Graph-Based Optimization Modeling Language

Bardhyl Miftari, Mathias Berger, Hatim Djelassi, Damien Ernst

Mar 28, 2024

CONTENTS:

1 About GBOML **1**

2 Installation **3**
- 2.1 Installation via pip and PyPI . . . 3
- 2.2 Manual Installation . . . 3
- 2.3 Installing Solvers . . . 4
  - 2.3.1 Gurobi . . . 4
  - 2.3.2 CPLEX . . . 4
  - 2.3.3 Xpress . . . 4
  - 2.3.4 Cbc/Clp . . . 4
  - 2.3.5 DSP . . . 5
  - 2.3.6 HiGHS . . . 5
- 2.4 Testing . . . 5

3 Abstract GBOML problem **7**

4 Grammar Basics **9**
- 4.1 Identifiers . . . 9
- 4.2 Numbers . . . 9
- 4.3 Operators . . . 9
- 4.4 Expressions . . . 10
- 4.5 Logical Conditions . . . 10
- 4.6 Comments . . . 11

5 Block Definitions **13**
- 5.1 Time Horizon . . . 14
- 5.2 Global Parameters . . . 14
- 5.3 Nodes . . . 15
  - 5.3.1 Parameters . . . 15
  - 5.3.2 Variables . . . 16
  - 5.3.3 Constraints . . . 16
  - 5.3.4 Objectives . . . 18
- 5.4 Hyperedges . . . 19
  - 5.4.1 Parameters . . . 20
  - 5.4.2 Constraints . . . 20

6 Advanced Features **21**
- 6.1 Hierarchical Models . . . 21
- 6.2 Importing Nodes and Hyperedges . . . 24
  - 6.2.1 Importing Nodes . . . 24
  - 6.2.2 Importing Hyperedges . . . 25

7 Useful Idioms **27**
7.1 Repeating Data . . . 27
7.2 Round Down Integer Division . . . 27

8 How to Use **29**
8.1 Command Line Interface . . . 29
8.2 Python Interface . . . 31
8.3 Solver APIs . . . 38

9 Examples **41**
9.1 Microgrid Example . . . 41
9.1.1 Problem Description . . . 41
9.1.2 GBOML Implementation . . . 41
9.1.3 Running the Example . . . 46
9.2 Remote Hub Example . . . 46
9.2.1 Problem Description . . . 46
9.2.2 Running the Example . . . 48
9.3 Python API Example . . . 49
9.4 Models and papers that use GBOML . . . 50

10 Citing GBOML **51**

11 Indices and tables **53**

Index **55**

CHAPTER

ONE

ABOUT GBOML

The Graph-Based Optimization Modeling Language (GBOML) is a modeling language for mathematical programming designed and implemented at the University of Liège, Belgium. GBOML enables the easy implementation of a broad class of structured mixed-integer linear programs typically found in applications ranging from energy system planning to supply chain management. More precisely, the language is particularly well-suited for representing problems involving the optimization of discrete-time dynamical systems over a finite time horizon and possessing a block structure that can be encoded by a hierarchical hypergraph. The language combines elements of both algebraic and object-oriented modeling languages in order to facilitate problem encoding and model re-use, speed up model generation, expose problem structure to specialised solvers and simplify post-processing. The GBOML parser, which is implemented in Python, turns GBOML input files into hierarchical graph data structures representing optimization models. The associated tool provides both a command-line interface and a Python API to construct models, and directly interfaces with a variety of open source and commercial solvers, including structure-exploiting ones.

CHAPTER

TWO

INSTALLATION

Two installation options are currently available. First, GBOML can be installed via the pip package manager and the Python Package Index (PyPI). Second, GBOML can be installed manually by cloning the git repository and installing the requirements. In addition, solvers must be installed separately, as described below.

2.1 Installation via pip and PyPI

GBOML can be installed via pip and the Python Package Index by typing the following commands in a terminal window:

```
pip install gboml
```

All dependencies (numpy, scipy and ply) will be automatically installed and the package should be ready for use.

2.2 Manual Installation

The git repository can be found here. The repository can be cloned by typing the following commands in a terminal window:

```
git clone https://gitlab.uliege.be/smart_grids/public/gboml
```

Then, a local installation can be performed by typing the following commands in a terminal window:

```
pip install .
```

If you only want GBOML as an uninstalled package, installing the requirements can be performed by typing the following commands:

```
pip install -r requirements.txt
```

2.3 Installing Solvers

GBOML currently interfaces with Gurobi, CPLEX, Xpress, Cbc/Clp, HiGHS and DSP. Only one of these is required to solve a GBOML model. Gurobi, CPLEX and Xpress are commercial solvers, while Cbc/Clp is an open-source solver. DSP is an experimental open-source project relying on Gurobi, CPLEX and SCIP to implement generic structure-exploiting algorithms (e.g., Dantzig-Wolfe, dual and Benders decompositions).

2.3.1 Gurobi

To use Gurobi, you must first install it. Instructions can be found here. Once the solver is installed, the Python API can be downloaded by typing the following commands in a terminal window:

```
python -m pip install -i https://pypi.gurobi.com gurobipy
```

Other installation options can be found in this post. Note that a license is also required to use Gurobi. Free licenses can be requested for academics, as discussed in the following post.

2.3.2 CPLEX

To use CPLEX, you must first install it. Instructions can be found here. Once the solver is installed, the Python API can be downloaded by typing the following commands in a terminal window:

```
pip install cplex
```

Note that a license is also required to use CPLEX. Licenses can be obtained for free for academics, as discussed in the following post.

2.3.3 Xpress

To use Xpress, you must first install it. Instructions can be found here. Once the solver is installed, the Python API can be downloaded by typing the following commands in a terminal window:

```
pip install xpress
```

Additional information can be found here. Note that a license is also required to use Xpress.

2.3.4 Cbc/Clp

To use Cbc or Clp, you must first install them. Instructions can be found here. The CyLP package is used to interface with the solver. This package can be installed by typing the following commands in a terminal window:

```
pip install cylp
```

2.3.5 DSP

To use DSP, you must first install it. At present, DSP developers recommend installing it on a Mac or Linux machine. Installing DSP with the Windows Subsystem Linux UBUNTU 18.04 distribution was tested and found to work too.

The recommended installation steps work as follows. First, the DSP repository must be cloned into a directory of choice. This can be achieved by creating a directory named, e.g., *your_DSP_directory*:

```
mkdir your_DSP_directory
cd your_DSP_directory
```

and cloning the DSP repository recursively:

```
git clone --recursive https://github.com/Argonne-National-Laboratory/DSP.git
```

Then, the absolute paths of the directories storing the libraries and header files of the solvers used to build DSP (e.g., Gurobi, CPLEX or SCIP) must be specified in a file named `UserConfig.cmake`, which must be placed in the cloned DSP directory. Note that these paths must be consistent with that of the directory in which the solver was installed in the first place. For example, on Mac, CPLEX library files may be stored in */Applications/CPLEX_Studio1210/cplex/lib/x86-64_osx/static_pic*, while header files may be stored in */Applications/CPLEX_Studio1210/cplex/include/ilcplex*. Gurobi libraries and header files may be stored in */Library/gurobi903/mac64/lib* and */Library/gurobi903/mac64/include*, respectively. In addition, it may sometimes be necessary to also add some of these libraries and DSP dependencies on the library path (e.g., by setting the value of the DYLD_LIBRARY_PATH environment variable in your bash profile on Mac) prior to proceeding to the build stage.

The next installation steps make use of cmake and make to build the DSP executable and library. Once cmake is installed, the following commands can be typed in a terminal window, starting in the cloned DSP directory:

```
mkdir build
cd build
cmake ..
make
```

If the make worked properly, an executable called runDsp and a shared library named libDsp should be created in the *src* subfolder of the build directory. Additional information can be found here.

2.3.6 HiGHS

To install HiGHS please download the solver from https://highs.dev/. The Python API is embedded in GBOML. However, you need the HiGHS shared object on your library path.

2.4 Testing

To manually test your installation, you can type the following commands in a terminal window:

```
python test.py
```

Note that running *test.py* tests all solver APIs (except that of DSP, which is still experimental). Therefore, installing only one solver will not result in all tests being passed.

CHAPTER

THREE

ABSTRACT GBOML PROBLEM

The modeling language is particularly well-suited for representing problems involving the optimization of discrete-time dynamical systems over a finite time horizon and exhibiting a natural block structure that may be encoded by a hierarchical hypergraph. A hypergraph abstraction is therefore employed to represent them. Nodes can therefore be viewed as hierarchical hypergraphs themselves representing optimization subproblems, while hyperedges express the relationships between nodes. A global discretized time horizon and associated set of time periods common to all nodes are also defined. Each node is equipped with a set of so-called *internal* and *external* (or *coupling*) variables. A set of constraints may be also defined for each node, along with a local objective function representing its contribution to a system-wide objective. Finally, for each hyperedge, constraints involving the coupling variables of the nodes to which it is incident are defined in order to express the relationships between nodes. In the following paragraphs, we formally define variables, constraints, objectives and formulate the abstract model that encapsulates the class of problems considered. For the sake of clarity, we do so for a model that can be represented by a hypergraph with a single *layer* (i.e., there is one level in the hierarchy).

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a (possibly directed) hypergraph encoding the block structure of the problem at hand, with node set $\mathcal{N}$ and hyperedge set $\mathcal{E} \subseteq 2^{\mathcal{N}}$ (i.e., each hyperedge corresponds to a subset of nodes), let T be the time horizon considered and let $\mathcal{T} = \{0, 1, \ldots, T-1\}$ be the associated set of time periods. Let $X^n \in \mathcal{X}^n$ and $Z^n \in \mathcal{Z}^n$ denote the collection of internal and coupling variables defined at node $n \in \mathcal{N}$. Note that variables may take values in discrete or continuous sets. In addition, for any hyperedge $e \in \mathcal{E}$, let $Z^e = \{Z^n | n \in e\}$ denote the collection of coupling variables associated with each node to which this hyperedge is incident.

Let F^n denote the function defining the local objective at node $n \in \mathcal{N}$. In this paper, we consider scalar objectives of the form

$$F^n(X^n, Z^n) = f_0^n(X^n, Z^n) + \sum_{t \in \mathcal{T}} f^n(X^n, Z^n, t),$$

where f_0^n and f^n are (scalar) affine functions of X^n and Z^n.

Both equality and inequality constraints may be defined at each node $n \in \mathcal{N}$. More precisely, an arbitrary number of constraints that can each be expanded over a subset of time periods may be defined. Hence, we consider equality constraints of the form

$$h_k^n(X^n, Z^n, t) = 0, \ \forall t \in \mathcal{T}_k^n$$

with (scalar) affine functions h_k^n and index sets $\mathcal{T}_k^n \subseteq \mathcal{T}$, $k = 1, \ldots, K^n$, as well as inequality constraints

$$g_k^n(X^n, Z^n, t) \leq 0, \ \forall t \in \bar{\mathcal{T}}_k^n,$$

with (scalar) affine functions g_k^n and index sets $\bar{\mathcal{T}}_k^n \subseteq \mathcal{T}$, $k = 1, \ldots, \bar{K}^n$.

Likewise, both equality and inequality constraints may be defined over any hyperedge $e \in \mathcal{E}$. These constraints, however, can only involve the coupling variables of the nodes to which hyperedge $e \in \mathcal{E}$ is incident (i.e., nodes such that $n \in e$). More precisely, let H^e and G^e be affine functions of Z^e used to define the equality and inequality constraints associated with a given hyperedge $e \in \mathcal{E}$.

Using this notation, the class of problems that can be represented in this framework reads

$$
\begin{aligned}
\min \quad & \textstyle\sum_{n\in\mathcal{N}} F^n(X^n, Z^n) \\
\text{s.t.} \quad & h_k^n(X^n, Z^n, t) = 0,\ \forall t \in \mathcal{T}_k^n,\ k = 1, \dots K^n,\ \forall n \in \mathcal{N} \\
& g_k^n(X^n, Z^n, t) \leq 0,\ \forall t \in \bar{\mathcal{T}}_k^n,\ k = 1, \dots \bar{K}^n,\ \forall n \in \mathcal{N} \\
& H^e(Z^e) = 0,\ \forall e \in \mathcal{E} \\
& G^e(Z^e) \leq 0,\ \forall e \in \mathcal{E} \\
& X^n \in \mathcal{X}^n, Z^n \in \mathcal{Z}^n,\ \forall n \in \mathcal{N}.
\end{aligned}
$$

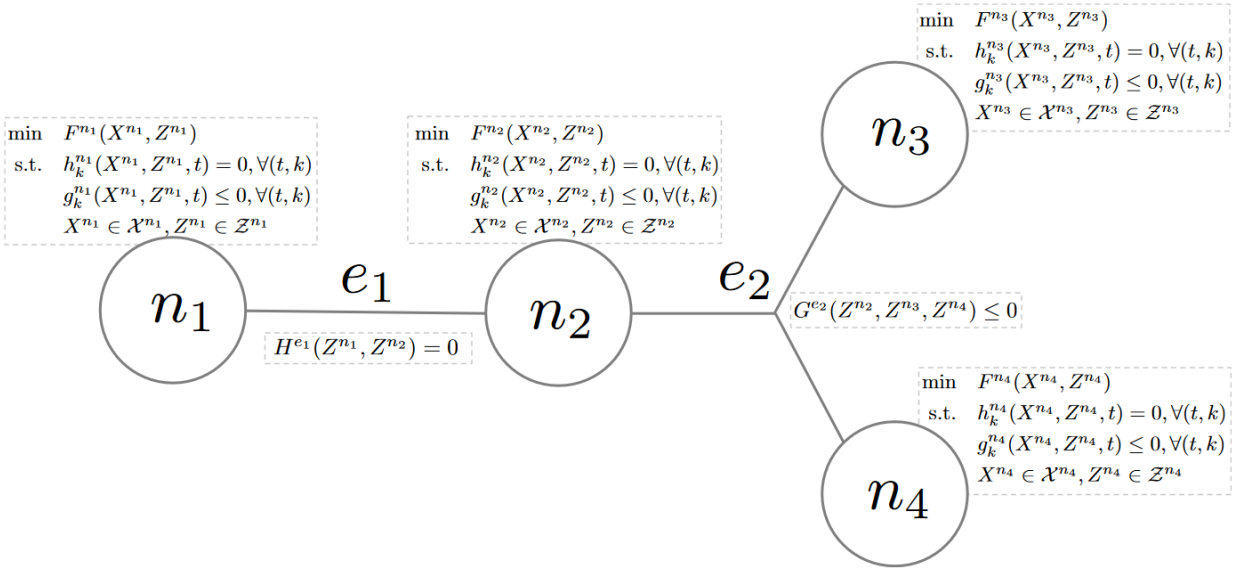

Fig. 3.1: Schematic representation of the abstract GBOML problem.

Fig. 3.1 schematically illustrates the class of problems that can be modelled in this framework.

CHAPTER

FOUR

GRAMMAR BASICS

The basic elements of the grammar include identifiers, numbers, operators, expressions and logical conditions. These different elements can be used in various code blocks in order to construct a complete model. The following elements are introduced in this section:

4.1 Identifiers

Identifiers are used to name different kinds of language objects such as nodes, hyperedges or variables. Identifiers may contain letters, numbers, underscores, and dollar signs but must begin with a letter or an underscore. Accordingly, the following identifiers are all valid: `mynode1`, `_SolarPlant_2` and `HydroStorage_a`.

Besides these lexical requirements, identifiers must also be unique in their respective scope. Hence, no two nodes may have the same identifier since this would prohibit the unambiguous identification of a particular node. Similarly, variables and parameters may not have the same identifier as a node or other variables and parameters belonging to the same node. However, the same identifier may be re-used to define variables or parameters that belong to different nodes.

4.2 Numbers

GBOML recognizes floating-point numbers and integers. Depending on the context, an integer may be called for, but in contexts where a floating-point number is required, an integer will also be accepted and converted to a floating-point number automatically. It should be noted that scientific notation is supported and automatically converted to floating-point numbers. Accordingly, the following are considered floating-point numbers,

`1e-5`, `-2.5e-10`, `2e10`.

4.3 Operators

GBOML provides the following operators for elementary floating-point arithmetic, which are listed in order of decreasing precedence:

1. Exponentiation: `**`
2. Multiplication: `*` and Division: `/`
3. Addition: `+` and Subtraction: `-`

Operator precedence can be overridden by using parentheses `(` and `)`. Besides these elementary arithmetic operators, GBOML provides the modulo operator and the sum operator as native functions:

- Modulo: `mod(<dividend>, <divisor>)`
- Sum: `sum(<expression> for <id> in [<start>: <end>])`

In addition, a set of comparison operators are available in the language:

- Equal to: `==`
- Not equal to: `!=`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
- Less than: `<`
- Greater than: `>`

Finally, a set of logical operators are available, listed below in order of decreasing precedence:

1. Negation: `not`
2. Conjunction: `and`
3. Disjunction: `or`

Note that the precedence of logical operators can also be overriden by using parentheses.

4.4 Expressions

Algebraic expressions are used to construct the different components of a model, such as its constraints and objective. Expressions typically involve numbers and identifiers referring to parameters and variables, along with some of the operators introduced previously. While numbers are always scalar, variables and parameters may be either scalars or vectors (of any length). Their entries are accessed via an index written in brackets `[` and `]`. For the sake of illustration, let `v` be an identifier referring to a vector quantity of length `L`. Then, the entries of `v` are accessed via

```
v[0],    v[1],    v[2],    ...,    v[L-1].
```

Note that the first index is `0` and not `1`.

Let `x` and `v` be the identifiers of a scalar quantity and a vector quantity, respectively. Then, the following expressions are valid in GBOML:

```
x**2 - 6.5*x - 9,    sum((i - 2.5)/3 for i in [0:10]),    v[0] + v[mod(3,2)]
```

Note that the above expressions contain whitespace characters, which are not required. Indeed, all kinds of whitespace characters (space, tabulation, line feed, form feed, and carriage return) are ignored by the GBOML compiler.

4.5 Logical Conditions

Besides their immediate use in model equations, expressions can be further used to construct logical conditions. More specifically, logical conditions are constructed from expressions, comparison operators, and logical operators.

Let `t` denote an integer. Then, the following logical conditions are valid in GBOML:

```
t > 0 and t <= 10,    t < 2 or t > 4,    not mod(t,5) == 0.
```

Note that conditions can be used to selectively enforce constraints over a subset of indices, which is discussed when introducing the `#CONSTRAINTS` block.

4.6 Comments

Comments are initiated by a double forward slash // and terminated by a line feed. The content that follows is ignored by the compiler.

CHAPTER

FIVE

BLOCK DEFINITIONS

In order to implement an instance of the abstract GBOML problem, the model must be encoded in an input file written in the GBOML grammar. This input file is structured into blocks, which are introduced by one of the following keywords, namely `#TIMEHORIZON`, `#GLOBAL`, `#NODE`, and `#HYPEREDGE`. The time horizon information is given in the `#TIMEHORIZON` block, which must be the first one defined in the input file. Global parameters must be defined in the `#GLOBAL` block, which must come in second position. Then, each node can be defined in a `#NODE` block, while each hyperedge can be defined in a `#HYPEREDGE` block. Note that the order in which `#NODE` and `#HYPEREDGE` blocks appear in the input file does not matter (i.e., hyperedge definitions may precede node definitions and vice-versa). Thus, an input file is typically structured as follows:

```
#TIMEHORIZON
// time horizon definition

#GLOBAL
// global parameters

#NODE <identifier>
// first node definition

#NODE <identifier>
// second node definition

#HYPEREDGE <identifier>
// first hyperedge definition

// possibly further node blocks

#HYPEREDGE <identifier>
// second hyperedge definition

// possibly further hyperedge blocks
```

These different blocks are further discussed in this section:

5.1 Time Horizon

The time horizon `T` defines the length of the optimization horizon (i.e., the number of time periods considered). This definition is contained in the `#TIMEHORIZON` block, which is the first one that should appear in a GBOML file. This block has the following structure:

```
#TIMEHORIZON
T = <expression>;
```

Therein, `<expression>` is an algebraic expression that should evaluate to a positive integer. If `<expression>` evaluates to a positive but non-integral value, it will be rounded to the closest integer automatically and a warning will be raised. Expressions that cannot be evaluated and expressions that evaluate to a negative value are not permitted. In addition, since the `#TIMEHORIZON` block is the first block of any input file and no parameters can been defined before it, `<expression>` may not depend on any parameters. An example of a valid `#TIMEHORIZON` block is given below:

```
#TIMEHORIZON
T = 24*5.5+6;
```

The definition of a time horizon has two effects on the remainder of the model. First, the time horizon can be addressed as a parameter anywhere in the model by referring to its identifier `T`. Accordingly, the identifier `T` is reserved and cannot be re-used for the definition of nodes, variables, or parameters in the remainder of the model. Second, constraints and objectives can be automatically expanded over the time full horizon by using the identifier `t` as an index with vector variables. In other words, the constraints or objectives that use `t` as an index are automatically expanded for each $\texttt{t} \in \{0, 1, ..., T-1\}$. Accordingly, `t` is a reserved identifier that cannot be used for any other purpose.

5.2 Global Parameters

The non-mandatory `#GLOBAL` block contains the definitions of parameters that can be accessed anywhere in the model. This block is structured as follows:

```
#GLOBAL
// global parameter definitions
```

A parameter definition maps an identifier to a fixed value, which may be either a scalar or a vector. The identifier must be unique within a given `#GLOBAL` block and a value can be assigned to a parameter through one of the following three syntax rules:

```
< identifier > = < expression >;
< identifier > = { < term > , < term > ,...};
< identifier > = import " < filename >";
```

First, a scalar parameter is defined according to the first rule. Therein, `<expression>` may be a scalar expression that evaluates to any floating-point number. In particular, it may contain global parameters that have been defined previously. If the parameter definition is valid with respect to these rules, a parameter named `<identifier>` will be created and assigned the value of `<expression>`. Second, a vector parameter can be defined directly by providing a comma-separated list of values according to the second rule. Therein, each `<term>` may be a floating-point number, a previously-defined scalar parameter, or an entry of a previously-defined vector parameter. The resulting vector parameter can be indexed in order to retrieve its constituent entries. Third, a vector parameter can be defined by providing an input file according to the third rule. Therein, `<filename>` refers to an input file in one of several delimiter-separated formats. The supported delimiter characters are comma, semicolon, space, and line feed. In contrast to the direct way of defining a vector parameter, the input file may only contain floating-point values and may not refer to other parameters.

Given these syntax rules, the following is an example of a valid `#GLOBAL` block:

```
#GLOBAL
pi = 3.1416;
two_pi = 2* pi ;
data = import " data . csv ";
len_data = 23;
angles = {0 , data [2] , two_pi };
sum_data = sum ( data [ i ] for i in [0: len_data -1]) ;
```

Notably, the example makes use of the fact that previously-defined parameters can be employed to define new parameters. Furthermore, the parameters defined in this block can be accessed in any other block by referring to their identifier with the prefix `global`. In other words, all global parameters are referred to as:

```
global.<parameter identifier>
```

in the blocks that follow their definition.

5.3 Nodes

In the hypergraph abstraction of optimization problems underpinning the GBOML language, nodes represent optimization subproblems. Hence, each node has its own set of parameters. It is also equipped with a set of variables, which are split into *internal* and *external* (or *coupling*) variables. In addition, a set of constraints can be defined for each node, along with a local objective function representing its contribution to a system-wide objective.

A unique identifier must be assigned to each `#NODE` block, and such a block is further divided into code blocks where parameters, variables, constraints, and objectives can be defined. Each of these blocks is introduced by one of the following keywords, namely `#PARAMETERS`, `#VARIABLES`, `#CONSTRAINTS`, and `#OBJECTIVES`. A typical `#NODE` block is therefore structured as follows:

```
#NODE <node identifier>
#PARAMETERS
// parameter definitions
#VARIABLES
// variable definitions
#CONSTRAINTS
// constraint definitions
#OBJECTIVES
// objective definitions
```

These different code blocks are discussed in further detail below.

5.3.1 Parameters

The parameters defined within a given `#NODE` block respect the same rules as those defined in the `#GLOBAL` block. However, node parameters are local to the present node and parameters defined in different nodes cannot be accessed in this scope.

For the sake of illustration, the following `#PARAMETERS` block is valid in GBOML:

```
#PARAMETERS
gravity       = 9.81;
speed         = import "speed.txt";
```

5.3.2 Variables

Variables are declared with one of the two keywords `internal` and `external`. While `internal` variables are meant to model the internal state of a node, `external` variables are meant to model the interaction between different nodes. That is, the coupling between nodes is modeled by imposing constraints on their `external` variables (which is further discussed when introducing `#HYPEREDGE` blocks). In addition, variables can represent either a scalar or a vector. The syntax for declaring variables in GBOML is as follows:

```
internal : <identifier>;
external : <identifier>;
internal : <identifier> [ <expression> ];
external : <identifier> [ <expression> ];
```

Variables defined only by an identifier are scalar variables, with the identifier giving its name to the variable (Rules 1 and 2). An expression can be added after the identifier to declare a vector variable and specify its length (Rules 3 and 4).

Furthermore, variables can be of different types, which can be specified by using one additional keyword when declaring a variable, namely `continuous`, `integer` or `binary`. Note that if no keyword is specified, variables are assumed to be continuous by default.

Given these syntax rules, the following `#VARIABLES` block is valid in GBOML:

```
#VARIABLES
internal integer : x; // internal integer scalar variable
internal binary  : y[T]; // internal binary variable vector of size T
external : inflow[1000]; // external continuous variable vector of size 1000
external : outflow[1000]; // external continuous variable vector of size 1000
```

5.3.3 Constraints

The syntax rules for the definition of basic equality and inequality constraints are as follows:

```
<expression> == <expression>;
<expression> <= <expression>;
<expression> >= <expression>;
```

Therein, both the left-hand side and the right-hand side of the constraints are general expressions while the type of the constraint is indicated by the comparison operator used. Furthermore, in line with the fact that parameter and variable definitions are local to a given node, constraints defined in a `#NODE` block must not reference quantities that are defined in other nodes.

An identifier can be also be assigned to constraints when defining them. The following syntax can be used to do so:

```
<constraint identifier>:  <constraint>;
```

Assigning an identifier to constraints makes it possible to uniquely identify them and query additional information from the solver (e.g., retrieve dual variables and slacks).

Given these syntax rules, the following is an example of valid constraint definitions within an appropriate node and time horizon context:

```
#TIMEHORIZON
T = 2;
```

(continues on next page)

(continued from previous page)

```
#NODE mynode
#PARAMETERS
a = {2,4};
#VARIABLES
internal : x[T];
external : outflow[T];
#CONSTRAINTS
initial_constraint : x[0] >= 0;
x[1] >= 0;
x[2] <= a[1];
outflow[1] == sum(x[i] for i in [0:T-1]);
#OBJECTIVES
// objective definitions
```

Note that the variables and the parameter are only accessed at indices that are consistent with their definitions.

GBOML provides two options to specify expansion ranges and define vectorized constraints, namely user-defined and automatic expansions.

First, user-defined expansions can be constructed as follows:

```
<constraint> <expansion range>;
```

where <constraint> is an equality or inequality constraint and <expansion range> can be expressed using the for and where keywords, according to the following syntax rules:

```
<expansion range>:= for <identifier> in [<start>:<end>];
                 := for <identifier> in [<start>:<step>:<end>];
                 := for <identifier> in [<start>:<end>] where <condition>;
                 := for <identifier> in [<start>:<step>:<end>] where <condition>;
```

The first rule defines a constraint that is applied for all integral values of <identifier> that lie in the range between <start> and <end> (both included). Note that <start> must be smaller than <end> for the range to be non-empty. If an empty range is given, a warning will be raised. The <identifier> may be any identifier that has not been used to define a parameter or a variable in the present node block. The t identifier is reserved for automatic expansion (discussed below) and may not be used for user-defined expansions. The second rule makes use of the optional definition of a <step> that is used to increment through the range between <start> and <end>. The third and fourth rules are only extensions of the first two, where a certain condition needs to be satisfied for the constraint to be expanded. Recall that such conditions are defined in terms of expressions, comparison operators, and logical operators. For a condition to be valid, it must be possible to evaluate it for a given value of <identifier>. In particular, conditions may depend on <identifier> and parameters but must not depend on variables. In addition, the indices over which expansions take place must be valid for vectors of parameters and variables involved in vectorized constraints. More precisely, an index is valid if it is non-negative and does not exceed the size of said vector. If an index that is not valid is used in the expansion, an error is returned.

Second, automatic expansions can be declared by using the t identifier in a constraint. The constraint is then expanded over all valid indices $t \in \{0, ..., T-1\}$.

For example, the following vectorized constraint

```
x[t] >= x[t-5];
```

will only be expanded over $t \in \{5, ..., T-1\}$ since the right-hand side expression is ill-defined for $t < 5$. A warning is also raised to indicate the values of t over which the constraint cannot be expanded. Furthermore, a condition can

be added in automatic expansions. The corresponding syntax rule can be written as:

```
<constraint> <condition>;
```

where `condition` may depend on `t` and parameters.

The following is an example illustrating both expansion methods and making use of the keywords `for` and `where` in order to compactly write selectively-imposed constraints:

```
#TIMEHORIZON
T = 20;

#NODE mynode
#PARAMETERS
a = import "data.csv"; // parameter vector with 20 entries
#VARIABLES
internal : x[T];
external : outflow[T];
#CONSTRAINTS
nonnegativity : x[t] >= 0;
x[i] <= a[i] for i in [1:(T-2)/2];
0 <= a[i]*x[i] for i in [2:2:10] where i != 4;
x[t] == 0 where t == 0 or t == T-1;
outflow[0] == x[0];
outflow[t] == outflow[t-1] + x[t];
#OBJECTIVES
// objective definitions
```

While the syntax discussed above is sufficiently expressive to define nonlinear equality and inequality constraints, the GBOML parser expects constraints to be affine with respect to all variables involved. Hence, encoding nonlinear constraints leads to an error being raised.

5.3.4 Objectives

Objective definitions are given by an expression and a keyword indicating whether the objective should be minimized or maximized. The syntax rules for the definition of objectives are as follows:

```
min : <expression>;
max : <expression>;
min : <expression> <expansion range>;
max : <expression> <expansion range>;
```

At least one node in a given model must possess at least one objective but all nodes may have multiple objectives. In case multiple objectives are given in the same `#NODE` block, all objectives are aggregated into a single one by summing them (respecting the sign associated with the keywords `min` and `max`). Since the abstract GBOML problem is a minimization problem, the signs of objectives that should be maximized are inverted before summation.

Objectives can also be expanded in two ways, namely via user-defined and automatic expansions. First, user-defined expansions make use of an `<identifier>` that will be expanded over each value in the `<expansion range>`. Second, automatic expansions can be constructed by using the `t` identifier directly in the objective. Since all local objectives defined in the same `#NODE` block are eventually aggregated, the following objectives are in fact equivalent:

```
min : x[t], min : sum(x[i] for i in [0:T-1])
```

Similarly to constraints, identifiers can be assigned to objectives when defining them using the following syntax:

```
min <identifier>: <expression>;
max <identifier>: <expression>;
min <identifier>: <expression> <expansion range>;
max <identifier>: <expression> <expansion range>;
```

The previous example can be completed by defining an objective function, which yields a complete and valid `#NODE` block:

```
#TIMEHORIZON
T = 20;

#NODE mynode
#PARAMETERS
a = import "data.csv"; // parameter vector with 20 entries
#VARIABLES
internal : x[T];
external : outflow[T];
#CONSTRAINTS
x[t] >= 0;
x[i] <= a[i] for i in [1:T-2];
x[t] == 0 where t == 0 or t == T-1;
outflow[0] == x[0];
outflow[t] == outflow[t-1] + x[t];
#OBJECTIVES
max final_outflow: outflow[T-1];
```

As for constraint definitions, the syntax for objective definitions is sufficiently expressive to define nonlinear objectives. However, the GBOML parser expects all objectives to be affine with respect to all variables.

5.4 Hyperedges

A hyperedge typically couples variables belonging to different nodes via equality or inequality constraints (or both). Each hyperedge is defined using a dedicated code block. This code block must be started by either the `#HYPEREDGE` keyword or the :`#LINK` keyword (the two can be used interchangeably). A hyperedge must have a unique `<identifier>` and no two hyperedges or hyperedge and node may have the same identifier. In addition, a hyperedge may have its own parameters and constraints. Hence, valid hyperedge blocks have the following structure:

```
#HYPEREDGE <identifier 1>
#PARAMETERS
// parameter definitions
#CONSTRAINTS
// constraint definitions

#LINK <identifier 2>
#PARAMETERS
// parameter definitions
#CONSTRAINTS
// constraint definitions
```

Parameters and constraints are further described below.

5.4.1 Parameters

Parameters defined in a `#HYPEREDGE` block follow the exact same rules as the ones defined in `#NODE` blocks.

5.4.2 Constraints

While affine constraints involving all variables declared in a `#NODE` block can be defined in the same block, constraints defined in `#HYPEREDGE` blocks couple `external` variables associated with any subset of nodes. The syntax for defining constraints is otherwise the same as the one used in `#NODE` blocks:

```
#CONSTRAINTS
 <expression> == <expression> <expansion range>;
 <expression> <= <expression> <expansion range>;
 <expression> >= <expression> <expansion range>;
```

Similarly to the constraints defined in nodes, hyperedges can also be named by adding an identifier with colon defore the constraint, as follows,

```
<constraint identifier>:  <constraint>;
```

Given these syntax rules, the following is an example including valid hyperedge blocks (and associated `#NODE` blocks):

```
#TIMEHORIZON
// time horizon definition

#NODE node1
#VARIABLES
external : x;
external : inflow[T];
// further node content

#NODE node2
#VARIABLES
external : y;
external : outflow[T];
// further node content

#HYPEREDGE hyperedge1
#CONSTRAINTS
node1.inflow[t] == node2.outflow[t];

#NODE node3
#VARIABLES
external : z;
// further node content

#LINK hyperedge2
#PARAMETERS
weight = {1/3,2/3};
#CONSTRAINTS
node1.x <= weight[0]*node2.y + weight[1]*node3.z;
node2.y <= node3.z;
```

CHAPTER

SIX

ADVANCED FEATURES

Advanced features include hierarchical model definitions and imports.

These features are discussed in the following sections:

6.1 Hierarchical Models

In the hierarchical hypergraph abstraction underpinning the GBOML language, each node can itself be viewed as a hierarchical hypergraph. Nodes may therefore be constructed in a bottom-up fashion, from *sub-nodes* linked by *sub-hyperedges*.

Sub-nodes and sub-hyperedges are defined between the `#PARAMETERS` and `#VARIABLES` blocks of a parent node. Thus, a typical hierarchical block `#NODE` is structured as follows:

```
#NODE <parent node identifier>
#PARAMETERS
// parent parameter definitions

   #NODE <sub-node identifier 1>
   #PARAMETERS
   // sub-node 1 parameter definitions
   #VARIABLES
   // sub-node 1 variable definitions
   #CONSTRAINTS
   // sub-node 1 constraint definitions
   #OBJECTIVES
   // sub-node 1 objective definitions

   #NODE <sub-node identifier 2>
   #PARAMETERS
   // sub-node 2 parameter definitions
   #VARIABLES
   // sub-node 2 variable definitions
   #CONSTRAINTS
   // sub-node 2 constraint definitions
   #OBJECTIVES
   // sub-node 2 objective definitions

   #HYPEREDGE <sub-hyperedge identifier>
   #PARAMETERS
   // sub-hyperedge parameter definitions
```

(continues on next page)

(continued from previous page)

```
   #CONSTRAINTS
   // sub-hyperedge constraint definitions

#VARIABLES
// parent variable definitions
#CONSTRAINTS
// parent constraint definitions
#OBJECTIVES
// parent objective definitions
```

Information can be exchanged between different levels in the hierarchy, notably through parameters and variables. However, the direction in which information can be shared between levels depends on its nature, as discussed below.

Parameters can be only passed from the top down. Hence, parameters defined in a parent node can be accessed in any child node or sub-hyperedge by prefixing the identifier of the parent node in any expression involving this parameter. In other words, parent node parameters can be accessed in child nodes as follows:

```
<parent node identifier>.<parameter identifier>
```

Given these syntax rules, the following is a valid example of hierarchical parameter use (with three levels):

```
#NODE A
#PARAMETERS
parameter_A = 1;

   #NODE B
   #PARAMETERS
   parameter_B = 2;

      #NODE C
      #PARAMETERS
      parameter_C = 3;
      sum_parameters = A.parameter_A + B.parameter_B + parameter_C; // = 6
```

Note that indenting node blocks corresponding to different levels in the hierarchy is not mandatory but makes for easier reading.

In contrast to parameters, variables can only be passed from the bottom up. Thus, a parent node can define some of its variables using those of a child node as follows:

```
<parent node identifier> <- <child node identifier>.<variable identifier>;
<parent node identifier> <- <child node identifier>.<variable identifier>[<expression>];
```

Note that parent variables defined in such fashion must have the same type as the underlying child variables and vector variables must also have the same length. In addition, parent variables can only be defined from child variables one level down in the hierarchy.

Given these syntax rules, the following is a valid example of hierarchical variable use:

```
#NODE A

   #NODE B
   #VARIABLES
   internal : x[10];

   #NODE C
   #VARIABLES
   internal : x[10];

#VARIABLES
internal : y[10] <- B.x[10];
external : z[10] <- C.x[10];
```

These two examples can be combined to produce a valid hierarchical model example:

```
#TIMEHORIZON
T = 10;

#NODE A
#PARAMETERS
parameter_A = 1;

   #NODE B
   #PARAMETERS
   parameter_B = 1+A.parameter_A;
   #VARIABLES
   internal : x[10];
   #CONSTRAINTS
   x[t] >= parameter_B;

   #NODE C
   #PARAMETERS
   parameter_C = 2+A.parameter_A;
   #VARIABLES
   internal : x[10];
   #CONSTRAINTS
   x[t] >= parameter_C;

#VARIABLES
internal : y[10] <- B.x[10];
external : z[10] <- C.x[10];
#CONSTRAINTS
y[t]+z[t] >= 6;
#OBJECTIVES
min: y[t]+z[t];
```

6.2 Importing Nodes and Hyperedges

6.2.1 Importing Nodes

A node can be defined by importing an existing node from a GBOML input file. The imported node can be used as such or some of its attributes may be re-defined (e.g., parameter values may be changed) based on the following syntax rules:

```
#NODE <new node identifier> = import <imported node identifier> from <filename> ;
#NODE <new node identifier> = import <imported node identifier> from <filename> with <re-
↪definitions>
```

If the imported node sits at the top of the node hierarchy in the GBOML input file, its identifier should be used as such after the *import* keyword. However, if the imported node happens to be deeper in the hierarchy, a sequence of dot-separated identifiers corresponding to its ancestors should be prefixed to its own identifier. For example, for a GBOML file with a 2-level hierarchy, a child node could be imported by using the following identifier:

<parent node identifier>.<child node identifier>

When importing a node, two types of re-definitions are possible:

- Re-defining parameter values (i.e., changing the value of an existing parameter).
- Re-defining variable type (i.e., from external to internal or vice-versa).

To re-define the values of parameters originally defined in the imported node, their identifiers must be followed by equality signs and new values:

```
#NODE <new node identifier> = import <imported node identifier> from <filename> with
↪<parameter identifier> = <new parameter value>;
```

Note that the re-definition of a parameter may not change its type (i.e., vectors must remain vectors and likewise for scalars). In addition, several parameters can be re-defined in such fashion, provided that each assignment is followed by a semi-colon.

Variable type can be re-defined with the following rules:

```
<variable identifier> external;
<variable identifier> internal;
```

To illustrate these features, let *file1.txt* be a GBOML input file from which a node should be imported:

```
//file1.txt
#NODE Consumers
#PARAMETERS
total_number = 10;

  #NODE consumer_1
  #PARAMETERS
  price_per_unit = 5;
  avg_number_of_units = 100;
  #VARIABLES
  internal : delivery[T];

#VARIABLES
internal : consumer_1_delivery[T] <- consumer_1.delivery[T];
```

Let *consumer_1* be the identifier of the node that should be imported. This node can be imported in another file *file2.txt* as follows:

```
//file2.txt
#NODE average_consumer = import Consumers.consumer_1 from "file1.txt" with
  price_per_unit = 6;
  delivery external;
```

This block defines the *average_consumer* node by importing the *consumer_1* node from *file1.txt*. It re-defines its *price_per_unit* parameter in the process and also changes the type of the *delivery* variable from *internal* to *external*.

6.2.2 Importing Hyperedges

Hyperedges may also be imported from a GBOML file using similar rules:

```
#HYPEREDGE <new hyperedge identifier> = import <identifiers> from <filename>
#HYPEREDGE <new hyperedge identifier> = import <identifiers> from <filename> with <re-
↪definitions>
```

The first rule works just like the one described above for nodes. The second rule, however, differs in its possible re-definitions. More precisely, parameter values may be re-defined but variable types may not, since hyperedges do not have their own variables. However, the identifiers of nodes appearing in a hyperedge may be modified as follows:

```
<old node identifier> <- <new node identifier>;
```

This rule changes all occurrences of the old node identifier by the new identifier in the hyperedge.

To illustrate these features, let *file3.txt* be a GBOML input file from which a hyperedge should be imported:

```
//file3.txt
#NODE A
#VARIABLES
external : x[t];
#CONSTRAINTS
x[t]>= 2;
#OBJECTIVES
min: x[t];

#NODE B
#VARIABLES
external : x[t];
#CONSTRAINTS
x[t]>= 3;
#OBJECTIVES
min: x[t];

#HYPEREDGE H
#CONSTRAINTS
A.x[t] + B.x[t] >=6;
```

Let `H` be the identifier of the hyperedge that should be imported. Let us consider a second file named *file4.txt* in which `H` should be re-named as `H_1` and link two nodes named `C` and `D`. This file is given as follows:

```
//file4.txt
#NODE C
#VARIABLES
external : x[t];
#CONSTRAINTS
x[t]>= 5;
#OBJECTIVES
min: x[t];

#NODE D
#VARIABLES
external : x[t];
#CONSTRAINTS
x[t]>= 6;
#OBJECTIVES
min: x[t];

#HYPEREDGE H_1 = import H from "file1.txