation of all features of the module.

1.4 Building and Installing

Building this module requires that the user have installed the GLPK 4.18 or later and GMP libraries. The module builds and appears to work on my test files in Python 2.4, and 2.5, with GLPK 4.18 through 4.31. Earlier versions of Python and GLPK will not work.

Ideally, the following will work:

- `make`
- `make test`
- `make install`, or perhaps `sudo make install`

See the HTML documentation for troubleshooting information.

1.5 Bugs and Commentary

Please send information on issues of usage to Thomas Finley at tfinley@gmail.com.

CHAPTER

TWO

DISCUSSION ABOUT PYGLPK

2.1 Existing Work

A Python binding to GLPK already exists in the form of Python-GLPK, but as it is automatically created through SWIG it is not very Pythonic.

2.2 Building and Installing

Building this module requires that the user have installed the GLPK and GMP libraries. The module builds and appears to work on my simple test files in Python 2.3, 2.4, and 2.5. Earlier versions of Python will not work.

In the ideal case (e.g., you have GLPK and GMP libraries and headers installed in default search paths), one would install this through `make` and `make install`, and be done.

The section on *Building and Installing* has more information.

2.3 High Level Comparison of C and PyGLPK

To use the C API, one would first `#include "glpk.h"`, create an LPX structure through `lpx_create_prob()`, and thence manipulate this structure with `lpx_*` functions to set the data, run the optimization, and retrieve the desired values.

To use this Python module, one would import the `glpk` module, create an LPX Python object through `glpk.LPX()`, and thence manipulate this object and the objects it contains to set the data, run the optimization, and retrieve the desired values. The Python module calls the C API to support these operations.

2.4 Design Philosophy

PyGLPK has objects floating around everywhere, and very few actual methods. Wherever possible and sensible, one gets and sets traits of the problem by accessing a method and directly assigning those traits. An example of this showing how PyGPLK works and how it does not work might be interesting.

PyGLPK is like this

```
lp.maximize = True
lp.cols.add(10)
for col in lp.cols:
```

(continues on next page)

(continued from previous page)

```
    col.name = 'x%d' % col.index
    col.bounds = 0.0, None
    lp.obj[col.index] = 1.0
del lp.cols[2,4,5]
```

It isn't like this

```
lp.set_maximize()
lp.add_cols(10)
for cnum in xrange(lp.num_cols()):
    lp.set_col_name(cnum, 'x%d' % cnum)
    lp.set_col_bounds(cnum, 0.0, None)
lp.set_obj_coef(cnum, 1.0)
lp.del_cols([2,4,5])
```

Both design strategies would accomplish the same thing, but there are advantages in the first way. For example, if I tell you only that columns of an LP are stored in a sequence `cols`, for free you already know a lot (assuming you're familiar with Python):

- You know how to get the number of columns in the LP.
- You know how to get a particular column or a range of columns.
- You know they're indexed from 0.
- You know how to delete them.

2.5 Differences Between C GLPK API and PyGLPK

2.5.1 Indexing

Unlike the C API, everything is indexed from 0, not 1: the user does not pass in arrays (or lists!) where the first element is ignored. Further, one indexes (for example) the first row by asking for row 0, not 1.

In the comparative examples of parallel C and Python code, wherever possible and appropriate I sprinkle +1 in the C code. Of course, only a lunatic would really write code that way, but I do this to highlight this difference, and second, make it more obvious which portions of C and Python correspond to each other: it's far easier to see the relation between `[7, 3, 1, 8, 6]` and `[7+1, 3+1, 1+1, 8+1, 6+1]`, versus `[8, 4, 2, 9, 7]`.

PyGLPK also honors Python's quirk of "negative indexing" used to count from the end of a sequence, e.g., where index -1 refers to the last item, -2 second to last item, and so forth. This can be convenient. For example, after adding a row, you can refer to this row by `lp.rows[-1]` rather than having to be aware of its absolute index.

2.5.2 Error Handling

The GLPK's approach to errors in arguments is deeply peculiar. It writes an error message and terminate the process, in contrast with many APIs that instead set or return an error code which can be checked. The PyGLPK takes the more Pythonic approach of throwing catchable exceptions for illegal arguments. Unfortunately, to avoid the process being terminated, this requires that absolutely every argument be vetted, requiring that PyGLPK have the additional overhead of doing sometimes rather detailed checks of arguments (which are, of course, checked yet again when the GLPK has access to them). It seems unlikely that GLPK's design will be improved in this area.

CHAPTER

THREE

SIMPLE EXAMPLE

To ground you, we show an example of the module's workings. (This example is covered in more detail in the examples section.) Taking the introductory example from the GLPK C API reference manual, we start with the following example linear program:

$$
\begin{array}{lll}
\underset{x}{\text{maximize}} & Z = 10x_0 + 6x_1 + 4x_2 & \\
\text{subject to} & p = x_0 + x_1 + x_2 & \\
 & q = 10x_0 + 4x_1 + 5x_2 & \\
 & r = 2x_0 + 2x_1 + 6x_2 & \\
\text{and bounds of variables} & -\infty p \leq 100 & 0 \leq x_0 \infty \\
 & -\infty q \leq 600 & 0 \leq x_1 \infty \\
 & -\infty r \leq 300 & 0 \leq x_2 \infty
\end{array}
$$

In the following, we show Python code to define and solve this problem, and subsequently print out the objective function value as well as the primal values of the structural variables.

```
import glpk                             # Import the GLPK module

lp = glpk.LPX()                         # Create empty problem instance
lp.name = 'sample'                      # Assign symbolic name to problem
lp.obj.maximize = True                  # Set this as a maximization problem
lp.rows.add(3)                          # Append three rows to this instance
for r in lp.rows:                       # Iterate over all rows
    r.name = chr(ord('p')+r.index)      # Name them p, q, and r
lp.rows[0].bounds = None, 100.0         # Set bound -inf < p <= 100
lp.rows[1].bounds = None, 600.0         # Set bound -inf < q <= 600
lp.rows[2].bounds = None, 300.0         # Set bound -inf < r <= 300
lp.cols.add(3)                          # Append three columns to this instance
for c in lp.cols:                       # Iterate over all columns
    c.name = 'x%d' % c.index            # Name them x0, x1, and x2
    c.bounds = 0.0, None                # Set bound 0 <= xi < inf
lp.obj[:] = [10.0, 6.0, 4.0]            # Set objective coefficients
lp.matrix = [
    1.0, 1.0, 1.0,                      # Set nonzero entries of the
    10.0, 4.0, 5.0,                     # constraint matrix.  (In this
    2.0, 2.0, 6.0                       # case, all are non-zero.)
]
lp.simplex()                            # Solve this LP with the simplex method
print 'Z = %g;' % lp.obj.value,         # Retrieve and print obj func value
print '; '.join(                        # Print struct variable names and primal values
```

(continues on next page)

(continued from previous page)

```
    '%s = %g' % (c.name, c.primal) for c in lp.cols
)
```

This may produce this output.

```
Z = 733.333; x0 = 33.3333; x1 = 66.6667; x2 = 0
```

CHAPTER

FOUR

DOWNLOAD

You may download the source package from here:

pyglpk-0.3.tar.bz2

An earlier version compatible with GLPK 4.4 onwards is here. This version also builds against GLPK as late as GLPK 4.31, but is missing much of the functionality added since GLPK 4.18:

pyglpk-0.2.tar.bz2

CHAPTER

FIVE

BUILDING AND INSTALLING

Building this module requires that the user have installed the GLPK 4.18 or later, and GMP libraries. The module builds and appears to work on my simple test files in Python 2.4 and 2.5. Earlier versions of Python may not work. I have developed this on both OS X and Linux. I do not know if it will work on Windows.

Here are the brief instructions for installing.

- Build the module using `make`. (Pay attention to warnings!)
- Run `make test` to execute the test suite. You want the last line of output to be `Tests succeeded!`
- Build the module using `make install` to install the module in your Python build's `site-packages` directory. On a typical system, you will need to run this as the superuser (probably `sudo make install`).

In the ideal case (that is, you have GLPK and GMP libraries, headers, executables and the like installed in default search paths), this will just work.

5.1 Troubleshooting

A few details just in case things don't go that easily:

How PyGLPK builds depends on which version of GLPK is installed. GLPK is an evolving library, and some functionality has been added only since older versions.

- Perhaps you don't have GLPK or GMP installed, or not the development versions with the headers.
- Perhaps your `INCLUDE_PATH`, `LIBRARY_PATH`, or `LD_LIBRARY_PATH` paths are not set correctly.
- Perhaps you have multiple versions of GLPK on your system in different locations, and it is confusing the system during building. E.g., it may build with one version, and then try to run the program with another.
- Perhaps the GLPK library was installed incorrectly. For example, on my Ubuntu box, I cannot build *anything* with the GLPK installed by my Debian package manager, not even a simple `main { LPX*lp=lpx_create_prob(); lpx_delete_prog(lp); }`, owing to a faulty compile of the libraries.

5.1.1 If You're Having Trouble Building

Below are some suggestions to try or avenues to pursue if you're having trouble building. This is not an exhaustive list, just "the most likely suspects."

- The build process assumes that the default Python (i.e., the Python you would get if you tried executing `python` within `sh`) is the one for which you want to build `glpk`. If this is untrue, modify the `Makefile` file to change the assignment `PYTHON:=python` to something like `PYTHON:=/path/to/other/python`.
- As described earlier, perhaps your environment variable paths are not set correctly. Relevant paths are `INCLUDE_PATH`, `LIBRARY_PATH`, `LD_LIBRARY_PATH` (yeah, I know).
- Perhaps you do not actually have a `glpk.h` file in the include directories. This may be because you have a binary non-developer installation of GLPK.
- Perhaps the `setup.py` script, in its attempt to do something reasonably clever, drew the wrong conclusions about where appropriate libraries and headers may be located, or the version of GLPK. (This may happen in certain situations where there are multiple builds of GLPK floating around with different versions, or your binaries are stored in strange locations.) Within `setup.py` is a small section beginning with `USERS DO CUSTOM INSTRUCTIONS HERE` where you can override its "cleverness" by manually defining the library name, location of appropriate library and include search directories, and other traits.
- Multiple versions of GLPK as described earlier. This may be addressed through editing the `setup.py`
- Faulty installation of GLPK.

5.1.2 If You're Having Trouble Testing

Once you have built the module successfully and there is a `glpk.so` symbolic link defined to the shared library, you should run the principle test suite through `make test`.

If you're running the test suite through `make test` and it throws some sort of exception, what the problem is depends on what type of error you see.

`ImportError: No module named glpk`
: Ensure that the module actually built.

`ImportError: (stuff) Symbol not found`
: Most likely, it cannot find a required library. Run `ldd glpk.so` (on Linux) or `otool -L glpk.so` (on OS X) to see which libraries it cannot find. Use this information to ensure that you environment variable `LD_LIBRARY_PATH` points to directories containing appropriate libraries. (Note that merely finding all libraries is not necessarily sufficient in the case where there are multiple GLPK libraries floating around. Ensure you're not building with one, but running with the other.)

`AssertionError` (or other error)
: The good news is that the module appears to load and run and it's throwing normal Python exceptions. The bad news is one of my tests failed. This should not happen, but if it does, send me a bug report.</dd>

5.1.3 If You're Having Trouble Installing

If it says permission denied when running `make install`, perhaps you need to run the command as the superuser, e.g., `sudo make install`.

5.1.4 Uninstalling

Like many Python modules, PyGLPK builds itself with the aid of Python's included `distutils` module. As of the time of this writing, `distutils` does not support uninstallation. I am not comfortable with writing my own solution. Given that this is a potentially destructive option, there may be unintended, unfortunate consequences on some person's configuration I did not anticipate. However, one may still uninstall PyGLPK by importing the `glpk` module, and checking the `glpk.__file__` attribute. This will tell you where the module file is stored. Then, you can go to that directory and remove the module file, and the associated `egg-info` file, and rid yourself of PyGLPK.

CHAPTER

SIX

EXAMPLES

6.1 Reference Manual Example

In this section we solve how to define and solve the example given at the beginning of the GLPK C API reference manual.

$$
\begin{array}{lll}
\underset{x}{\text{maximize}} & Z = 10x_0 + 6x_1 + 4x_2 & \\
\text{subject to} & p = x_0 + x_1 + x_2 & \\
 & q = 10x_0 + 4x_1 + 5x_2 & \\
 & r = 2x_0 + 2x_1 + 6x_2 & \\
\text{and bounds of variables} & -\infty p \leq 100 & 0 \leq x_0 \infty \\
 & -\infty q \leq 600 & 0 \leq x_1 \infty \\
 & -\infty r \leq 300 & 0 \leq x_2 \infty
\end{array}
$$

6.1.1 The Implementation

Here is the implementation of that linear program within PyGLPK:

```
import glpk                          # Import the GLPK module

lp = glpk.LPX()                      # Create empty problem instance
lp.name = 'sample'                   # Assign symbolic name to problem
lp.obj.maximize = True               # Set this as a maximization problem
lp.rows.add(3)                       # Append three rows to this instance
for r in lp.rows:                    # Iterate over all rows
    r.name = chr(ord('p')+r.index)   # Name them p, q, and r
lp.rows[0].bounds = None, 100.0      # Set bound -inf < p <= 100
lp.rows[1].bounds = None, 600.0      # Set bound -inf < q <= 600
lp.rows[2].bounds = None, 300.0      # Set bound -inf < r <= 300
lp.cols.add(3)                       # Append three columns to this instance
for c in lp.cols:                    # Iterate over all columns
    c.name = 'x%d' % c.index         # Name them x0, x1, and x2
    c.bounds = 0.0, None             # Set bound 0 <= xi < inf
lp.obj[:] = [10.0, 6.0, 4.0]         # Set objective coefficients
lp.matrix = [
    1.0, 1.0, 1.0,                   # Set nonzero entries of the
    10.0, 4.0, 5.0,                  # constraint matrix. (In this
    2.0, 2.0, 6.0                    # case, all are non-zero.)
```

(continues on next page)

(continued from previous page)

```
]
lp.simplex()                          # Solve this LP with the simplex method
print 'Z = %g;' % lp.obj.value,       # Retrieve and print obj func value
print '; '.join(                      # Print struct variable names and
                                      # primal values
    '%s = %g' % (c.name, c.primal) for c in lp.cols
)
```

6.1.2 The Explanation

We shall now go over this implementation section by section.

```
import glpk
```

First we need the module of course.

```
lp = glpk.LPX()                       # Create empty problem instance
```

Here we construct a new linear program object with a call to the LPX constructor.

```
lp.name = 'sample'                    # Assign symbolic name to problem
```

Linear program objects have various attributes. One of them is `name`, which holds a symbolic name for the program. Initially a program has no name, and `name` correspondingly has the value `None`. Here we name the program `'sample'`. Programs do not necessarily require names, but a user may wish to give a program a name nonetheless.

```
lp.obj.maximize = True                # Set this as a maximization problem
```

Linear program objects contain other less trivial attributes. One of the most important is `obj`, an object representing the linear program's objective function. In this case, we are assigning `lp.obj`'s `maximize` attribute the value `True`, informing out linear program that we want to maximize our objective function.

```
lp.rows.add(3)                        # Append three rows to this instance
```

Another very important component of `lp` is the `rows` attribute, holding an object which indexes over the rows of this linear program. In this case, we call the `lp.rows` method `add`, telling it to add three rows to the linear program.

```
for r in lp.rows:                     # Iterate over all rows
    r.name = chr(ord('p')+r.index)    # Name them p, q, and r
```

The `lp.rows` object is also used for accessing particular rows. In this case, we are iterating over each row. In the course of this iteration, `r` holds the first, second, and third row. We want to name these rows `'p'`, `'q'`, and `'r'`, in order.

Note that an individual row is an object in itself. It also has a `name` attribute, to which we assign the character with ASCII value of `p` plus whatever the index of this row is. The first row has `index` of 0, the next 1, the next and last 2. So, this will give us the desired names.

```
lp.rows[0].bounds = None, 100.0       # Set bound -inf < p <= 100
lp.rows[1].bounds = None, 600.0       # Set bound -inf < q <= 600
lp.rows[2].bounds = None, 300.0       # Set bound -inf < r <= 300
```

In addition to iterating over all rows, we can access a particular row by indexing the `lp.rows` object. In this case we index by the numeric row index. (Now that we have set their names, we could alternatively index them by their names!)

In this case, we are using the row's `bounds` attribute to set the bounds for the corresponding auxiliary variable. Bounds consist of a lower and upper bound. In this case, we are specifying that we always want the lower end unbounded (by assigning `None`, indicating no bound in that direction), and otherwise setting an appropriate upper bound.

```
lp.cols.add(3)                          # Append three columns to this instance
```

In addition to the `rows` object, there is also a `cols` object for creating and accessing columns. Indeed, the two objects have the same type. In this case, we see we are adding three columns to the linear program.

```
for c in lp.cols:                       # Iterate over all columns
    c.name = 'x%d' % c.index            # Name them x0, x1, and x2
    c.bounds = 0.0, None                # Set bound 0 <= xi < inf
```

Similar to how we iterated over and assigned names to the rows, in this case we assign appropriate names to our columns. We also assign bounds to each column's associated structural variable, though in this case we want each structural variable to be greater than 0, and have no upper bound.

```
lp.obj[:] = [10.0, 6.0, 4.0]            # Set objective coefficients
```

There is one objective coefficient for every column. In this, we set all the coefficients at once to their desired values. Note that these `lp.obj` objects act like sequences over the objective coefficient values, just as the row and column collections do over rows and the columns.

```
lp.matrix = [
    1.0, 1.0, 1.0,                      # Set nonzero entries of the
    10.0, 4.0, 5.0,                     # constraint matrix. (In this
    2.0, 2.0, 6.0                       # case, all are non-zero.)
]
```

We are setting the non-zero entries of the coefficient constraint matrix by assigning to the linear program's `matrix` attribute. Matrix entries are either (1) values, or (2) tuples specifying the row index, column index, and value. In the first case, if it is just a value with the indices omitted, it assumes that the value specified is for the next value in the constraint matrix, read top to bottom, left to right. We could also have explicitly defined the indices with this equivalent statement:

```
lp.matrix = [
    (0, 0, 1.0), (0, 1, 1.0), (0, 2, 1.0),
    (1, 0, 10.0), (1, 1, 4.0), (1, 2, 5.0),
    (2, 0, 2.0), (2, 1, 2.0), (2, 2, 6.0)
]
```

But we did not. Let's move on.

```
lp.simplex()                            # Solve this LP with the simplex method
```

Here we are calling a simplex solver to solve the defined linear program!

```
print 'Z = %g;' % lp.obj.value,      # Retrieve and print obj func value
print '; '.join(                     # Print struct variable names and
                                     # primal values
    '%s = %g' % (c.name, c.primal) for c in lp.cols
)
```

After optimization, we want to print out the value of the objective function (as stored in `lp.obj.value`), and the value of the primal variable for each of the columns (as stored in each column's `primal` attribute).

This all results in this output.

```
Z = 733.333; x0 = 33.3333; x1 = 66.6667; x2 = 0
```

6.2 Max Flow Example

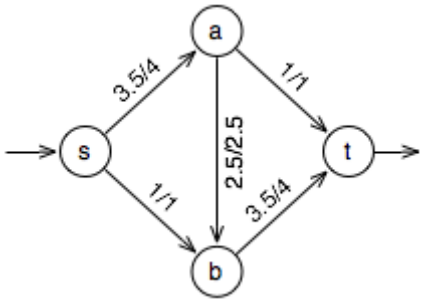
In this section we show a simple example of how to use PyGLPK to solve max flow problems.

6.2.1 How to Solve

Suppose we have a directed graph with a source and sink node, and a mapping from edges to maximal flow capacity for that edge. Our goal is to find a maximal feasible flow. This is the maximum flow problem.

This is our strategy of how to solve this with a linear program:

- For each edge, we define a structural variable (a column). The primal value of this structural variable is the flow assigned to the corresponding edge. We constrain this so it cannot exceed the edge's capacity.
- We define our objective function as the net flow of the source, a quantity we wish to maximize.
- For each non-source and non-sink node, we must have 0 net flow. So, for each non-source/sink node, we define an auxiliary variable (a row) equal to the sum of flows in minus the sum of flows out, which we constrain to be 0.
- We run the solver, and return the assignment of edges to flows.



For the purpose of our function, we encode our capacity graph as a list of three element tuples. Each tuple consists of a from node, a to node, and a capacity of the conduit from the from to the to node. Nodes can be anything Python can hash.

The function accepts such a capacity graph, and returns the a similar list of three element tuples. The list is identical as the input list, except the capacities are replaced with the assigned flow.

In the example graph seen at right (with given assigned/capacity weights given for the nodes), we have a source and sink `'s'` and `'t'`, and other nodes `'a'` and `'b'`. We would represent this capacity graph as `[('s','a',4), ('s','b',1), ('a','b',2.5), ('a','t',1), ('b','t',4)]`

6.2.2 The Implementation

Here is the implementation of that function:

```
import glpk


def maxflow(capgraph, s, t):
    # map non-source/sink nodes to row num
    node2rnum = {}
    for nfrom, nto, cap in capgraph:
        if nfrom != s and nfrom != t:
            node2rnum.setdefault(nfrom, len(node2rnum))
        if nto != s and nto != t:
            node2rnum.setdefault(nto, len(node2rnum))

    # empty LP instance
    lp = glpk.LPX()

    # stop annoying messages
    glpk.env.term_on = False

    # as many columns cap-graph edges
    lp.cols.add(len(capgraph))
    # as many rows as non-source/sink nodes
    lp.rows.add(len(node2rnum))

    # net flow for non-source/sink is 0
    for row in lp.rows:
        row.bounds = 0

    # will hold constraint matrix entries
    mat = []

    for colnum, (nfrom, nto, cap) in enumerate(capgraph):
        # flow along edge bounded by capacity
        lp.cols[colnum].bounds = 0, cap

        if nfrom == s:
            # flow from source increases flow value
            lp.obj[colnum] = 1.0
        elif nto == s:
            # flow to source decreases flow value
            lp.obj[colnum] = -1.0

        if nfrom in node2rnum:
            # flow from node decreases its net flow
            mat.append((node2rnum[nfrom], colnum, -1.0))
        if nto in node2rnum:
            # flow to node increases its net flow
            mat.append((node2rnum[nto], colnum, 1.0))

    # want source s max flow maximized
```

(continues on next page)

(continued from previous page)

```
    lp.obj.maximize = True
    # assign 0 net-flow constraint matrix
    lp.matrix = mat

    # this should work unless capgraph bad
    lp.simplex()

    # return edges with assigned flow
    return [
        (nfrom, nto, col.value)
        for col, (nfrom, nto, cap) in zip(lp.cols, capgraph)
    ]

capgraph = [
    ('s', 'a', 4), ('s', 'b', 1),
    ('a', 'b', 2.5), ('a', 't', 1), ('b', 't', 4)
]
print(maxflow(capgraph, 's', 't'))

[('s', 'a', 3.5), ('s', 'b', 1.0), ('a', 'b', 2.5),
 ('a', 't', 1.0), ('b', 't', 3.5)]
```

6.2.3 The Explanation

We shall now go over this function section by section!

```
    # map non-source/sink nodes to row num
    node2rnum = {}
    for nfrom, nto, cap in capgraph:
        if nfrom != s and nfrom != t:
            node2rnum.setdefault(nfrom, len(node2rnum))
        if nto != s and nto != t:
            node2rnum.setdefault(nto, len(node2rnum))
```

This is pure non-PyGLPK code, but it is doing something important for the linear program. Recall that we wanted a row for every non-source/sink node. In order to facilitate this, we first go through the capacity graph and map each node identifier (except the source and sink) to a unique integer, counting from 0 onwards.

```
    # empty LP instance
    lp = glpk.LPX()
```

We create an empty LPX instance.

```
    # stop annoying messages
    glpk.env.term_on = False
```

Linear program objects contain several objects through which one can access and set some of the data associated with a linear program. One of these is the `params` object, which holds attributes that help control the behavior of the linear program object when running a solver, writing data, and other routines. In this case, we are setting the `msglev` (message level) attribute to 0, to quiet the linear program.

```
    # as many columns cap-graph edges
    lp.cols.add(len(capgraph))
    # as many rows as non-source/sink nodes
    lp.rows.add(len(node2rnum))
```

In addition, the linear program object has two (largely identical) objects for accessing and setting traits of columns and rows, `cols` and `rows`, respectively. In this case, we are calling the `add` method of both objects.

Recall that we want as many structural variables (columns) as there are edges, to represent the assigned flow to edge edge. Correspondingly, we add as many columns as there are edges in the capacity graph. Also, we want as many auxiliary variables (rows) as there are non-source/sink nodes, in order to enforce the zero-net-flow constraint.

```
    # net flow for non-source/sink is 0
    for row in lp.rows:
        row.bounds = 0
```

In addition to being used to add rows and columns, the `rows` and `cols` objects serve as sequences, used to access row and column objects. In this case, we are iterating over all rows.

Once we have a row, we set its `bounds` attribute to 0 to force the row's auxiliary variable (and consequently the net flow for the corresponding node) to be zero. The `bounds` attribute can be assigned one or two values, depending on whether we want to assign an equality or range constraint. In this case, we want an equality constraint, and so assign the single value 0.

```
    # will hold constraint matrix entries
    mat = []
```

What's so special about an empty list? Nothing yet. However, what we are going to do is to give it elements of the linear program constraint matrix, and then set this as the linear program's constraint matrix.

The reason for this list is practical: We can set entries of the LP matrix either all at one, a whole row at a time, or a whole column at a time. None is really convenient given the structure of this problem, so we just save all the entries we want to be non-zero, and set them all at once when we have collected all of them.

Entries of the constraint matrix are given in the form of three element tuples describing the row index, column index, and the value at this location.

```
    for colnum, (nfrom, nto, cap) in enumerate(capgraph):
        # flow along edge bounded by capacity
        lp.cols[colnum].bounds = 0, cap

        if nfrom == s:
            # flow from source increases flow value
            lp.obj[colnum] = 1.0
        elif nto == s:
            # flow to source decreases flow value
            lp.obj[colnum] = -1.0

        if nfrom in node2rnum:
            # flow from node decreases its net flow
            mat.append((node2rnum[nfrom], colnum, -1.0))
        if nto in node2rnum:
            # flow to node increases its net flow
            mat.append((node2rnum[nto], colnum, 1.0))
```

We are iterating over all the edges in the capacity graph. A lot is happening inside the loop, so we shall take it a piece at a time.

```
31      for colnum, (nfrom, nto, cap) in enumerate(capgraph):
```

Since columns correspond to edges in the capacity graph, it is convenient to just suppose that edge `capgraph[i]` corresponds to column `lp.cols[i]`.

```
32          # flow along edge bounded by capacity
33          lp.cols[colnum].bounds = 0, cap
```

Each structural variable will get the value of the flow along its corresponding edge. Naturally, we want to constrain the flow assignments to be between 0 and the edge capacity. So, we assign to the `bounds` attribute of the column at index `colnum`. Note that this is an instance of a range bound (with lower bound `0` and upper bound `cap`), unlike our previous equality bound.

```
35          if nfrom == s:
36              # flow from source increases flow value
37              lp.obj[colnum] = 1.0
38          elif nto == s:
39              # flow to source decreases flow value
40              lp.obj[colnum] = -1.0
```

Recall we are trying to find the maximum flow across the graph, which equals the net flow from the source. The net flow increases whenever there is flow along an edge from the source, so if the edge is from the source, we set the corresponding structural variable's objective function coefficient to 1.0 (with the effect that the assigned flow is added to the objective). Conversely, the net flow decreases whenever there is flow along an edge to the source, so if the edge is to the source, we set the corrresponding coefficient to -1.0 (with the effect that the assigned flow is subtracted from the objective).

Similar to the `rows` and `cols` attributes of linear program objects, the `obj` attribute also acts like a sequence. We can assign objective function coefficients through simple assignments like this. There are as many objective coefficients as there are structural variables (columns).

```
42          if nfrom in node2rnum:
43              # flow from node decreases its net flow
44              mat.append((node2rnum[nfrom], colnum, -1.0))
45          if nto in node2rnum:
46              # flow to node increases its net flow
47              mat.append((node2rnum[nto], colnum, 1.0))
```

The intent of this is very similar to our coefficients set in the objective function. We wish the net flow into a node to be zero. Correspondingly, for each edge, we add (at most) two entries to the matrix of constraint coefficients. For the "from" node of an edge, we add a -1.0 coefficient to the "from" node's corresponding row, effectively subtracting off the value of the edge's structural variable from the "from" node's auxiliary variable. For the "to" node of an edge, we add a 1.0 coefficient to the "to" node's corresponding row, effectively adding the value of the edge's structural variable to the "to" node's auxiliary variable.

The `if` statements are present because we only want to add this structural variables for nodes that correspond to rows, i.e., non-source/sink nodes.

This marks the end of that loop.

```
49      # want source s max flow maximized
50      lp.obj.maximize = True
```

The `obj` object has an attribute `maximize` that controls whether we are trying to maximize or minimize the objective function. In this case, we want a maximizing assignment of flow to edges.

```
51      # assign 0 net-flow constraint matrix
52      lp.matrix = mat
```

We set the constraint matrix to the entries that we have collected.

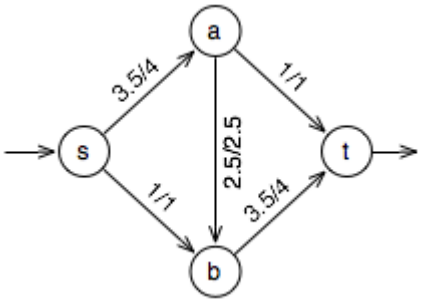
```
54      # this should work unless capgraph bad
55      lp.simplex()
```

Next we run the simplex algorithm to optimize this linear program. The simplex algorithm has strong theoretical ties to the max augmenting path algorith (think about the operations that are taking place in the simplex tableau), so if we have defined a valid capacity graph this should converge with no problems.

```
57      # return edges with assigned flow
58      return [
59          (nfrom, nto, col.value)
60          for col, (nfrom, nto, cap) in zip(lp.cols, capgraph)
61      ]
```

More or less straightforward Python code to construct the return value. For all the columns and the corresponding edges, we return the triples of "from," "to," and the variable value, which is the assigned flow. Note the use of `col.value` attribute to extract the primal variable value for this column.

6.2.4 Example Run



Imagine that we run this call to find the max-flow for the given graph.

```
capgraph = [
    ('s', 'a', 4), ('s', 'b', 1),
    ('a', 'b', 2.5), ('a', 't', 1), ('b', 't', 4)
]
print(maxflow(capgraph, 's', 't'))
```

This will produce the output

```
[
    ('s', 'a', 3.5), ('s', 'b', 1.0), ('a', 'b', 2.5),
    ('a', 't', 1.0), ('b', 't', 3.5)
]
```

corresponding to the flow shown in the graph.

6.3 SAT Example

In this section we show a simple example of how to use PyGLPK to build a SAT solver.

6.3.1 How to Solve

Suppose one has a CNF expression , that is, a conjunction (and-ing) of several disjunctions (or-ing) of logical literals, e.g.:

$$(\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

Suppose we want to find truth values to all four <var>x_i</var> variables so that the CNF expression is true. This problem has been viewed from many different ways, but we'll see how to encode and (we hope) solve it within a mixed-linear program. We will build a function <code class="py">solve_sat</code> to satisfy a given CNF.

First, we need to define how we encode our input CNF expressions that we want to satisfy:

- Each logical literal is represented as either a positive or negative integer, where `i` and `-i` correpond to the logical literals x_i and $\neg x_i$, respectively.
- Each clause in the expression, i.e., disjunction of literals, is represented as a tuple of such encoding of literals, e.g., `(-1, 2, -3)` represents the disjunction $(\neg x_1 \vee x_2 \vee \neg x_3)$.
- The entire conjunctive expression is a list of such tuples, e.g., the expression above would have encoding

```
[
    (-1, -3, -4),
    (2, 3, -4),
    (1, -2, 4),
    (1, 3, 4),
    (-1, 2, -3)
]
```

The function will return either `None` if it could not find a satisfying assignment, or a list of booleans `assignment` representing the satisfying assignment, where the truth of each logical variable x_i is held in `assignment[i-1]`.

This is our strategy of how to solve this with a mixed integer program:

- For each logical variable x_i, have a structural variable (column) representing both its positive and negative literals x_i and $\neg x_i$. These structural variables should be either 0 or 1 depending on whether the corresponding literal is false or true, respectively.
- Because we want literal consistency, we specify that the sum of all literal pair structural variables must be 1. This forbids literals for a given logical variable from being set both false or both true.
- For each clause, we define a constraint specifying that the sum of all its literal structural variables must be at least 1. This forces each clause to be true.
- First we run the simplex solver (imlying a relaxed problem where the structural variables can range from 0 to 1). Then we run the integer solver (the structural variables can be either 0 or 1).
- If the integer solver finds an optimal solution, excellent. (If not we return `None`.) If a positive literal has a corresponding structural variable with value 1, then we assign its logical variable to true.

6.3.2 The Implementation

Here is the implementation of that function:

```
import glpk


def solve_sat(expression):
    # trivial case
    if len(expression) == 0:
        return []

    # otherwise count vars
    numvars = max([max([abs(v) for v in clause]) for clause in expression])

    # empty LP instance
    lp = glpk.LPX()

    # stop annoying messages
    glpk.env.term_on = False

    # as many columns as there are literals
    lp.cols.add(2*numvars)
    # literal must be between false and true
    for col in lp.cols:
        col.bounds = 0.0, 1.0

    # function to compute column index
    def lit2col(lit):
        return [2*(-lit)-1, 2*lit-2][lit > 0]

    # ensure "oppositeness" of literals
    for i in xrange(1, numvars+1):
        lp.rows.add(1)
        lp.rows[-1].matrix = [(lit2col(i), 1.0), (lit2col(-i), 1.0)]
        # must sum to exactly 1
        lp.rows[-1].bounds = 1.0

    # ensure "trueness" of each clause
    for clause in expression:
        lp.rows.add(1)
        lp.rows[-1].matrix = [(lit2col(lit), 1.0) for lit in clause]
        # at least one literal must be true
        lp.rows[-1].bounds = 1, None

    # try to solve the relaxed problem
    retval = lp.simplex()

    # should not fail in this fashion
    assert retval is None
    # ff no relaxed solution, no exact sol
    if lp.status != 'opt':
        return None
```

(continues on next page)

(continued from previous page)

```
50
51      for col in lp.cols:
52          col.kind = int
53
54      # try to solve this integer problem
55      retval = lp.integer()
56      # should not fail in this fashion
57      assert retval is None
58      if lp.status != 'opt':
59          return None
60
61      return [col.value > 0.99 for col in lp.cols[::2]]
62
63  exp = [
64      (-1, -3, -4),
65      (2, 3, -4),
66      (1, -2, 4),
67      (1, 3, 4),
68      (-1, 2, -3)
69  ]
70  print(solve_sat(exp))
```

6.3.3 The Explanation

We shall now go over this function section by section!

```
5       # trivial case
6       if len(expression) == 0:
7           return []
8
9       # otherwise count vars
10      numvars = max([max([abs(v) for v in clause]) for clause in expression])
```

Pretty straightforward non-PyGLPK Python code. The first line just takes care of the boundary case where we have an empty expression. In the second line, from the expression, we find the maximum indexed logical variable we have, and use that as our count of the number of logical variables.

```
12      # empty LP instance
13      lp = glpk.LPX()
```

Calls the LPX constructor to construct an empty linear program.

```
15      # stop annoying messages
16      glpk.env.term_on = False
```

Within the `glpk` module member `env`, of type `Environment`. By assigning to various attributes contained within `env`, you can affect behavior of the GLPK object. In this case, we are assigning `False` to the `term_on` (terminal output on) parameter, to suppress all output.

```
18      # as many columns as there are literals
19      lp.cols.add(2*numvars)
```

Recall that we want as many structural variables (columns) in the linear program as there are possible literals over all our logical variables. Each logical variable x_i has two posible literals: itself (x_i), and its negation (neg x_i).

Initially we have no columns at all. So, we get the `lp.cols` object, the LP's column container, and call its `add` method, telling it to add as many columns as there are twice the number of logical variables.

```
    # literal must be between false and true
    for col in lp.cols:
        col.bounds = 0.0, 1.0
```

In addition to creating new columns, the `lp.cols` collection object is used to access individual columns. In this case, we are iterating over every column. Once we have each column, we assign its bounds to be between 0 and 1. This will force the structural variable associated with this column to fall between these values.

These `lp.cols` objects act like sequences (albeit with restrictions on their content). In order to access their elements (in this case, columns), we can either iterate over the columns as we do here, or index into them directly as `lp.cols[colnum]`.

```
    # function to compute column index
    def lit2col(lit):
        return [2*(-lit)-1, 2*lit-2][lit > 0]
```

This is just a helper function for our own benefit. Recall that we have a structural variable for each possible literal. This function merely maps a literal code to a column index. This function maps literal code 1 to column index 0, -1 to column index 1, 2 to 2, -2 to 3, 3 to 4, -3 to 5, 4 to 6, and so forth.

```
    # ensure "oppositeness" of literals
    for i in xrange(1, numvars+1):
        lp.rows.add(1)
        lp.rows[-1].matrix = [(lit2col(i), 1.0), (lit2col(-i), 1.0)]
        # must sum to exactly 1
        lp.rows[-1].bounds = 1.0
```

These are our consistency constraints to make sure two opposite literals are not both true or not both false. For each logical variable, we add one new row (what will be a consistency constraint). Notice that we are now using the `lp.rows` object! This is similar to the `lp.cols` object (in reality they are the same type), except it holds rows instead of columns.

Then we get the last row, which is the one we just added (note the use of the -1 index to address the last row), and assign to its `matrix` attribute. The `matrix` attribute for any row or column corresponds to the entries of the row or column vector in our constraint matrix. In this case, we are setting the two locations of this constraint matrix row corresponding to the two structural variables for x_i and :math:*neg x_i* to 1.0.

Finally, we set the `bounds` attribute for this row's auxiliary variable to `1.0`. Note that this differs from the previous bound definition: here we use only one number. This indicates we want an equality constraint. (It would have been equivalent to assign `1.0, 1.0`.)

```
    # ensure "trueness" of each clause
    for clause in expression:
        lp.rows.add(1)
        lp.rows[-1].matrix = [(lit2col(lit), 1.0) for lit in clause]
        # at least one literal must be true
```

These are our clause satisfiability constraints, to make sure that at least one literal in each clause is true. For each clause we, again, add a single row.

We access this last added row, and assign to its `matrix` attribute. In this case, we are specifying that the row's constraint coefficients should be 1.0 for each structural variable corresponding to each literal within this clause.

Finally, we set the `bounds` attribute for this row, establishing the lower bound `1` and upper bound `None`. An assignment of `None` indicates unboundedness in this direction.

```
41
42      # try to solve the relaxed problem
43      retval = lp.simplex()
44
45      # should not fail in this fashion
```

Now we employ the simplex solver to attempt to solve a relaxed version of the problem. (Relaxed in the sense that the variables can be non-integers.) We do this because, at the point it is called, the integer optimization method requires an existing optimal basic solution.

The method returns `None` unless the method was unable to start the search due to a fault in the problem definition (which returns the string `'fault'`), or because the simplex search terminated prematurely (due to one of several possible conditions).

In a real application we would probably be interested in seeing what went wrong, and try to fix it. However, for this toy example, we just noisily fail with an exception.

```
46      assert retval is None
47      # ff no relaxed solution, no exact sol
48      if lp.status != 'opt':
```

Note that "not terminating prematurely" does not mean "an optimal solution was found!" It merely means that the search did not terminate abnormally. In order to check whether we found an optimal solution (as opposed to, say, having determined that the problem is infeasible), we check the `status` attribute. If it does not hold `'opt'`, then we return `None` to indicate that we could not find a satisfying assignment.

At this point we hold an optimal basic solution to the relaxed problem. We now go about turning this into a mixed-integer program.

```
50
51      for col in lp.cols:
```

We first assign the columns the `kind` of `int` to indicate that we want this to be an integer program. The `kind` attribute can be either `float`, `int`, or `bool`. (What a horrible abuse of types!)

```
53
54      # try to solve this integer problem
55      retval = lp.integer()
56      # should not fail in this fashion
57      assert retval is None
58      if lp.status != 'opt':
```

This is very similar to our invocation of the `simplex` solver, except this time we are using the `integer` solver. Again, we fail noisily if we encounter something unexpected, and quietly return `None` if we could not find a satisfying assignment.

```
60
```

This function is supposed to return a satisfying truth assignment to all our variables if such an assignment is possible. Since we have gotten this far without returning `None`, we know we have one: a variable is true if its positive literal has a corresonding structural variable of 1.

Note that literal x_1 corresponds to column 0, x_2 to column 2, x_3 to column 4, and so forth. We go over each of the even columns (using the slice `::2` to indicate every column from beginning to end, counting by 2s), test whether the value of this columns variable is 1, and return the resulting list as our satisfying assignment.

We are done!

6.3.4 Example run

So, how does this work? Recall our CNF formula.

$$(\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

This has the encoding `[(-1, -3, -4), (2, 3, -4), (1, -2, 4), (1, 3, 4), (-1, 2, -3)]`

Suppose we run this in our Python interpreter.

```
exp = [
    (-1, -3, -4),
    (2, 3, -4),
    (1, -2, 4),
    (1, 3, 4),
    (-1, 2, -3)
]
```

This prints out:

```
[True, True, False, False]
```

So, $x_1 = T$, $x_2 = T$, $x_3 = F$, and $x_4 = F$. Is this a satisfying assignment? The first and second clauses are true because $\neg x_4$. The third and fourth clauses are true because x_1. The fifth (last) clause is true because x_2.

Now suppose we input the expression $x_1 \wedge \neg x_1$, which is plainly unsatisfiable.

```
exp = [(-1,), (1,)]
print solve_sat(exp)
```

This prints out:

```
None
```

Success! Or, at least, what we should expect.

6.3.5 Fun Variants

This problem is a little unusual in that we did not specify an objective function, leaving it to its default constant 0 value. We do not care which assignment we get. However, what if we wanted as many of our logical variables to be true as possible?

Suppose, right after the `lp.cols.add(2*numvars)` statement in our function, we added the following snippet.

```
lp.obj[::2] = 1
lp.obj.maximize = True
```

We assign all even indexed objective coefficients (i.e., those corresponding to the positive literal structural variables) to 1, and say we want our LP to maximize this objective function. In other words, we want to maximize the sum of structural variables corresponding to positive assignments. If we repeat our run, we get

```
[True, True, True, False]
```

Different, but still a satisfying assignment! Now we have 3 logical variables true instead of 2. Suppose now we say we want to minimize this function, that is, we edit the snippet so now it reads

```
lp.obj[::2] = 1
lp.obj.maximize = False
```

Repeating our run again, we get

```
[False, False, True, False]
```

Now only one logical variable is true!

6.4 Hamiltonian Path Example

In this section we show a simple example of how to use PyGLPK to solve the Hamiltonian path problem. This particular example is intended to be much more high level for those frustrated by lengthly explanations with excessive hand holding.

6.4.1 How to solve

Suppose we have an undirected graph. We want to find a path from any node to any other node such that we visit each node exactly one. This is the Hamiltonian path problem!

This is our strategy of how to solve this with a mixed integer program:

- For each edge in the graph we define a structural variable (a column). This will be a binary variable with 1 indicating that this edge is part of the path.
- All nodes must have either 1 or 2 incident edges selected as part of the path, no more, no fewer. Therefore we introduce a row for each node to encode this constraint.
- Exactly as many edges should be selected as the number of nodes minus one. This constraint requires another node.

For the purpose of our function, we encode graphs as a list of two element tuples. Each tuple represents an edge between two vertices. Vertices can be anything Python can hash. We assume we can infer the vertex set from this edge set, i.e., there are no isolated vertices. (If there are, then the problem is trivial anyway.) For example, `[(1,2), (2,3), (3, 4), (4,2), (3,5)]` would represent a graph over five nodes, containing a K_3 subgraph (among vertices 2,3,4) with additional vertices 1 attached to 2 and 5 attached to 3.

The function accepts such a list of edges, and return the sublist comprising the path, or `None` if it could not find a path.

6.4.2 The Implementation

Here is the implementation of that function:

```
import glpk


def hamiltonian(edges):
    # maps node to col indices of incident edges
    node2colnums = {}
    for colnum, edge in enumerate(edges):
        n1, n2 = edge
        node2colnums.setdefault(n1, []).append(colnum)
        node2colnums.setdefault(n2, []).append(colnum)

    # empty LP instance
    lp = glpk.LPX()

    # stop annoying messages
    glpk.env.term_on = False

    # a struct var for each edge
    lp.cols.add(len(edges))
    # constraint for each node
    lp.rows.add(len(node2colnums) + 1)

    # make all struct variables binary
    for col in lp.cols:
        col.kind = bool

    # for each node, select at least 1 and at most 2 incident edges
    for row, edge_column_nums in zip(lp.rows, node2colnums.values()):
        row.matrix = [(cn, 1.0) for cn in edge_column_nums]
        row.bounds = 1, 2

    # we should select exactly (number of nodes - 1) edges total
    lp.rows[-1].matrix = [1.0] * len(lp.cols)
    lp.rows[-1].bounds = len(node2colnums) - 1

    # should not fail this way
    assert lp.simplex() is None
    # if no relaxed sol., no exact sol.
    if lp.status != 'opt':
        return None

    # should not fail this way
    assert lp.integer() is None
    # could not find integer solution
    if lp.status != 'opt':
        return None

    # return edges whose associated struct var has value 1
    return [edge for edge, col in zip(edges, lp.cols) if col.value > 0.99]
```

(continues on next page)

(continued from previous page)

```
"""
1----2----3----5
      \  /
       \/  Has one H path!
       4

Path:
[(1, 2), (3, 4), (4, 2), (3, 5)]
"""

g1 = [(1, 2), (2, 3), (3, 4), (4, 2), (3, 5)]
print(hamiltonian(g1))

"""
4    5    6
|    |    |
|    |    | Has no H path!
1----2----3

Path:
None
"""

g2 = [(1, 2), (2, 3), (1, 4), (2, 5), (3, 6)]
print(hamiltonian(g2))

"""
4    5----6
|    |    |
|    |    | Has two H paths!
1----2----3

Path:
[(1, 2), (1, 4), (2, 5), (3, 6), (5, 6)]
"""

g3 = g2 + [(5, 6)]
print(hamiltonian(g3))
```

6.4.3 The Explanation

We shall now go over this function section by section, but not quite in such exhaustive detail as before.

```
    # maps node to col indices of incident edges
    node2colnums = {}
    for colnum, edge in enumerate(edges):
        n1, n2 = edge
        node2colnums.setdefault(n1, []).append(colnum)
        node2colnums.setdefault(n2, []).append(colnum)
```

This is pure Python non-PyGLPK code. It is simply computing a mapping of nodes to a list of column numbers

corresponding to each node's incident edges. Note that each edge will have the same index in both the column collection, as well as the input `edges` list.

```
    # empty LP instance
    lp = glpk.LPX()

    # stop annoying messages
    glpk.env.term_on = False

    # a struct var for each edge
    lp.cols.add(len(edges))
    # constraint for each node
    lp.rows.add(len(node2colnums) + 1)
```

In this section, we create a new empty linear program, make it quiet, and add as many columns as there are edges, and as many rows as there are vertices, plus one additional row to encode the constraint that exactly as many edges should be selected as there are vertices minus one.

```
    # make all struct variables binary
    for col in lp.cols:
        col.kind = bool
```

Here, we set our LP as a MIP, and going over the columns set the associated structural variable to be binary (i.e., have 0 to 1 bounds and be an integer variables).

```
    # for each node, select at least 1 and at most 2 incident edges
    for row, edge_column_nums in zip(lp.rows, node2colnums.values()):
        row.matrix = [(cn, 1.0) for cn in edge_column_nums]
        row.bounds = 1, 2
```

These are the constraints that say for each node, either one or two edges should be selected. For each node, we set a row to have a constraint which sums over all of the structural variables representing edges incident to that node, and forces this sum to be between 1 and 2.

```
    # we should select exactly (number of nodes - 1) edges total
    lp.rows[-1].matrix = [1.0] * len(lp.cols)
    lp.rows[-1].bounds = len(node2colnums) - 1
```

Similarly, have the last row in the constraint matrix encode the constraint that the sum of all the structural variables (i.e., the number of edges selected) should be the number of vertices minux one.

Note how in this case we do not specify the column indices in our matrix assignment: we just assign a long list of 1.0 values, and use how matrix assignments will implicitly assue that each single value will be placed in the entry directly after the last assigned value.

```
    # should not fail this way
    assert lp.simplex() is None
    # if no relaxed sol., no exact sol.
    if lp.status != 'opt':
        return None

    # should not fail this way
    assert lp.integer() is None
    # could not find integer solution
```

(continues on next page)

(continued from previous page)

```
45      if lp.status != 'opt':
46          return None
```

As in the SAT example, we run the simplex solver to come up with an initial continuous relaxed basic solution to this problem. We fail, we miss the assertion, and if the optimal solution was not found, we return that there is no solution. We then run the integer optimizer.

```
48      return [edge for edge, col in zip(edges, lp.cols) if col.value > 0.99]
```

Finally, in this case, we select out those columns which have a value close to 1 (indicating this edge was selected) and return the associated edge using our `colnum2edge` map we constructed at the start of the function.

6.4.4 Example Run

Suppose we have, after this function definition, these calls.

```
"""
1----2----3----5
      \  /
       \/  Has one H path!
       4

Path:
[(1, 2), (3, 4), (4, 2), (3, 5)]
"""

g1 = [(1, 2), (2, 3), (3, 4), (4, 2), (3, 5)]
print(hamiltonian(g1))
```

```
"""
4    5    6
|    |    |
|    |    | Has no H path!
1----2----3

Path:
None
"""

g2 = [(1, 2), (2, 3), (1, 4), (2, 5), (3, 6)]
print(hamiltonian(g2))
```

```
"""
4    5----6
|    |    |
|    |    | Has two H paths!
1----2----3

Path:
[(1, 2), (1, 4), (2, 5), (3, 6), (5, 6)]
"""
```

(continues on next page)

(continued from previous page)

```
g3 = g2 + [(5, 6)]
print(hamiltonian(g3))
```

This will produce this output.

```
[(1, 2), (3, 4), (4, 2), (3, 5)]
None
[(1, 2), (1, 4), (2, 5), (3, 6), (5, 6)]
```

6.4.5 Fun TSP Variant

Note that we did not define an objective function value. If we wanted, we could solve the traveling salesman problem (with symmetric weights) by making the following modifications:

- Providing each edge with a cost of taking this edge. (Perhaps as a three element tuple instead of a two element tuple.) We could then set the associated edge's objective function value to this edge, and set this to a minimization problem.
- Given that the TSP computes cycles and not paths, change the 1 or 2 bounds to equality on 2 (each node has exactly 2 incident selected edges), and further refine the last constraint that as many edges as nodes must be selected.

Here is the resulting function. Sections with important changes have been bolded.

```
import glpk


def tsp(edges):
    # Maps node to col indices of incident edges.
    node2colnums = {}
    for colnum, edge in enumerate(edges):
        n1, n2, cost = edge
        node2colnums.setdefault(n1, []).append(colnum)
        node2colnums.setdefault(n2, []).append(colnum)

    # empty LP instance
    lp = glpk.LPX()

    # stop annoying messages
    glpk.env.term_on = False

    # A struct var for each edge
    lp.cols.add(len(edges))
    # constraint for each node
    lp.rows.add(len(node2colnums)+1)

    # try to minimize the total costs
    lp.obj[:] = [e[-1] for e in edges]
    lp.obj.maximize = False

    # make all struct variables binary
```

(continues on next page)

(continued from previous page)

```
    for col in lp.cols:
        col.kind = bool

    # for each node, select two edges, i.e.., an arrival and a departure
    for row, edge_column_nums in zip(lp.rows, node2colnums.values()):
        row.matrix = [(cn, 1.0) for cn in edge_column_nums]
        row.bounds = 2

    # we should select exactly (number of nodes) edges total
    lp.rows[-1].matrix = [1.0]*len(lp.cols)
    lp.rows[-1].bounds = len(node2colnums)

    assert lp.simplex() is None
    # if no relaxed sol., no exact sol.
    if lp.status != 'opt':
        return None

    # should not fail this way
    assert lp.integer() is None
    if lp.status != 'opt':
        # could not find integer solution
        return None

    # return the edges whose associated struct var has value 1
    return [edge for edge, col in zip(edges, lp.cols) if col.value > 0.99]
```

CHAPTER

SEVEN

BRIEF REFERENCE

The following table is a brief overview of the functionality of PyGLPK. The intention was to present functionality in roughly the same order and groupings that they do in the GLPK reference manual, at least where appropriate. This section contains a brief description of some functionality, some simple Python code illustrating the general principles of usage, the related C API function (for the benefit of those familiar with GLPK; this information can be safely ignored if you are not), and a link to more detailed documentation on this subject with the **?** links.

7.1 Problem Attributes

Create or delete problem object

```
lp = glpk.LPX()  # glp_create_prob
del(lp)          # glp_delete_prob
```

Reinitialize problem object

```
lp.erase()  # glp_erase_prob
```

Set or get problem name

```
lp.name = "pname"  # glp_set_prob_name
del(lp.name)       # glp_set_prob_name

lp.name            # glp_get_prob_name
```

Set or get objective function name

```
lp.obj.name = "oname"  # glp_set_obj_name
del(lp.obj.name)       # glp_set_obj_name

lp.obj.name            # glp_get_obj_name
```

Set or get optimization direction

```
lp.obj.maximize = True  # glp_set_obj_dir
lp.obj.maximize         # glp_get_obj_dir
```

Add new rows or columns to a problem

```
lp.rows.add(num_to_add)  # glp_add_rows
lp.cols.add(num_to_add)  # glp_add_cols
```

Set or get row or column name

```
lp.rows[ri].name = "rname"  # glp_set_row_name
del(lp.rows[rnum].name)     # glp_set_row_name

lp.cols[ci].name = "cname"  # glp_set_col_name
del(lp.cols[cnum].name)     # glp_set_col_name

lp.rows[ri].name            # glp_get_row_name
lp.cols[ci].name            # glp_get_col_name
```

Set or get row or column bounds

```
lp.rows[ri].bounds = lower, upper  # glp_set_row_bnds
lp.rows[ri].bounds = equals        # glp_set_row_bnds

lp.cols[ci].bounds = lower, upper  # glp_set_col_bnds
lp.cols[ci].bounds = equals        # glp_set_col_bnds

lp.rows[ri].bounds                 # glp_get_row_type
                                   # glp_get_row_lb
                                   # glp_get_row_ub

lp.cols[ci].bounds                 # glp_get_col_type
                                   # glp_get_col_lb
                                   # glp_get_col_ub
```

Set or get objective coefficient or shift term

```
# use index None to get shift term
lp.obj[ci] = coef  # glp_set_obj_coef
lp.obj[ci]         # glp_get_obj_coef
```

Set or get row or column of the constraint matrix

```
lp.rows[ri].matrix = [(ci1,val1), (ci2,val2), ...]  # glp_set_mat_row
lp.cols[ci].matrix = [(ri1,val1), (ri2,val2), ...]  # glp_set_mat_col
lp.rows[ri].matrix                                  # glp_get_mat_row
lp.cols[ci].matrix                                  # glp_get_mat_col
```

Set or get the whole constraint matrix

```
lp.matrix = [(ri1,ci1,val1), (ri2,ci2,val2), ...]  # glp_load_matrix
lp.matrix
```

Delete rows or columns from problem object

```
del(lp.rows[ri1, ri2, ...])  # glp_del_rows
del(lp.rows[r_lo:r_hi+1])    # glp_del_rows

del(lp.cols[ci1, ci2, ...])  # glp_del_cols
del(lp.cols[c_lo:c_hi+1])    # glp_del_cols
```

Delete problem object

```
# or just let garbage collector handle it
del(lp)  # glp_delete_prob
```

7.2 Indexing rows and columns by name

Index row or column by its name

```
lp.rows['rowname']  # glp_find_row
# to check for membership
'rowname' in lp.rows

lp.cols['colname']  #
# to check for membership
'colname' in lp.cols
```

The index is created when required.

7.3 Problem Scaling

Automatically scale or unscale problem data

```
lp.scale()    # glp_scale_prob
lp.unscale()  # glp_unscale_prob
```

Set or get scaling of row and column data

```
lp.rows[i].scale = factor  # glp_set_rii
lp.cols[i].scale = factor  # glp_set_sjj
lp.rows[i].scale           # glp_get_rii
lp.cols[i].scale           # glp_get_sjj
```

7.4 Basis operations

Construct trivial initial LP basis

```
lp.std_basis()  # glp_std_basis
```

Construct advanced initial LP basis

```
lp.adv_basis()  # glp_adv_basis
```

Construct advanced initial LP basis with Bixby's algorithm

```
lp.cpx_basis()  # glp_cpx_basis
```

Read initial LP basis from a file

```
lp.read_basis(filename)  # lpx_read_bas
```

Set row or column status

```
lp.rows[ri].status = newstatus  # glp_set_row_stat
lp.cols[ci].status = newstatus  # glp_set_col_stat
```

7.5 Basic simplex solvers

Solve problem with the simplex method

```
lp.simplex()  # glp_simplex
```

Solve problem with an exact arithmetic using simplex method

```
lp.exact()  # lpx_exact
```

Get generic, primal, or dual status of basic solution

```
lp.status         # glp_get_status
lp.status_s       # to force simplex status
lp.status_primal  # glp_get_prim_stat
lp.status_dual    # glp_get_dual_stat
```

Get objective value

```
lp.obj.value    # glp_get_obj_val
lp.obj.value_s  # to force simplex value
```

Get row or column status

```
# a string, one of 'bs', 'nl','nu','nf','ns'
lp.rows[ri].status  # glp_get_row_stat

# a string, one of 'bs', 'nl','nu','nf','ns'
lp.cols[ci].status  # glp_get_col_stat
```

Get row or column primal or dual value

```
lp.rows[ri].primal    # glp_get_row_prim
lp.rows[ri].primal_s  # to force simplex value

lp.rows[ri].dual      # glp_get_row_dual
lp.rows[ri].dual_s    # to force simplex value

lp.cols[ci].primal    # glp_get_col_prim
lp.cols[ci].primal_s  # to force simplex value

lp.cols[ci].dual      # glp_get_col_dual
lp.cols[ci].dual_s    # to force simplex value
```

Get non-basic variable causing unboundness

```
# a row or column, or None if none has been identified
lp.ray ()  # lpx_get_ray_info
```

Check solution's Karush-Kuhn-Tucker conditions

```
lp.kkt()  # lpx_check_kkt
```

7.6 Manual simplex tableau operations

Warm-up LP basis

```
lp.warm_up()  # glp_warm_up
```

Compute a row of the simplex tableau

```
lp.rows[ri].eval_tab_row()  # glp_eval_tab_row
```

Compute a column of the simplex tableau

```
lp.cols[ci].eval_tab_col()  # glp_eval_tab_col
```

Transform explicitly specified row

```
lp.transform_row()  # glp_transform_row
```

Transform explicitly specified column

```
lp.transform_column()  # glp_transform_col
```

Perform primal ratio test

```
lp.prime_ratio_test()  # glp_prim_ratio_test
```

Perform dual ratio test

```
lp.dual_ratio_test()  # glp_dual_ratio_test
```

7.7 Interior-point solver

Solve problem with the interior-point method

```
lp.interior()  # lpx_interior
```

Get status of interior-point solution

```
lp.status    # glp_ipt_status
lp.status_i  # to force interior point status
```

Get objective value

```
lp.obj.value    # glp_ipt_obj_val
lp.obj.value_i  # to force interior point value
```

Get row or column primal or dual value

```
lp.rows[ri].primal    # glp_ipt_row_prim
lp.rows[ri].primal_i  # to force interior point value

lp.rows[ri].dual      # glp_ipt_row_dual
lp.rows[ri].dual_i    # to force interior point value

lp.cols[ci].primal    # glp_ipt_col_prim
lp.cols[ci].primal_i  # to force interior point value

lp.cols[ci].dual      # glp_ipt_col_dual
lp.cols[ci].dual_i    # to force interior point value
```

7.8 Mixed-Integer Programming Solvers

Set or get problem class

```
lp.kind  # lpx_get_class
```

Set or get column kind

```
lp.cols[ci].kind = int    # glp_set_col_kind
lp.cols[ci].kind = bool
lp.cols[ci].kind = float

lp.cols[ci].kind          # glp_get_col_kind
```

Get number of integer columns

```
lp.nint  # glp_get_num_int
```

Get number of binary columns

```
lp.nbin  # glp_get_num_bin
```

Solve MIP problem with the B&B method

```
lp.integer()  # glp_intopt
```

Solve MIP problem with the advanced B&B solver

```
lp.intopt()  # lpx_intopt
```

Get status of MIP solution

```
lp.status    # glp_mip_status
lp.status_m  # to force MIP status
```

Get objective value

```
lp.obj.value    # glp_mip_obj_val
lp.obj.value_m  # to force MIP value
```

Get row or column value

```
lp.rows[ri].value    # glp_mip_row_val
lp.rows[ri].value_m  # to force MIP value

lp.cols[ci].value    # glp_mip_col_val
lp.cols[ci].value_m  # to force MIP value
```

Check solution's integer feasibility conditions

```
lp.kktint()  # lpx_check_int
```

7.9 MIP Branch & Cut Advanced Interface

Access the problem object

```
tree.lp  # glp_ios_get_prob
```

Determine the size of the branch and bound tree

```
tree.num_active  # glp_ios_tree_size
tree.num_all     # tree.num_total
```

Determine current active subproblem

```
tree.curr_node  # glp_ios_curr_node
```

Determine first active subproblem

```
tree.first_node  # glp_ios_next_node
```

Determine last active subproblem

```
tree.last_node  # glp_ios_prev_node
```

Determine next active subproblem

```
node.next  # glp_ios_next_node
```

Determine previous active subproblem

```
node.prev  # glp_ios_prev_node
```

Determine parent subproblem

```
node.up  # glp_ios_up_node
```

Determine subproblem level

```
node.level  # glp_ios_node_level
```

Determine subproblem local bound

```
node.bound  # glp_ios_node_bound
```

Find active subproblem with best local bound

```
tree.best_node  # glp_ios_best_node
```

Compute relative MIP gap

```
tree.gap  # glp_ios_mip_gap
```

Select subproblem to continue the search

```
tree.select(node)  # glp_ios_select_node
```

Provide solution found by heuristic

```
tree.heuristic(values)  # glp_ios_heur_sol
```

Check if can branch upon specified variable

```
tree.can_branch(colnum)  # glp_ios_can_branch
```

Choose variable to branch upon

```
tree.branch_upon(colnum, 'D')  # glp_ios_branch_upon
```

Terminate the solution process

```
tree.terminate()  # glp_ios_terminate
```

7.10 Environment

Get GLPK version

```
glpk.env.version  # glp_version
```

Monitor memory usage

```
glpk.env.blocks       # glp_mem_usage
glpk.env.blocks_peak
glpk.env.bytes
glpk.env.bytes_peak
```

Limit memory usage

```
# max_megabytes should be an integer
glpk.env.mem_limit = max_megabytes  # glp_mem_limit
```

Control terminal output

```
glpk.env.term_on = True           # glp_term_out
glpk.env.term_hook = output_func  # glp_term_hook
```

7.11 Problem readers

Read fixed MPS format file

```
lp = glpk.LPX(mps=filename)  # lpx_read_mps
```

Read free MPS format file

```
lp = glpk.LPX(freemps=filename)  # lpx_read_freemps
```

Read GNU LP format file

```
lp = glpk.LPX(glp=filename)  # lpx_read_prob
```

Read CPLEX LP format file

```
lp = glpk.LPX(cpxlp=filename)  # lpx_read_cpxlp
```

Read GNU MathProg model file

```
lp = glpk.LPX(gmp=filename)  # lpx_read_model
lp = glpk.LPX(gmp=(model_file, data_file, output_file))
```

7.12 Problem and data writers

Write problem to fixed MPS format file

```
lp.write(mps=filename)  # lpx_write_mps
```

Write problem to free MPS format file

```
lp.write(freemps=filename)  # lpx_write_freemps
```

Write problem to GNU LP format file

```
lp.write(glp=filename)  # lpx_write_prob
```

Write problem to CPLEX LP format file

```
lp.write(cpxlp=filename)  # lpx_write_cpxlp
```

Write LP basis to fixed MPS format file

```
lp.write(bas=filename)  # lpx_write_bas
```

Write problem to plain text file

```
lp.write(prob=filename)  # lpx_print_prob
```

Write basic solution to plain text file

```
lp.write(sol=filename)  # lpx_print_sol
```

Write bounds sensitivity information to plain text file

```
lp.write(sens_bnds=filename)  # lpx_print_sens_bnds
```

Write interior point solution to plain text file

```
lp.write(ips=filename)  # lpx_print_ips
```

Write MIP solution to plain text file

```
lp.write(mip=filename)  # lpx_print_mip
```

CHAPTER

EIGHT

OBJECT DOCUMENTATION

This document explains the function of the classes defined in PyGLPK. In addition to descriptions, for the benefit of those familiar with the C API, it proceeds by example and analogy to the C API: there is often some C code in one color followed by some Python code in a different color which tries to do the same thing, in this spirit:

```
printf("Hello world!");
total = 0;
for (i=0; i<10; ++i) total += i*i;
```

```
print "Hello world!"
total = sum(i*i for i in range(10))
```

Hopefully, in the case of multiline examples, it should be obvious which lines correspond to each other.

Reading this document through is not necessarily the best way to gain an understanding of PyGLPK if you are completely new to it. This is organized in order of object above any other consideration: advanced obscure subjects relating to LPX objects occur before simple functionality of row and column objects. Rather, this is intended to be a reference.

8.1 Linear Program

The most basic object in PyGLPK is the LPX class, just as in the GLPK C API it is the LPX structure. As a rule, it holds data and methods that are relevant to the linear problem as a whole. Throughout the code in this document, the token `lp` refers to an LPX object.

8.1.1 Creating, Erasing, and Deleting Problems

One may create the Python instance of an LPX problem with a constructor. An LPX that has been modified can be restored to its original pristine state with a call to the `erase` method. One may explicitly delete the object. Note that one does not need to delete the LPX object as it should be automatically garbage collected, though it may be desirable to do so, just as it is sometimes desirable to free a Python object without waiting until it falls out of scope.

```
lp = glp_create_prob();
glp_erase_prob(lp);
glp_delete_prob(lp);
```

```
lp = glpk.LPX()
lp.erase()
del(lp)
```

If you have maintained objects which point back to the LPX object (e.g., rows, columns, objective function), the LPX can be deleted, but the underlying object shall not be freed until these references are discarded. In other words, there is nothing special about deleting the object versus any other Python object. The point is that `del(lp)` will not necessarily result in deallocation of the LPX structure.

8.1.2 Naming Problems

One gets and sets the name for a problem by querying or assigning to the `name` attribute. As with the C API, names are limited to 255 characters. Unset names are `None`. One unsets a name by assigning `None` or deleting the `name`.

```
glp_set_prob_name(lp, "some name");
glp_set_prob_name(lp, NULL);
char *thename = glp_get_prob_name(lp);
```

```
lp.name = "some name"
del(lp.name)  # Alternately, lp.name = None
thename = lp.name
```

8.1.3 Constraint Matrix

One gets and sets the entries of the entire constraint matrix by querying or assigning the attribute `matrix`. Assignments to `matrix` will remove previous entries.

Retrieving `matrix` will yeild a list of three element `(int, int, float)` tuples over the non-zero entries in this matrix, each of the form `(ri, ci, value)`, indicating that row `ri` and column `ci` (counting from 0) hold value `value`.

For example, consider if `lp` encoded the constraint matrix:

$$p = x_0 + x_1 + x_2$$
$$q = 10x_0 + 4x_1 + 5x_2$$
$$r = 2x_0 + 6x_2$$

Then `print lp.matrix` outputs `[(0, 0, 1.0), (0, 1, 1.0), (0, 2, 1.0), (1, 0, 10.0), (1, 1, 4.0), (1, 2, 5.0), (2, 0, 2.0), (2, 2, 6.0)]`.

For setting rather than getting, one may set all non-zero entries of the constraint matrix by assigning an iterable with similar structure to the `matrix` attribute. The iterable must yield values each in one of these two forms:

- The integer-integer-float tuple `(ri, ci, value)` where `index` >= 0 specifies that element `index` should have value `value` (negative indices are permitted in this context if you like)
- The single float item `value` which specifies an object equivalent to `(ri, ci+1, value)` (or `(ri+1, 0, value)` if `ci+1` goes past the end of the column) where `ri, ci` was the last location considered. If this single-value form is used on the first entry, the location 0, 0 is assumed.

Indices out of bounds will result in an `IndexError` and duplicate indices will result in an `ValueError`. Order does not matter, except of course for single value entries, as their location depends on the previous entry.

One may set all entries of a row or column in the constraint matrix to zero by assigning `None` to or deleting the `matrix` attribute.

Suppose we wanted to set rather than get the earlier matrix.

```
int    ia[] = {0+1, 0+1, 0+1,  1+1, 2+1, 1+1, 1+1, 2+1};
int    ja[] = {0+1, 1+1, 2+1,  0+1, 0+1, 1+1, 2+1, 2+1};
double ar[] = {1.0, 1.0, 1.0, 10.0, 2.0, 4.0, 5.0, 6.0}
glp_load_matrix(lp, sizeof(ia), ia, ja, ar);
```

```
lp.matrix = [(0, 0, 1.0), (0, 1, 1.0), (0, 2, 1.0), (1, 0, 10.0),
             (2, 0, 2.0), (1, 1, 4.0), (1, 2, 5.0), (2, 2, 6.0)]
```

One could also do the following.

```
lp.matrix = [ 1.0, 1.0, 1.0,
              10.0, 4.0, 5.0,
              2.0, 0.0, 6.0 ]
```

8.1.4 Non-Zero Constraint Matrix Entries

One gets the number of non-zero constraint matrix entries by querying the `nnz` integer attribute.

```
int numnonzero = glp_get_num_nz(lp);
```

```
numnonzero = lp.nnz
```

8.1.5 Basis Definition

The user may want to define the initial LP basis prior to starting simplex optimization. There are several automatic ways of constructing this basis.

- `std_basis` method constructs a trivial LP basis.
- `adv_basis` method constructs an advanced LP basis that tries to have as few fixed variables as possible while maintaining the triangularity of the basis matrix.
- `cpx_basis` method constructs an advanced LP basis as described in R. Bixby. "Implementing the Simplex method: The initial basis." *ORSA Journal on Computing*, 4(3), 1992.
- `read_basis` reads a basis stored in the fixed MPS file format from a given file name. If this method fails, it throws a `RuntimeError`.

```
glp_std_basis(lp);
glp_adv_basis(lp, 0);
glp_cpx_basis(lp);
lpx_read_basis(lp, "/path/to/file");
```

```
lp.std_basis()
lp.adv_basis()
lp.cpx_basis()
lp.read_basis("/path/to/file")
```

8.1.6 Scaling

Prior to optimization, it is often help to scale your problem, in part to avoid numerical instability. The method `scale` tells the linear program to transform the program into an alternate equivalent formulation with better numerical properties. **Note that this transformation is transparent to the user.** This is a matter of internal representation used to help the solver. This procedure obeys the following flags defined as integers in the LPX class, which can be ORed together to produce a combination of effects:

SF_GM
perform geometric mean scaling

SF_EQ
perform equilibration scaling

SF_2N
round scale factors to the nearest power of two

SF_SKIP
skip scaling, if the problem is well scaled

SF_AUTO
choose scaling options automatically

By using the `unscale` method, one can cancel any previous scaling.

```
glp_scale_prob(lp, GLP_SF_AUTO);
glp_scale_prob(lp, GLP_SF_GM | GLP_SF_2N);
glp_unscale_prob(lp);
```

```
lp.scale()
lp.scale(LPX.SF_GM | LPX.SF_2N)
lp.unscale()
```

8.1.7 Problem Kind, Continuous or Mixed Integer

One gets the kind of problem (the default linear program or mixed integer) by querying the attribute `kind`. This will hold either `float` if this is a pure linear program (LP), or `int` if this is a mixed integer program (MIP) by having any integer or binary column variables. A linear program becomes a mixed integer program by having some of its columns <a href="#rc_mip">assigned to either `bin` or `int` kind.

```
int thekind = lpx_get_class(lp);
```

```
thekind = lp.kind
```

8.1.8 Integer and Binary Columns

One gets the number of integer and binary (i.e., integer with 0, 1 bounds) column variables by querying the `nint` and `nbin` integer attributes, respectively. If this is not a mixed integer problem, these attributes always hold 0.

```
int num_int = glp_get_num_int(lp);
int num_bin = glp_get_num_bin(lp);
```

```
num_int = lp.nint
num_bin = lp.nbin
```

8.1.9 Solving the Problem

When it comes time to actually solving a linear program, one calls a `lp.solver()` method, where `solver` refers to one of several solver methods. There are several choices available.

- `simplex` is a standard simplex method.
- `exact` is an `exact<http://gmplib.org/>`_ simplex method.
- `interior` is an interior point method.
- `integer` is a method that uses a branch-and-bound based method to solve a mixed integer program (MIP). This method requires an existing optimal basic solution as acquired through either `simplex()` or `exact()`.
- `intopt` is a more advanced branch-and-bound MIP solver. This does not require an existing optimal basic solution.

Return values are either `None` if the solver terminated normally, or a string denoting one of several possible error messages. See help for each method to review these possible return values.

Some of these solver routines may accept additional keyword parameters to control the behavior of the underlying solver. See help for each method to review possible control parameters and default values.

```
glp_simplex(lp, params);
lpx_exact(lp);
lpx_interior(lp);
glp_intopt(lp, params);  // These two may be applied only to MIP problems
lpx_intopt(lp);
```

```
lp.simplex()
lp.exact()
lp.interior()
lp.integer()  # These two may be applied only to MIP problems
lp.intopt()
```

Note, the solver not returning a message simply means that it terminated without error. **It does not mean that an optimal solution or indeed any solution was found!** For example, a solver could terminate without error if it determines that there is no feasible solution.

8.1.10 Solution Status

One gets the solution status for the last solver by querying the `status` attribute. This takes the form of a string with several possible values.

- `opt` meaning the solution is optimal.
- `undef` meaning the solution is undefined.
- `feas` meaning the solution is feasible, but not necessarily optimal.
- `infeas` meaning the solution is infeasible.
- `nofeas` meaning the problem has no feasible solution.
- `unbnd` meaning the problem has an unbounded solution.

```
int stat = glp_get_status(lp);  // or glp_(ipt|mip)_status
```

```
stat = lp.status
```

Unlike the C API, PyGLPK remembers which solver was used last and retrieves the corresponding status value. If for whatever reason you wish to retrieve the status of a solver's solution other than what was used last, you may ask for `status_s` (for `simplex` and `exact`), or `status_i` (for `interior`), or `status_m` (for `integer` or `intopt`).

Additionally, if one has used the simplex solver, one can get the primal and dual status with the `status_primal` and `status_dual` attributes.

```
int pstat = glp_get_prim_stat(lp);
int dstat = glp_get_dual_stat(lp);
```

```
pstat = lp.status_primal
dstat = lp.status_dual
```

8.1.11 Ray

If, after running a simplex optimizer, your basic solution is unbounded, you may retrieve the row or column corresponding to the non-basic variable causing primal unboundedness within the attribute `ray`. The meaning of this is that corresponding variable is able to infinitely change in some unbounded direction to improve the objective function.

```
int the_var_index = lpx_get_ray_info(lp);
```

```
row_or_col = lp.ray
```

8.1.12 File Input

In addition to programmatically defining a linear problem, there are methods to read linear programs and MIPs from files. We have seen the empty LPX constructor employed to create an empty problem. The LPX constructor also has the ability to accept a single keyword argument: the keywords specifies a file format, and the argument specifies the filename, as in `lp=glpk.LPX(format=filename)`. If successfully read, an LPX instance will be created with the file data.

All formats accept a single string representing the path to the file to be read. Valid formats include the following.

- `gmp` for reading a model and a data file in the GNU MathProg modeling language
- `mps` for reading a fixed **`MPS<http://en.wikipedia.org/wiki/MPS_(format)>`_** formatted files
- `freemps` for reading a free MPS formatted file
- `cpxlp` for reading a CPLEX LP formatted file
- `glp` for reading a GNU LP formatted file

The format `gmp` (GNU MathProg), in addition to accepting a single string argument, may optionally accept a three element tuple instead, containing these elements:

- A file name argument specifying the GMP model file.
- A file name argument specifying the GMP data file. This may optionally be `None` if the data is included in the model file.
- A file name argument specifying the output file, where the output of any "display" statements in the GMP are output. This may optionally be `None` to send output to standard output.

For the `gmp` option, if you input a single string `filename` instead of a tuple, it is equivalent to inputing the tuple `(filename, None, None)`.

```
lp = lpx_read_mps("/path/to/mps_file")
lp = lpx_read_freemps("/path/to/free_mps_file")
lp = lpx_read_cpxlp("/path/to/cplexlp_file")
lp = lpx_read_model("modelfile", NULL, NULL)
lp = lpx_read_model("modelfile", "datafile", "output.txt")
lp = lpx_read_prob("/path/to/gnulp_file")
```

```
lp = glpk.LPX(mps="/path/to/mps_file")
lp = glpk.LPX(freemps="/path/to/free_mps_file")
lp = glpk.LPX(cpxlp="/path/to/cplexlp_file")
lp = glpk.LPX(gmp="modelfile")
lp = glpk.LPX(gmp=("modelfile", "datafile", "output.txt"))
lp = glpk.LPX(glp="/path/to/gnulp_file")
```

8.1.13 File Output

One may export data about a linear program to a file in a variety of formats conveying a variety of different types of information using the method `write`. The method accepts a large number of keyword arguments: each keyword specifies a file format, and the argument a file name, as in `lp.write(format=filename)`. Upon invocation, the LPX object will attempt to write the data specified by the format into the indicated file.

Valid formats include the following.

- `mps` for problem data in the fixed MPS format.
- `bas` for the LP basis in fixed MPS format.
- `freemps` for problem data in the free MPS format.
- `cpxlp` for problem data in the CPLEX LP format.
- `glp` for problem data in the GNU LP format.
- `prob` for problem data in a plain text format.
- `sol` for basic solution in printable format.
- `sens_bnds` for bounds sensitivity information.
- `ips` for interior-point solution in printable format.
- `mip` for MIP solution in printable format.

Note that you can specify multiple formats and output files in a single call to `write` in order to write multiple files in multiple formats in one go. For example, you might want to simultaneously write out printable problem data, solutions, and bounds sensitivity information all in one go with something like `lp.write(prob="foo.prob", sol="foo.sol", sens_bnds="foo.bnds")` .

```
lpx_write_mps(lp, filename)
lpx_write_bas(lp, filename)
lpx_write_freemps(lp, filename)
lpx_write_prob(lp, filename)
lpx_write_cpxlp(lp, filename)
lpx_print_prob(lp, filename)
lpx_print_sol(lp, filename)
```

(continues on next page)

(continued from previous page)

```
lpx_print_sens_bnds(lp, filename)
lpx_print_ips(lp, filename)
lpx_print_mip(lp, filename)
```

```
lp.write(mps=filename)
lp.write(bas=filename)
lp.write(freemps=filename)
lp.write(glp=filename)
lp.write(cpxlp=filename)
lp.write(prob=filename)
lp.write(sol=filename)
lp.write(sens_bnds=filename)
lp.write(ips=filename)
lp.write(mip=filename)
```

8.2 Objective Function

A linear program objective function specifies what linear function the LP is attempting to either minimize or maximize. Correspondingly, the objective object allows one to set objective function coefficients and the direction of optimization, and retrieve the objection function value after optimization.

The objective function for an LPX object `lp` is contained within `lp.obj`. This objects is an instance of the `Objective` class. Through this object one can set the objective coefficients and retrieve the objective value.

8.2.1 Naming Objective Function

Similar to how one names problems, one gets and sets the name for the objective function by querying or assigning to the `name` attribute. As with the C API, names are limited to 255 characters. Unset names are `None`. One unsets a name by assigning `None` or deleting the `name`.

```
glp_set_obj_name(lp, "some name");
glp_set_obj_name(lp, NULL);
char *thename = glp_get_obj_name(lp);
```

```
lp.obj.name = "some name"
del lp.obj.name
thename = lp.obj.name
```

8.2.2 Minimize or Maximize

One gets and sets whether this is a minimization or maximization problem by querying or assigning to the `maximize` boolean attribute.

```
glp_set_obj_dir(lp, GLP_MIN);
glp_set_obj_dir(lp, GLP_MAX);
int ismax = (glp_get_obj_dir(lp) == GLP_MAX);
```

```
lp.obj.maximize = False
lp.obj.maximize = True
ismax = lp.obj.maximize
```

8.2.3 Objective Coefficients

One gets and sets the objective function coefficients by indexing into the `obj` object, e.g., `lp.obj[index]`. There are as many objective coefficients as there are columns, so valid indices include `0` through `len(lp.cols)-1` as well as (for negative indexing) `-1` through `-len(lp.cols)`.

One can access and change these objective coefficients through either a single index, or access or change multiple coefficients by defining multiple indices through either a series of indices or a slice.

When assigning new objective coefficients, valid assignments include single numbers (in which case all indexed coefficients receive this same value) or an iterable object (in which case all indexed coefficients receive values specified in turn).

The objective function's constant shift term can be accessed either by using `None` as an index, or by accessing the `shift` attribute, that is, `lp.obj.shift`.

```
glp_set_obj_coef(lp, 2+1, 3.0);
for (i=0; i&lt;glp_get_num_cols(lp); ++i)
    glp_set_obj_coef(lp, i+1, 1.0)
glp_set_obj_coef(lp, 0+1, 3.14159); glp_set_obj_coef(lp, 2+1, -2.0);
glp_set_obj_coef(lp, 0, 0.5);
glp_set_obj_coef(lp, glp_get_num_cols(lp), 25.0);
double c = glp_get_obj_coef(lp, 3+1);
double c1 = glp_get_obj_coef(lp, 1+1), c2 = glp_get_obj_coef(lp, 2+1);
```

```
lp.obj[2] = 3.0
lp.obj[:] = 1.0
lp.obj[0,2] = 3.14159, -2.0
lp.obj.shift = 0.5  # Alternately, lp.obj[None] = 0.5
lp.obj[-1] = 25.0
c = lp.obj[3]
c1, c2 = lp.obj[1,2]
```

8.2.4 Objective Function Value

One gets the value for the objective function by querying the `value` attribute.

```
double oval = glp_get_obj_val(lp); //  or glp_(ipt|mip)_obj_val
```

```
oval = lp.obj.value
```

Unlike the C API, PyGLPK remembers which solver was used last and retrieves the corresponding objective function value. If for whatever reason you wish to retrieve an objective function from a solver type different from what you used last, you can force the issue by asking for `value_s` (for `simplex` and `exact`), or `value_i` (for `interior`), or `value_m` (for `integer` or `intopt`).

```
double soval = glp_get_obj_value(lp);
double ioval = glp_ipt_obj_value(lp);
double moval = glp_mip_obj_value(lp);
```

```
soval = lp.obj.value_s
ioval = lp.obj.value_i
moval = lp.obj.value_m
```

8.3 Rows and Columns

In a linear program, rows and columns correspond to variables. Correspondingly, individual rows and column objects contain methods and data pertaining to individual variables: bounds, values after optimization, status, relevant entries of the constraint matrix, and other such objects.

Rows and columns live all live within two objects stored within an LPX object `lp` as `lp.rows` and `lp.cols`. Both of these objects is an instance of the `BarCollection` class. Individual rows and columns, all of type `Bar`, can be accessed by indexing or iteration over these collections.

8.3.1 Adding Rows and Columns

To add rows or columns, call the `add` method on either the `row` or `column` subcontainer. As in the C API, the newly created rows and columns are initially empty, and the return value of the `add` method holds the first newly valid index.

```
int rnew = glp_add_rows(lp, nrs);
int cnew = glp_add_cols(lp, ncs);
```

```
rnew = lp.rows.add(nrs)
cnew = lp.cols.add(ncs)
```

8.3.2 Number of Rows and Columns

One gets the number of rows or columns by querying the length of the LP's `row` and `column` containers.

```
int nrs = glp_get_num_rows(lp);
int ncs = glp_get_num_cols(lp);
```

```
nrs = len(lp.rows)
ncs = len(lp.cols)
```

8.3.3 Indexing Rows and Columns

One accesses particular rows and columns by indexing into the `lp.rows` and `lp.cols` collections. For example, `lp.rows[ri]` returns the row at index `ri`. This index may also be a negative index counting backwards from the end of the collection, e.g., `lp.cols[-1]` to get the last column of the LP.

These structures adopt much of the familiar behavior of Python sequences. Among other implications, this means that unlike in the C API, rows and columns are indexed from 0.

As we shall see, rows and columns can be named. One may also index named rows and columns by their names.

```
int rownum = glp_find_row(lp, "rowname")
```

```
row = lp.rows["rowname"]
```

In addition to single integer or string values, one may specify multiple values in this index to retrieve a list of all specified rows or columns.

```
lp.cols[2,5,"bob",6]  # columns 2, 5, one named "bob", 8
```

Indexing by slicing is supported as well. This will result in a list of all indices specified by the slice.

```
lp.rows[4:9]  # rows 4 through 8
lp.cols[-3:]  # the last 3 columns
lp.rows[::2]  # every row with an even index
```

One may also iterate over the `lp.rows` and `lp.cols` collections. Here is a comparative example of setting each column to name `"x%d"` so the columns will be named `x0`, `x1`, `x2`, etc.

```
char buff[10];
for (i=1; i<=glp_get_num_cols(lp); ++i) {
    snprintf(buff, sizeof(buff), "x%d", i-1);
    glp_set_col_name(lp, i, buff);
}
```

```
for col in lp.cols:
    col.name = "x%d" % col.index
```

8.3.4 Naming Rows and Columns

As with the problem and the objective function, one gets and sets the name for a row or column by querying or assigning the attribute `name`.

Note the use of an index into the `rows` or `cols` collections to retrieve a particular row or columns. As with the C API, indices are integral, though we count from 0.

```
glp_set_row_name(lp, ri+1, "row name");
glp_set_col_name(lp, ci+1, "col name");
char *rname = glp_get_row_name(lp, ri+1);
char *cname = glp_get_col_name(lp, ci+1);
```

```
lp.rows[ri].name = "row name"
lp.cols[ci].name = "col name"
```

(continues on next page)

(continued from previous page)

```
rname = lp.rows[ri].name
cname = lp.cols[ci].name
```

After the user names a row or column, they may index this row or column by its name.

```
lp.rows[ri].name = "xi"
therow = lp.rows["xi"]
```

8.3.5 Bounding Rows and Columns

One gets and sets the bounds for a row or column by querying or assigning the attribute `bounds`. To set bounds, one may assign one or two values to the `bounds`, where values are either `None` or numeric.

One `None` (or two `None``s) sets the row's auxiliary (or column's structural) variable unbounded. (One may also delete the bounds.) One numeric value (or two equal numeric values) sets an equality bound. In the case of two values, the first is interpreted as a lower bound, the second as an upper bound, with ``None` indicating unboundedness in that direction. Setting a lower bound greater than an upper bound causes a `ValueError`.

In this code, we see instances of setting free (unbounded), lower, upper, double, and fixed (equality) bounds, respectively on a row and column.

```
glp_set_row_bnds(lp, ri+1, GLP_FR,  0,   0);
glp_set_row_bnds(lp, ri+1, GLP_LO,  2,   0);
glp_set_col_bnds(lp, ci+1, GLP_UP,  0,   5);
glp_set_col_bnds(lp, ci+1, GLP_DB, -1,   3.14159);
glp_set_row_bnds(lp, ri+1, GLP_FX,  3.4, 3.4);
```

```
lp.rows[ri].bounds = None     # Or, lp.rows[ri].bounds = None, None
                              # Or, del lp.rows[ri].bounds
lp.rows[ri].bounds = 2, None
lp.cols[ci].bounds = None, 5
lp.cols[ci].bounds = -1, 3.14159
lp.rows[ri].bounds = 3.4      # Or, lp.rows[ri].bounds = 3.4, 3.4
```

Accessing bounds always yields two values (again, either `None` or numeric) representing lower and upper bounds respectively, even if the bounds resulted from either a single value assignment or a deletion. Again, `None` represents unboundedness in that direction.

8.3.6 Matrix Entries of Rows and Columns

One gets and sets the entries of a row or column in the constraint matrix by querying or assigning the attribute `matrix`. Assignments to `matrix` will remove previous entries.

Retrieving `matrix` will yield a list of two element tuples over the non-zero entries in this row or column, each of the form `(index, value)`. The `index` is the index (counting from 0) of the entry holding value `value` in this row or column.

If we have an LPX object with r rows and c columns, then valid indices for rows are 0 through $c - 1$ and valid entries for columns are 0 through $r - 1$.

For example, suppose for an object `lp` the row r_2 (that is, row at index 2) encodes the constraint:

$$p = 10x_0 13.14159x_1 + 0.5x_3$$

Then `print lp.rows[2].matrix` outputs `[(0, 10.0), (1, -3.14159), (3, 0.5)]`.

One may set all non-zero entries of a row or column by assigning an iterable with similar structure to the `matrix` attribute. Suppose our LPX object has $numr$ rows and $numc$ columns. The iterable must yield values each in one of these two forms:

- The integer-float tuple `(index, value)` which specifies that element `index` should have value `value` (note that negative indices are permitted in this context if you like)
- The single float item `value` which specifies an object equivalent to `(index+1, value)` where `index` was the last index used in this iterable, or 0 if this is the first object in the iterable

For example, where one interested in defining (rather than simply retrieving) the entries of the constraint row used in the example above, if there are four columns, all of the following are equivalent:

```
# Define constraint p = 10*x0 - 3.14159*x1 + 0.5*x3
lp.rows[2].matrix = [(0, 10), (1, -3.14159), (3, 0.5)]
lp.rows[2].matrix = [(0, 10), (1, -3.14159), (-1, 0.5)]
lp.rows[2].matrix = [(1, -3.14159), (0, 10), (3, 0.5)]
lp.rows[2].matrix = [10, -3.14159, 0, 0.5]
lp.rows[2].matrix = [10, -3.14159, (-1, 0.5)]
lp.rows[2].matrix = [10, -3.14159, (3, 0.5)]
```

Indices out of bounds will result in an `IndexError` and duplicate indices will result in an `ValueError`. Order does not matter, except of course for single value entries, as their index depends on the previous entry.

One may set all entries of a row or column in the constraint matrix to zero by assigning `None` to or deleting the `matrix` attribute.

8.3.7 Number of Non-Zero Constraint Row and Column Entries

One gets the number of non-zero constraint elements within a row or a column by querying the `nnz` integer attribute.

```
int rnnz = glp_get_mat_row(lp, ri+1, NULL, NULL);
int cnnz = glp_get_mat_col(lp, ci+1, NULL, NULL);
```

```
rnnz = lp.row[ri].nnz
cnnz = lp.col[ci].nnz
```

8.3.8 Deleting Rows and Columns

To delete rows or columns, delete as one would from a typical Python list. Note the methods of indexing into the row and column collections. Accepted indices include single values, lists of values, or slices.

```
int indices1[] = { 2+1 };
glp_del_cols(lp, 1, indices1-1);
int indices2[] = { 2+1,5+1,6+1 };
glp_del_rows(lp, 3, indices2-1);
int indices3[] = { 3+1, 4+1, 5+1, 6+1 };
glp_del_cols(lp, 4, indices3-1);
```

```
del lp.cols[2]      # Remove col indexed at 2
del lp.rows[2,5,6]  # Remove rows indexed at 2,5,6
del lp.cols[3:7]    # Remove cols indexed at 3,4,5,6
```

8.3.9 Row and Column Scaling Factors

The constraint matrix A undergoes a linear transformation with diagonal positive matrices R and S (row and column scaling matrices, respectively) to come up with an implicit new constraint matrix $\tilde{A} = RAS$. The transformed matrix has entries $\tilde{a}_{ij} = r_{ii}a_{ij}s_{jj}$. Though most users may wish to set this <a href="#lp_scale">scaling automatically, one may set and get the row and column scaling factors manually with the `scale` attribute. Changing the scaling factor for row i or column j corresponds to changing element r_{ii} or s_{jj} in the diagonal scaling matrices, respectively.

```
glp_set_rii(lp, 2, 3.14159);
glp_set_sjj(lp, 4, 2.0);
double row3scale = glp_get_rii(lp, 3);
double col3scale = glp_get_sjj(lp, 3);
```

```
lp.rows[2].scale = 3.14159
lp.cols[4].scale = 2.0
row3scale = lp.rows[3].scale
col3scale = lp.cols[3].scale
```

8.3.10 Special Attributes of Rows and Columns

As is clear from the previous examples, row and column collections (e.g., as accessed by `lp.rows`) and rows and columns (e.g., as accessed by `lp.rows[i]`) as well as the rows and columns themselves are bona fide objects. (This was a design choice: rather than having only the LPX class where one defines a hundred or so get and set methods, as the C API must, one retrieves the rows and columns and operates on them instead.)

As a point of implementation, these row and column objects do not contain the row and column data. In reality, they just contain a pointer back to the LPX and an index. We shall see consequences of this in this subsection.

Aside from attributes which have obvious analogies to functions in the C API (e.g., `name` with the `glp_[gs]et_(row|col)_name` functions), rows and columns have other special attributes that do not have analogies in the C API which are exposed to Python users in the hope they may find them useful.

`index` is an integer attribute containing the index of this row or column.

`valid` is a boolean attribute containing whether this row or column is valid. A row or column may become invalid if its index points to somewhere beyond the current size of the LPX. This is mostly useless: one can track the size of the program, and even if you do not, using an out of date row or column safely throws exceptions.

`isrow` and `iscol` are boolean attributes indicating whether this is a row or a column. Naturally these two attributes are inverses of each other.

Example usage of these principles to elucidate these implementations is illustrated in this example. All assertions in this snippet are satisfied.

```
lp = glpk.LPX()
lp.rows.add(3)
lp.rows[0].name, lp.rows[1].name, lp.rows[2].name = 'p', 'q', 'r'
row1, row2 = lp.rows[1], lp.rows[2]
assert row1.name == 'q' and row2.name == 'r'
del lp.rows[1]
assert row1.name == 'r' and row1.valid and not row2.valid
assert row1.isrow
```

8.3.11 Row and Column Basis Status

One gets and sets the current basis status for a row or column by querying or assigning the attribute `status`. This is a two-character string with the following possible values.

- `bs` meaning this row/column is basic.
- `nl` meaning this row/column is non-basic.
- `nu` meaning this row/column is non-basic and set to the upper bound. On assignment, if this row/column is not double bounded, this is equivalent to `nl`.
- `nf` meaning this row/column is non-basic and free. On assignment this is equivalent to `nl`.
- `ns` meaning this row/column is non-basic and fixed. On assignment this is equivalent to `nl`.

```
if (glp_get_row_stat(lp, ri+1) == GLP_BS)
    printf("row is basic\n");
else
    printf("row is non-basic\n");
```

```
if lp.rows[ri].status == "bs":
    print "row is basic"
else:
    print "row is non-basic"
```

As an example of setting the status, the user may wish to assign to this attribute in order to manually define the initial basis and not rely upon the automatic basis definition methods `lp.*_basis()`. To illustrate this, here is the code within the GLPK standard basis code in both C and Python versions.

```
int i, j, m, n, type;
double lb, ub;
// all auxiliary variables are basic
m = glp_get_num_rows(lp);
for (i = 1; i <= m; i++)
    glp_set_row_stat(lp, i, GLP_BS);
// all structural variables are non-basic
n = glp_get_num_cols(lp);
for (j = 1; j <= n; j++) {
    type = glp_get_col_type(lp, j);
    lb = glp_get_col_lb(lp, j);
    ub = glp_get_col_ub(lp, j);
    if (type != GLP_DB || fabs(lb) <= fabs(ub))
        glp_set_col_stat(lp, j, GLP_NL);
    else
        glp_set_col_stat(lp, j, GLP_NU);
}
```

```
# all auxiliary variables are basic
for row in lp.rows:
    row.status = "bs"
# all structural variables are non-basic
for col in lp.cols:
    lb, ub = col.bounds
    if lb==None or ub==None or abs(lb)<=abs(ub):
```

(continues on next page)

(continued from previous page)

```
        col.status = "nl"
    else:
        col.status = "nu"
```

8.3.12 Column Variable Kind, Continuous, Integer, or Binary

One gets and sets the kind of variable (the default continuous, or integer) by querying or assigning the attribute `kind`. This will hold either `float` if this is a continuous variable, `int` if this is an integer variable, or `bool` if this is a binary variable.

```
glp_set_col_kind(lp, ci+1, GLP_CV);
glp_set_col_kind(lp, ci+1, GLP_IV);
glp_set_col_kind(lp, ci+1, GLP_BV);
int kind = glp_get_col_kind(lp, ci+1);
```

```
lp.cols[ci].kind = float
lp.cols[ci].kind = int
lp.cols[ci].kind = bool
kind = lp.cols[ci].kind
```

Note that PyGLPK and GLPK do not make any distinction between setting a column as binary, versus setting the column as integral with [0, 1] bounds.

Another note, rows must be continuous. As a matter of implementation, because they are the same type of object as columns, they may also be queried and assigned to in this fashion. However, their `kind` attribute always returns and only accepts `float`.

8.3.13 Row and Column Variable Values

One gets a row or column's variable value by querying the `primal`, `dual`, or `value` attribute.

```
double pval = glp_get_row_prim(ri+1); // if simplex
double dval = glp_get_col_dual(ci+1);
```

```
pval = lp.rows[ri].primal
dval = lp.cols[ci].dual
```

The two attributes `primal` and `value` are interchangeable. The term value is used to refer to the solutions of the MIP since it only has one type of value (no dual). However, since it is the same type of value (loosely speaking), this "link" between the two was established.

Note that unlike the C API, this remembers which solver was used last and retrieves the appropriate corresponding variable value. If for whatever reason you wish to retrieve a variable function from a solver type different from what you used last (e.g., reviewing the relaxed basic solution after calling `integer()`), you can ask for `primal_s` or `dual_s` (for primals and duals from `simplex` and `exact`), or `primal_i` or `dual_i` (for primals and duals from `interior`), or `value_m` (for values from `integer` or `intopt`).

```
double prim_sim_row = glp_get_row_prim(lp, ri+1);
double prim_sim_col = glp_get_col_prim(lp, ci+1);
double dual_sim_row = glp_get_row_dual(lp, ri+1);
double dual_sim_col = glp_get_col_dual(lp, ci+1);
```

(continues on next page)

(continued from previous page)

```
double prim_ipt_row = glp_ipt_row_prim(lp, ri+1);
double prim_ipt_col = glp_ipt_col_prim(lp, ci+1);
double dual_ipt_row = glp_ipt_row_dual(lp, ri+1);
double dual_ipt_col = glp_ipt_col_dual(lp, ci+1);
double valu_mip_row = glp_mip_row_val (lp, ri+1);
double valu_mip_col = glp_mip_col_val (lp, ci+1);
```

```
prim_sim_row, prim_sim_col = lp.rows[ri].primal_s, lp.cols[ci].primal_s
dual_sim_row, dual_sim_col = lp.rows[ri].dual_s,   lp.cols[ci].dual_s
prim_ipt_row, prim_ipt_col = lp.rows[ri].primal_i, lp.cols[ci].primal_i
dual_ipt_row, dual_ipt_col = lp.rows[ri].dual_i,   lp.cols[ci].dual_i
valu_mip_row, valu_mip_col = lp.rows[ri].value_m,  lp.cols[ci].value_m
```

8.4 MIP Callbacks and Search Trees

In this section we describe the MIP solver callback interface, and the `Tree` and `TreeNode` objects supporting this interface for affecting the MIP solver.

8.4.1 Callback Objects

One of the more esoteric parts of the GLPK mixed integer programming solver is the use of callbacks to let the user code affect the flow of the search process. Within PyGLPK, one can define a callback object which will be invoked at various parts of the algorithm, through the use of the optional `callback` keyword parameter to the MIP solver, whose argument we will term `cb`:

```
lp.integer(callback=cb)
```

What this `cb` callback object is is not strictly defined, but this object `cb` should respond to calls of the form `cb.method(tree)`, where `method` is one of `select`, `prepro`, `rowgen`, `heur`, `cutgen`, `branch`, or `bingo`. These different methods represent the MIP solver seeking the callback object's input at various phases of the input. (If a method does not exist, PyGLPK will try the `default` method instead, and if that does not exist, it will ignore the callback for that method.)

The `tree` argument to these methods is a `Tree` instance, a representation of the search tree of the method. The Tree instance contains data about the problem being solved.

```
void callback_func(glp_tree *tree, void *info) {
    switch (glp_ios_reason(tree)) {
    case GLP_ISELECT: // some code to select subproblems here...
    case GLP_IPREPRO: // some code for preprocessing here...
    case GLP_IROWGEN: // some code for providing constraints here...
    case GLP_IHEUR:   // some code for providing heuristic solutions here...
    case GLP_ICUTGEN: // some code for providing constraints here...
    case GLP_IBRANCH: // some code to choose a variable to branch on here...
    case GLP_IBINGO:  // some code to monitor the situation here...
    }
}
...
glp_iocp parm;
glp_init_iocp(&parm);
```

(continues on next page)

(continued from previous page)

```
parm.cb_func = callback_func;
glp_intopt(lp, &parm);
```

```
class Callback:
    def select(self, tree):
        # some code to select subproblems here...
    def prepro(self, tree):
        # some code for preprocessing here...
    def rowgen(self, tree):
        # some code for providing constraints here...
    def heur(self, tree):
        # some code for providing heuristic solutions here...
    def cutgen(self, tree):
        # some code for providing constraints here...
    def branch(self, tree):
        # some code to choose a variable to branch on here...
    def bingo(self, tree):
        # some code to monitor the situation here...

lp.integer(callback=Callback())
```

The `Tree` instance passed along to the function contains active subproblems being searched, where each subproblem corresponds to a `TreeNode` instance.

Each of these seven phases of GLPK's implementation of the branch and cut algorithm correspond to these seven methods. By calling a particular method, the GLPK indicates that it desires some sort of input from the user. While a full description of the branch and cut algorithm is beyond the scope of this document (see the "Branch-and-cut interface routines" section of the "GNU Linear Programming Kit Reference Manual" that came with you GLPK distribution), and unfortunately a full understanding of what to do in each instance is beyond this author, we briefly describe the phases here, and what may be helpful in each instance to do what GLPK wants us to do.

Note that all operations are optional. One does not need to implement each method, and it is fine for a method to not do what is being requested: even just having a `pass` statement in a method is fine. The GLPK has default behavior for all of the methods in case the user does not choose to affect the solution process.

`select`, request for subproblem selection

There is no current node, so set one of the active subproblems as the current node with the `tree.select` method. The default behavior of the GLPK is to select the node with the best local bound, equivalent to this:

```
def select(self, tree):
    tree.select(tree.best_node)
```

`prepro`, request for preprocessing

The GLPK manual suggests that one may take advantage of this to perform preprocessing, perhaps of the form of tightening or loosening bounds of some variables, through modification of the `tree.lp` program object.

`rowgen`, request for row generation

When the current subproblem has been solved to optimality and the LP relaxation has been solved with a solution better than the best known integer feasible solution, this procedure may be called upon to add "lazy" constraints to the `tree.lp`, which is done as one normally adds rows (`tree.lp.rows.add(...)`, and so on).

`heur`, request for a heuristic solution

When the current subproblem being solved to optimality is integer infeasible (i.e., some integer problems are fractional), though with a better objective value than the best known integer solution, one may call `tree.heuristic(newsol)` where `newsol` is some iterable object (like a list, or an iterator) which can yield at least

`len(tree.lp.cols)` float values (with integral values for integral columns), to serve as the new primal values. (The method will check to see if it is better.) Note that feasibility of this solution is not checked by the method, so use caution.

cutgen, request for cut generation
Similar to `rowgen`, called when the subproblem being solved is integer infeasible but better than the best known integer solution, with the intent being that one adds constraints to cut off the current solution.

branch, request for branching
In the case of integer infeasibility, we have some integer variable (e.g., column) with non-integer value V. Branching is the process of splitting this process by adding two subproblems to the active list with the column's value set to $\lfloor V \rfloor$ and $\lceil V \rceil$. For some column index `j`, which we have confirmed we can branch upon with the `tree.can_branch` method, we call the `tree.branch` method with that index to add the two corresponding subproblems.

```
def branch(self, tree):
    # Find the first fractional integer variable, and branch on it
    for j in xrange(len(tree.lp.cols)):
        if tree.can_branch(j):
            tree.branch(j)
            break
```

bingo, better integer solution found
When the LP relaxation finds an integer feasible solution, this method is called. This is intended only for informational purposes, and should not modify any problem data.

8.4.2 Tree Linear Program

One may retrieve the `LPX` problem object used by the MIP solver with the `lp` member, i.e., `tree.lp`. This object is not necessarily the same `LPX` instance as that for which we called `lp.integer()` if some preprocessing was performed by the MIP solver.

Modification of the underlying LPX object is an important part of the callback procedures, especially in the `rowgen`, `cutgen`, and `prepro` methods. It is important to note that not all operations you may perform on LPX objects within this callback are necessarily safe: modifying the problem object out from under the procedure in the middle of optimization might cause problems. However, it is difficult to distinguish a "problematic" change versus one which is helpful for, say, preprocessing, or computing cuts, or what have you.

In order to aid ease of use, the PyGLPK implements "generic" solution retrieval methods. For instance, an LP potentially has multiple objective function values: one for the last simplex solution `lp.obj.value_s`, one for the last interior point solution `lp.obj.value_i`, and one for the last MIP solution `lp.obj.value_m`. However, there is also a `lp.obj.value`, which sensibly supposes that a user is interested in the solution the solver (which covers the vast majority of use cases). However, in the midst of integer solution, the MIP solver hasn't yet become the "last" solver, so be sure to be explicit when retrieving values of columns and objective function values.

```
glp_prob *lp = glp_ios_get_prob(tree);
```

```
lp = tree.lp
```

8.4.3 Tree Methods for Affecting Search

The `Tree` instance has many methods that allow one to affect the search process, as described earlier: `can_branch` and `branch_upon` to choose a column to set as integer, `heuristic` to set a new integer feasible solution, `select` to select an active subproblem for expansion, and `terminate` to just terminate the solution outright.

```
glp_ios_can_branch(tree, 5+1);
glp_ios_branch_upon(tree, 5+1, 'D');
double *values;
glp_ios_heur_sol(tree, values)
glp_ios_select_node(tree, node_num);
glp_ios_terminate(tree);
```

```
tree.can_branch(5)
tree.branch_upon(5, 'D')
tree.heuristic(values)
tree.select(node)
tree.terminate()
```

8.4.4 Tree Node Traversal

The tree instance contains an active subproblem list, corresponding to current entries in the search tree which have not yet been explored. The tree contains several members that let one access the tree nodes, corresponding to subproblems in the list:

- `curr_node`, the current active subproblem's node, which will be `None` in the selection phase when there is no current subproblem.
- `first_node`, the first node in the active subproblem list.
- `last_node`, the last node in the active subproblem list.
- `best_node`, the node whose active subproblem has the best local bound.

For instance:

```
int node_num;
node_num = glp_ios_curr_node(tree);
node_num = glp_ios_next_node(tree, 0);
node_num = glp_ios_prev_node(tree, 0);
node_num = glp_ios_best_node(tree);
```

```
node = tree.curr_node
node = tree.first_node
node = tree.last_node
node = tree.best_node
```

One may also iterate over `tree` to get all of the nodes in the active list.

In addition to `Tree` members, each `TreeNode` has members to connect them to other `TreeNode` objects in the tree.

- `next`, which gets the next active subproblem's node, or `None` if this is the last active node or an inactive node.
- `prev`, which gets the previous active subproblem's node, or `None` if this is the first active node or an inactive node.
- `up`, which gets the parent node that generated this node, or `None` if this is the root node.

- `level`, which is this node's distance from the root, `0` if this is the root node
- `subproblem`, which is the integral subproblem number, assigned in order from 1 onwards.

```
int other_node_num, lev;
other_node_num = glp_ios_next_node(tree, node_num);
other_node_num = glp_ios_prev_node(tree, node_num);
other_node_num = glp_ios_up_node(tree, node_num);
lev = glp_ios_node_level(tree, node_num);
// node_num is the subproblem id number
```

```
other_node = node.next
other_node = node.prev
other_node = node.up
other_node = node.level
subproblem_num = node.subproblem
```

8.4.5 Tree Size

One may get the tree size through the use of various members.

- `num_active`, the number of active nodes.
- `num_all`, the number of active and inactive nodes in the tree.
- `num_total`, the number of nodes which were generated, active, inactive, and those nodes which have already been removed.

```
int count;
glp_ios_tree_size(tree, &count, NULL, NULL);
glp_ios_tree_size(tree, NULL, &count, NULL);
glp_ios_tree_size(tree, NULL, NULL, &count);
```

```
count = tree.num_active
count = tree.num_all
count = tree.num_total
```

8.4.6 Tree MIP Gap

One may get the current relative gap between the integer and relaxed solution with the `gap` member.

```
double gap = glp_ios_mip_gap(tree);
```

```
gap = tree.gap
```

8.4.7 Tree Node Bound

One may get the lower (in minimization) or upper (in maximization) bound on the integer optimal solution to a node's subproblem with the `bound` member of a `TreeNode` instance.

```
double bound = glp_ios_node_bound(tree, node_num);
```

```
bound = node.bound
```

8.4.8 Tree Callback Reason

The `Tree` member `reason` holds a string indicating the reason why the callback was invoked. While the reason for the callback is the same as the method name called in the callback (e.g., `select` is called only if `tree.reason='select'`), if one implements the `default` method in lieu of specific methods, one may wish to extract the reason with this member.

```
int reasoncode = glp_ios_reason(tree);
```

```
reason = tree.reason
```

8.5 Environment

Contained within `glpk.env` object is an `Environment` instance, through which one controls the global behavior of the GLPK routines.

8.5.1 Version

In the environment is a tuple `version` that reflects the version of GLPK that the build process believed it was linking against at compilation time. For example, if the module believed it was linking against GLPK 4.31, the tuple would be `(4, 31)`.

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

```
print 'GLPK version is %d.%d\n' % glpk.env.version
```

8.5.2 Memory

The GLPK monitors its own memory use, and this information can be retrieved from these members of the `glpk.env` object. The `blocks` member holds the current number of allocated memory blocks, while `blocks_peak` holds the maximum this ever reached. The `bytes` and `bytes_peak` members are similar, except for bytes. Note that blocks are not a particular size, but are multiple

```
int blocks, blocks_peak;
glp_long bytes, bytes_peak;
glp_mem_usage(&blocks, &blocks_peak, &bytes, &bytes_peak);
```

```
blocks = glpk.env.blocks
blocks_peak = glpk.env.blocks_peak
bytes = glpk.env.bytes
bytes_peak = glpk.env.bytes_peak
```

One may also set the maximum number of megabytes with the `mem_limit` member.

```
glp_mem_limit(50);
```

```
glpk.env.mem_limit = 50
```

8.5.3 Terminal Output

The GLPK has many functions that produce output. The user may at their option turn off or on the output by setting the environment's `term_on` attribute to `False` or `True`.

```
glp_term_out(GLP_OFF);
glp_term_out(GLP_ON);
```

```
glpk.env.term_on = False
glpk.env.term_on = True
```

In addition to turning it on and off, one may enable more fine grained control by intercepting all terminal output with a function hook. The function will be called with a single string argument whenever the GLPK chooses to print something. Note that this will intercept only that output which is produced by the GLPK itself – other output from Python will be completely unaffected.

```
int term_hook(void *info, const char *output) {
    FILE *logfile = (FILE *)info;
    fputs(output, logfile);
    return 1;
}
...
FILE *lf = fopen("glpk_logfile.txt", "w");
glp_term_hook(term_hook, (void*)lf);
```

```
logfile = file('glpk_logfile.txt', 'w')
def term_hook(output):
    file.write(output)
glpk.env.term_hook = term_hook
```

One may remove the terminal hook function (e.g., resume default terminal output) by assigning the value `None`.

```
glp_term_hook(NULL, NULL);
```

```
glpk.env.term_hook = None
```

8.6 Karush-Kuhn-Tucker Conditions

The linear program object has the ability to return a KKT type objects with which the user may evaluate the fitness of either simplex or MIP solutions.

8.6.1 Retrieving KKT Objects

One can compute and retrieve these conditions for simplex solvers with the `kkt` and for integer solvers with the `kktint` methods. The `kkt` method has an optional argument that allows one to specify whether one wants to compute the conditions for the internally scaled version of the problem (by default false).

```
LPXKKT kkt, skkt, ikkt;
lpx_check_kkt(lp, 0,  &kkt);  // unscaled simplex KKT
lpx_check_kkt(lp, 1, &skkt);  // scaled simplex KKT
lpx_check_int(lp,    &ikkt);  // integer conditions
```

```
kkt  = lp.kkt()     # unscaled simplex KKT
skkt = lp.kkt(True) # scaled simplex KKT
ikkt = lp.kktint()  # integer conditions
```

These objects have KKT statistics about the absolute and relative errors and worst rows and columns in the primal and (in the case of non-integer problems) dual solutions. See the inline help for more information about these fields.

8.7 Miscellaneous Notes

8.7.1 Help

In addition to this documentation, like most Python objects, the objects have built in inline help, accessible from an interactive Python session. For example, to access the built in documentation for the `glpk` module:

```
help(glpk)
```

CHAPTER

NINE

REFERENCE

The PyGLPK module, encapsulating the functionality of the GNU Linear Programming Kit. Usage of this module will typically start with the initialization of an LPX instance to define a linear program, and proceed from there to add data to the problem and ultimately solve it. See help on the LPX class, as well as the HTML documentation accompanying your PyGLPK distribution.

class glpk.**Bar**

Bar objects are used to refer to a particular row or column of a linear program. Rows and columns may be retrieved by indexing into the rows and cols sequences of LPX instances.

bounds

The lower and upper bounds, where None signifies unboundedness.

dual

The dual value of this variable by the last solver.

dual_i

The dual value of this variable by the interior-point solver.

dual_s

The dual value of this variable by the simplex solver.

eval_tab_col() → [*Bar*, float, ...]

Returns the column of the current simplex tableau for this variable. The column is returned as a list of tuples containing a reference to a basic variable and the corresponding coefficient from the simplex tableau

If this variable is basic, this method throws a ValueError.

eval_tab_row() → [*Bar*, float, ...]

Returns the row of the current simplex tableau for this variable. The row is returned as a list of tuples containing a reference to a non-basic variable and the corresponding coefficient from the simplex tableau.

If this variable is non-basic, this method throws a ValueError.

index

The index of the row or column this object refers to.

iscol

Whether this is a column.

isrow

Whether this is a row.

kind

Either the type 'float' if this is a continuous variable, 'int' if this is an integer variable, or 'bool' if this is a binary variable.

matrix

Non-zero constraint coefficients in this row/column vector as a list of two-element (index, value) tuples.

name

Row/column symbolic name, or None if unset.

nnz

Number of non-zero constraint elements in this row/column.

primal

The primal value of this variable by the last solver.

primal_i

The primal value of this variable by the interior-point solver.

primal_s

The primal value of this variable by the simplex solver.

scale

The scale for the row or column. This is a factor which one may set to improve conditioning in the problem. Most users will want to use the LPX.scale() method rather than setting these directly. The resulting constraint matrix is such that the entry at row i and column j is (for the purpose of optimization) (ri)*(aij)*(sj) where ri and sj are the row and column scaling factors, and aij is the entry of the constraint matrix.

status

Row/column basis status

This is a two character string with the following possible values:

bs
This row/column is basic.

nl
This row/column is non-basic.

nu
This row/column is non-basic and set to the upper bound. On assignment, if this row/column is not double bounded, this is equivalent to 'nl'.

nf
This row/column is non-basic and free. On assignment this is equivalent to 'nl'.

ns
This row/column is non-basic and fixed. On assignment this is equivalent to 'nl'.

valid

Whether this row or column has a valid index in its LP.

value

The value of this variable by the last solver.

value_m

The value of this variable by the MIP solver.

class glpk.**BarCollection**

Bar collection objects

An instance is used to index into either the rows and columns of a linear program. They exist as the 'rows' and 'cols' attributes of LPX instances.

One accesses particular rows or columns by indexing the appropriate bar collection object, or iterating over it. Valid indices include particular row and column names (a user defined string) or numbers (counting from 0), a slice specifying a range of numeric elements, or a series of individual indices. For example, for an LPX instance lp, we may have:

```
lp.rows[0]          # the first row
lp.rows[-1]         # the last row
lp.cols[:3]         # the first three columns
lp.cols[1,'name',5] # column 1, a column named 'name', and column 5
```

One may also query the length of this sequence to get the number of rows or columns, and del to get rid of rows or columns, e.g.:

```
len(lp.cols)        # the number of columns in the problem
del lp.rows['arow'] # deletes a row named 'arow'
del lp.rows[-2:]    # deletes the last two rows
```

add(*n*)

Add n more rows (constraints) or columns (struct variables). Returns the index of the first added entry.

class glpk.**BarCollectionIter**

Bar collection iterator objects

Created for iterating over the rows and columns contained with a bar collection.

class glpk.**Environment**

This represents the PyGLPK environment. Through this, one may control the global behavior of the GLPK. One instance of this exists, named env in the glpk module.

blocks

The number of currently allocated memory blocks.

blocks_peak

The peak value of the blocks attribute.

bytes

The number of currently allocated memory bytes.

bytes_peak

The peak value of the bytes attribute.

mem_limit

The memory limit in megabytes. None if no limit is set.

term_hook

Function to intercept all terminal output. This should be a callable object that accepts a single string argument, or None to indicate that no hook is set (e.g., all output goes to the terminal, default behavior). Note that when the function is called, there is no guarantee that the input string will be a full line, or even non-empty. All exceptions thrown by the function will go ignored and unreported.

term_on

Whether or not terminal output for the underlying GLPK procedures is on or off.

version

Tuple holding the major version and minor version of the GLPKthat this PyGLPK module was built upon. For example, if built against GLPK 4.31, version==(4,31).

class glpk.KKT

Karush-Kuhn-Tucker conditions.

This is returned from a check on quality of solutions. Four types of conditions are stored here:

- KKT.PE conditions are attributes prefixed by 'pe' measuring error in the primal solution.
- KKT.PB conditions are attributes prefixed by 'pb' measuring error in satisfying primal bound constraints, i.e., feasibility.
- KKT.DE and KKT.DB are analogous, but for the dual.

db_ae_ind

Index of the variable with the largest absolute error.

db_ae_max

Largest absolute error.

db_quality

Character representing the quality of primal feasibility. 'H', high, 'M', medium, 'L', low, or '?' wrong or infeasible.

db_re_ind

Index of the variable with the largest relative error.

db_re_max

Largest relative error.

de_ae_max

Largest absolute error.

de_ae_row

Index of the column with the largest absolute error.

de_quality

Character representing the quality of the primal solution. 'H', high, 'M', medium, 'L', low, or '?' wrong or infeasible.

de_re_max

Largest relative error.

de_re_row

Index of the column with the largest relative error.

pb_ae_ind

Index of the variable with the largest absolute error.

pb_ae_max

Largest absolute error.

pb_quality

Character representing the quality of primal feasibility. 'H', high, 'M', medium, 'L', low, or '?' wrong or infeasible.

pb_re_ind

Index of the variable with the largest relative error.

pb_re_max

Largest relative error.

pe_ae_max

Largest absolute error.

pe_ae_row

Index of the row with the largest absolute error.

pe_quality

Character representing the quality of the primal solution. 'H', high, 'M', medium, 'L', low, or '?' wrong or infeasible.

pe_re_max

Largest relative error.

pe_re_row

Index of the row with the largest relative error.

class glpk.**LPX**

class glpk.**LPX**(*gmp=filename*) → linear program with data read from a GNU MathProg file

containing model and data LPX(mps=filename) -> linear program with data read from a datafile in fixed

MPS format

LPX(freemps=filename) -> linear program with data read from a datafile in
free MPS format

LPX(cpxlp=filename) -> linear program with data read from a datafile in
fixed CPLEX LP format

LPX(glp=filename) -> linear program with data read from a datafile in GNU
LP format

LPX(gmp=(model_filename[, data_filename[, output_filename]]) -> linear
program from GNU MathProg input files. The first element is a path to model second, the second to the data section. If the second element is omitted or is None then the model file is presumed to also hold the data. The third element holds the output data file to write display statements to. If omitted or None, the output is instead put through to standard output.

This represents a linear program object. It holds data and offers methods relevant to the whole of the linear program. There are many members in this class, but the most important are: obj -> represents the objective function rows -> a collection over which one can access rows cols -> same, but for columns

adv_basis()

Construct an advanced initial basis, triangular with as few variables as possible fixed.

cols

Column collection. See the help on class BarCollection.

copy()

Copies the content of this problem into a new problem and returns it.

cpx_basis()

Construct an advanced Bixby basis. This basis construction method is described in: Robert E. Bixby. Implementing the Simplex Method: The Initial Basis. ORSA Journal on Computing, Vol. 4, No. 3, 1992, pp. 267-84.

`dual_ratio_test`([*(glpk.Bar, float)*, *int*, *float*]) → int

Perform dual ratio test using an explicitly specified row of the simplex tableau.

The row of the simplex tableau is given as a list of tuples, with each tuple containing a basic variable and a coefficient. The second argument is an integer specifying the direction in which the variable changes when entering the basis: +1 means increasing, -1 means decreasing. The third argument is an absolute tolerance used by the routine to skip small coefficients.

Returns the index of the input row corresponding to the pivot element.

`erase`()

Erase the content of this problem, restoring it to the state it was in when it was first created.

`exact`()

Attempt to solve the problem using an exact simplex method.

This returns None if the problem was successfully solved. Alternately, on failure it will return one of the following strings to indicate failure type:

fault
There are no rows or columns, or the initial basis is invalid, or the initial basis matrix is singular or ill-conditioned.

itlim
Iteration limited exceeded.

tmlim
Time limit exceeded.

`integer`()

MIP solver based on branch-and-bound.

This procedure has a great number of optional keyword arguments to control the functioning of the solver. We list these here, including descriptions of their legal values:

msg_lev
Controls the message level of terminal output.

LPX.MSG_OFF
no output (default)

LPX.MSG_ERR
error and warning messages

LPX.MSG_ON
normal output

LPX.MSG_ALL
full informational output

br_tech
Branching technique option.

LPX.BR_FFV
first fractional variable

LPX.BR_LFV
last fractional variable

LPX.BR_MFV
most fractional variable

LPX.BR_DTH
heuristic by Driebeck and Tomlin (default)

LPX.BR_PCH
hybrid pseudo-cost heuristic

bt_tech
Backtracking technique option.

LPX.BT_DFS
depth first search

LPX.BT_BFS
breadth first search

LPX.BT_BLB
best local bound (default)

LPX.BT_BPH
best projection heuristic

pp_tech
Preprocessing technique option.

LPX.PP_NONE
disable preprocessing

LPX.PP_ROOT
perform preprocessing only on the root level

LPX.PP_ALL
perform preprocessing on all levels (default)

sr_heur
Simple rounding heuristic (default True) (requires glpk >= 4.57.0)

fp_heur
Feasibility pump heurisic (default False)

ps_heur
Proximity search heuristic (default False)

ps_tm_lim
Proximity search time limit in milliseconds (default 60000)

gmi_cuts
Use Gomory's mixed integer cuts (default False)

mir_cuts
Use mixed integer rounding cuts (default False)

cov_cuts
Use mixed cover cuts (default False)

clq_cuts
Use clique cuts (default False)

tol_int
Tolerance used to check if the optimal solution to the current LP relaxation is integer feasible.

tol_obj
Tolerance used to check if the objective value in the optimal solution to the current LP is not better than the best known integer feasible solution.

mip_gap
elative mip gap tolerance (default 0.0)

tm_lim
Search time limit in milliseconds. (default is max int)

out_frq
Terminal output frequency in milliseconds. (default 5000)

out_dly
Terminal output delay in milliseconds. (default 10000)

presolve
MIP presolver (default False)

binarize
Binarization option, used only if presolver is enabled (default False)

callback
A callback object the user may use to monitor and control the solver. During certain portions of the optimization, the solver will call methods of callback object. (default None)

The last parameter, callback, is worth its own discussion. During the branch-and-cut algorithm of the MIP solver, at various points callback hooks are invoked which allow the user code to influence the proceeding of the MIP solver. The user code may influence the solver in the hook by modifying and operating on a Tree instance passed to the hook. These hooks have various codes, which we list here:

select
request for subproblem selection

prepro
request for preprocessing

rowgen
request for row generation

heur
request for heuristic solution

cutgen
request for cut generation

branch
request for branching

bingo
better integer solution found

During the invocation of a hook with a particular code, the callback object will have a method of the same name as the hook code called, with the Tree instance. For instance, for the 'cutgen' hook, it is equivalent to:

```
callback.cutgen(tree)
```

being called with tree as the Tree instance. If the method does not exist, then instead the method 'default' is called with the same signature. If neither the named hook method nor the default method exist, then the hook is ignored.

This method requires a mixed-integer problem where an optimal solution to an LP relaxation (either through simplex() or exact()) has already been found. Alternately, try intopt().

This returns None if the problem was successfully solved. Alternately, on failure it will return one of the following strings to indicate failure type.

fault
There are no rows or columns, or it is not a MIP problem, or integer variables have non-int bounds.

nopfs
No primal feasible solution.

nodfs
Relaxation has no dual feasible solution.

itlim
Iteration limited exceeded.

tmlim
Time limit exceeded.

sing
Error occurred solving an LP relaxation subproblem.

`interior()`

Attempt to solve the problem using an interior-point method.

This returns None if the problem was successfully solved. Alternately, on failure it will return one of the following strings to indicate failure type:

fault
There are no rows or columns.

nofeas
The problem has no feasible (primal/dual) solution.

noconv
Very slow convergence or divergence.

itlim
Iteration limited exceeded.

instab
Numerical instability when solving Newtonian system.

`intopt()`

More advanced MIP branch-and-bound solver than integer(). This variant does not require an existing LP relaxation.

This returns None if the problem was successfully solved. Alternately, on failure it will return one of the following strings to indicate failure type.

fault
There are no rows or columns, or it is not a MIP problem, or integer variables have non-int bounds.

nopfs
No primal feasible solution.

nodfs
Relaxation has no dual feasible solution.

itlim
Iteration limited exceeded.

tmlim
Time limit exceeded.

sing
Error occurred solving an LP relaxation subproblem.

kind

Either the type 'float' if this is a pure linear programming (LP) problem, or the type 'int' if this is a mixed integer programming (MIP) problem.

kkt([*scaled=False*])

Return an object encapsulating the results of a check on the Karush-Kuhn-Tucker optimality conditions for a basic (simplex) solution. If the argument 'scaled' is true, return results of checking the internal scaled instance of the LP instead.

kktint()

Similar to kkt(), except analyzes solution quality of an mixed-integer solution. Note that only the primal components of the KKT object will have meaningful values.

matrix

The constraint matrix as a list of three element (row index, column index, value) tuples across all non-zero elements of the constraint matrix.

name

Problem name, or None if unset.

nbin

The number of binary column variables, i.e., integer with 0 to 1 bounds. Always 0 if this is not a mixed integer problem.

nint

The number of integer column variables. Always 0 if this is not a mixed integer problem.

nnz

Number of non-zero constraint coefficients.

obj

Objective function object.

prime_ratio_test([*(glpk.Bar, float)*, *int*, *float*]) → int

Perform primal ratio test using an explicitly specified column of the simplex tableau.

The column of the simplex tableau is given as a list of tuples, with each tuple containing a basic variable and a coefficient. The second argument is an integer specifying the direction in which the variable changes when entering the basis: +1 means increasing, -1 means decreasing. The third argument is an absolute tolerance used by the routine to skip small coefficients.

Returns the index of the input column corresponding to the pivot element.

ray

A non-basic row or column the simplex solver has identified as causing primal unboundness, or None if no such variable has been identified.

rows

Row collection. See the help on class BarCollection.

scale([*flags=LPX.SF_AUTO*])

Perform automatic scaling of the problem data, in order to improve conditioning. The behavior is controlled by various flags, which can be bitwise ORed to combine effects. Note that this only affects the internal state of the LP representation. These flags are members of the LPX class:

SF_GM
perform geometric mean scaling

SF_EQ
perform equilibration scaling

SF_2N
round scale factors to the nearest power of two

SF_SKIP
skip scaling, if the problem is well scaled

SF_AUTO
choose scaling options automatically

`simplex`([*keyword arguments*])

Attempt to solve the problem using a simplex method.

This procedure has a great number of optional keyword arguments to control the functioning of the solver. We list these here, including descriptions of their legal values.

msg_lev
Controls the message level of terminal output.

LPX.MSG_OFF
no output (default)

LPX.MSG_ERR
error and warning messages

LPX.MSG_ON
normal output

LPX.MSG_ALL
full informational output

meth
Simplex method option

LPX.PRIMAL
use two phase primal simplex (default)

LPX.DUAL
use two phase dual simplex

LPX.DUALP
use two phase dual simplex, primal if that fails

pricing
Pricing technique

LPX.PT_STD
standard textbook technique

LPX.PT_PSE
projected steepest edge (default)

r_test
Ratio test technique

LPX.RT_STD
standard textbook technique

LPX.RT_HAR
Harris' two-pass ratio test (default)

tol_bnd
Tolerance used to check if the basic solution is primal feasible. (default 1e-7)

tol_dj
Tolerance used to check if the basic solution is dual feasible. (default 1e-7)

tol_piv
Tolerance used to choose pivotal elements of the simplex table. (default 1e-10)

obj_ll
Lower limit of the objective function. The solver terminates upon reaching this level. This is used only in dual simplex optimization. (default is min float)

obj_ul
Upper limit of the objective function. The solver terminates upon reaching this level. This is used only in dual simplex optimization. (default is max float)

it_lim
Simplex iteration limit. (default is max int)

tm_lim
Search time limit in milliseconds. (default is max int)

out_frq
Terminal output frequency in iterations. (default 200)

out_dly
Terminal output delay in milliseconds. (default 0)

presolve
Use the LP presolver. (default False)

This returns None if the problem was successfully solved. Alternately, on failure it will return one of the following strings to indicate failure type.

fault
There are no rows or columns, or the initial basis is invalid, or the initial basis matrix is singular or ill-conditioned.

objll
The objective reached its lower limit.

objul
The objective reached its upper limit.

itlim
Iteration limited exceeded.

tmlim
Time limit exceeded.

sing
The basis matrix became singular or ill-conditioned.

nopfs
No primal feasible solution. (Presolver only.)

nodfs
No dual feasible solution. (Presolver only.)

`status`

The status of solution of the last solver.

This takes the form of a string with these possible values:

opt
The solution is optimal.

undef
The solution is undefined.

feas
The solution is feasible, but not necessarily optimal.

infeas
The solution is infeasible.

nofeas
The problem has no feasible solution.

unbnd
The problem has an unbounded solution.

`status_dual`

The status of the dual solution of the simplex solver.

Possible values are 'undef', 'feas', 'infeas', 'nofeas' in similar meaning to the .status attribute.

`status_i`

The status of the interior point solver's solution.

`status_m`

The status of the MIP solver's solution.

`status_primal`

The status of the primal solution of the simplex solver.

Possible values are 'undef', 'feas', 'infeas', 'nofeas' in similar meaning to the .status attribute.

`status_s`

The status of the simplex solver's solution.

`std_basis()`

Construct the standard trivial inital basis for this LP.

`transform_col`([*(glpk.Bar*, *float)*, ...]) → [*glpk.Bar*, float, ...]

Transforms the explicitly specified column

The column to be transformed is given as a list of tuples, with each tuple containing a variable (i.e., an instance of glpk.Bar) and a coefficient. The input variables should be auxiliary variables (i.e., elements of LPX.rows).

The column is returned as a list of tuples containing a reference to a basic variable and the corresponding coefficient from the simplex tableau.

`transform_row`([*(glpk.Bar*, *float)*, ...]) → [*glpk.Bar*, float, ...]

Transforms the explicitly specified row

The row to be transformed is given as a list of tuples, with each tuple containing a variable (i.e., an instance of glpk.Bar) and a coefficient. The input variables should be structural variables (i.e., elements of LPX.cols).

The row is returned as a list of tuples containing a reference to a non-basic variable and the corresponding coefficient from the simplex tableau.

`unscale()`

This unscales the problem data, essentially setting all scale factors to 1.

`warm_up()` → string

Warms up the LP basis.

Returns None if successful, otherwise one of the following error strings:

badb
the basis matrix is invalid

sing
the basis matrix is singular

cond
the basis matrix is ill-conditioned

`write`(*format=filename*)

Output data about the linear program into a file with a given format. What data is written, and how it is written, depends on which of the format keywords are used. Note that one may specify multiple format and filename pairs to write multiple types and formats of data in one call to this function.

mps
For problem data in the fixed MPS format.

bas
The current LP basis in fixed MPS format.

freemps
Problem data in the free MPS format.

cpxlp
Problem data in the CPLEX LP format.

glp
Problem data in the GNU LP format.

sol
Basic solution in printable format.

sens_bnds
Bounds sensitivity information.

ips
Interior-point solution in printable format.

mip
MIP solution in printable format.

`class glpk.Objective`

Objective function objects for linear programs.

An instance is used either to access objective function values for solutions, or to access or set objective function coefficients. The same indices valid for a BarCollection object over the columns (that is, column numeric indices, column names, slices, multiple values) are also valid for indexing into this object. The special index None is used to specify the shift term. If we have an LPX instance lp, we may have:

```
lp.obj[0]    # the first objective coefficient
lp.obj[None] # the shift term
lp.obj[-3:]  # the last three objective coefficients
```

(continues on next page)

(continued from previous page)

```
lp.obj[1, 4] # the objective coefficients 1, 4
```

When assigning objective coefficients, for single indices one may assign a single number. For multiple indices, one may assign a single number to make all indicated coefficients identical, or specify an iterable of equal length to set them all individiaully. For example:

```
lp.obj[0] = 2.5          # set the first objective coef to 2.5
lp.obj[-3:] = 1.0        # the last three obj coefs get 1.0
lp.obj[1, 4] = -2.0, 2.0 # obj coefs 1, 4 get -2.0, 2.0
```

maximize

True or False depending on whether we are trying to maximize or minimize this objective function, respectively.

name

Objective name, or None if unset.

shift

The constant shift term of the objective function.

value

The current value of the objective function.

value_i

The current value of the interior point objective function.

value_m

The current value of the MIP objective function.

value_s

The current value of the simplex objective function.

class glpk.**ObjectiveIter**

Objective function iterator objects, used to cycle over the coefficients of the objective function.

class glpk.**Tree**

Tree instances are passed to MIP solver callback function.

They are used to indicate the state of the solver at some intermediate point in a call to LPX.integer(). There are nodes within the tree, instances of TreeNode, corresponding to subproblems within the search tree. The currently active subproblem is stored in the curr_node member of an instance.

best_node

The node of the current active subproblem with best local bound. If the tree is empty, this is None.

branch_upon(*col_index*, *select='N'*)

Given the index of a column in the LP, this will add two new subproblems, down and up branches (in that order) to the active list, where the down and up branches are the problems with the column's variable set to the floor and ceil of the value, respectively. The select parameter controls which of the two branches is selected to next continue the search with 'D', 'U', and 'N' corresponding to choosing the down, up, or letting GLPK select a branch, respectively.

can_branch(*col_index*)

Given the index of a column in the LP, this will return True if one can branch upon this column's varible, that is, continue the search with this column's variable set as an integer. Note that this function should be called only when the reason member of the tree is 'branch'.

curr_node

The node of the current active subproblem. If there is no current active subproblem in the tree, this will return None.

first_node

The node of the first active subproblem. If there is no current active subproblem in the tree, this is None.

gap

The relative MIP gap (duality gap), that is, the gap between the best MIP solution (best_mip) and best relaxed solution (best_bnd) given by this formula:

$$\text{gap} = \frac{|\text{best\_mip} - \text{best\_bnd}|}{|\text{best\_mip}| + \text{epsilon}}$$

heuristic(*values*)

Provide an integer feasible solution of the primal problem, where values is an iterable object yielding at least as many float values as there are columns in the problem. If the provided solution is better than the existing one, the solution is accepted and the problem updated. This function returns True or False depending on whether the solution was accepted or not. Note that this function should be called only when the reason member of the tree is 'heur'.

last_node

The node of the last active subproblem. If there is no current active subproblem in the tree, this is None.

lp

Problem object used by the MIP solver.

num_active

The number of active nodes.

num_all

The number of all nodes, both active and inactive.

num_total

The total number of nodes, including those already removed.

reason

A string with the reason the callback function has been called.

select(*node*)

Selects a tree node to continue search from. Note that this function should be called only when the reason member of the tree is 'select'.

terminate()

Prematurely terminate the MIP solver's search.

class glpk.**TreeIter**

Tree iterator objects.

Created for iterating over the active subproblems of the search tree.

class glpk.**TreeNode**

Represent specific subproblem instances in the search Tree object used by the MIP solver.

active

Whether this node represents an active subproblem.

bound

The local bound for this node's subproblem.

level

The level of the node in the tree, with 0 if this is the root.

next

The next active subproblem node, None if there is no next active subproblem, or if this is not an active subproblem.

prev

The previous active subproblem node, None if there is no previous active subproblem, or if this is not an active subproblem.

subproblem

The reference number of the subproblem corresponding to this node.

up

The parent subproblem node, None if this is the root.

CHAPTER

TEN

GLPK VERSION SPECIFIC NOTES

When building PyGLPK, certain functions will be disabled depending on what version of GLPK the build process believes it is using. Below is a list of GLPK versions, and descriptions of what functionality will not work in earlier versions. (Or, in some cases, later versions.)

The examples may use a hypothetical `glpk.LPX` instance named `lp` for illustrative purposes.

GLPK 4.39

- The interior point solver often does not detect infeasibility on infeasible problems.

GLPK 4.38

- The interior point solver often crashes on infeasible problems.

GLPK 4.37 and higher

- On OS X, the build of the shared library is broken for all versions after and including GLPK 4.37.

GLPK 4.33

- This release had a bug in the integer-solver routines that make parameters cause GLPK to crash. This was fixed in subsequent versions.

GLPK 4.31

- Addition of a `LPX.DUAL` constant for the `meth` keyword parameter of the `LPX.simplex` solver method.
- Addition of the `LPX.SF_GM`, `LPX.SF_EQ`, `LPX.SF_2N`, `LPX.SF_SKIP`, and `LPX.SF_AUTO` constants, for the `LPX.scale` method's argument.

GLPK 4.23

- Addition of the `gmi_cuts` keyword parameter for the `LPX.integer` solver.

GLPK 4.23

- Addition of the `mir_cuts` keyword parameter for the `LPX.integer` solver.

GLPK 4.21

- The ability of the `LPX.integer` callback to have `select`, `prepro`, and `branch` methods called by the callback procedure. For the same reasons, the `Tree` methods `select`, `can_branch`, and `branch_upon` do not appear until this version.
- Addition of the `pp_tech` keyword parameter for the `LPX.integer` solver, and associated constants.

GLPK 4.20

- The keyword parameters of the `LPX.integer` MIP solver method including callbacks, and associated constants for the many keyword parameters, as well as all types associated therein, most notably `Tree` and `TreeNode`.

GLPK 4.19

- The `Environment.mem_limit` attribute for setting the maximum number of megabytes the GLPK will allocate.

CHAPTER

ELEVEN

PYGLPK RELEASE NOTES

PyGLPK 0.3
September 22 2008

- The PyGLPK has been updated to build against GLPK 4.31, and has been updated to take advantage of many of the features of the new version. Nearly all feature changes in this version are to reflect changes in the PyGLPK. This version of PyGLPK will not build against anything prior to GLPK 4.18.
- It is no longer necessary to set the `kind` of a linear program explicitly to `int` to make it a MIP.
- The `Environment` class has been added, with a single instance `env` at the root level of the `glpk` module. With this class one may monitor and control memory usage and terminal output.
- The `simplex` and `integer` solvers now accept many keyword parameters to control their functionality.
- Support for callbacks to control the mixed integer programming solution search have been addeed. See the `Tree` and `TreeNode` types.
- The `Params` class has been removed. Much of its functionality has been obviated as parameters are passed directly to the solvers, and the functionality of the `Environment` class. (If you wish to build a version of PyGLPK with the `Params` type as it was in previous versions, you may edit `useparams = False` to be `useparams = True`.)
- It is now possible to manually change the scaling of rows and columns.

PyGLPK 0.2
July 9 2007

- Added weak reference support to all the PyGLPK objects.
- Testing code is now more exhaustive, and with more testing of smaller components rather than exclusively testing PyGLPK's ability to solve full problems.
- Fixed freeing some Python objects too early: When assigning values to the constraint matrix, matrix value reference counts were not incremented correctly when using implicit indexing (i.e., when the index of an entry was inferred from the previous entry) in matrix assignments to both the entire matrix and individual rows/columns.
- The integer parameter `outfrq` was erroneously identified as being a float parameter in code.

PyGLPK 0.1
March 28 2007

- First release.

CHAPTER

TWELVE

LICENSE

PyGLPK is licensed under the GNU General Public License version 3. It is copyright Thomas Finley at `tfinley@gmail.com`. See the `COPYING` file in the PyGLPK distribution for the GPL v3 license.

I am placing this documentation under the same license, that is, GPL.

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX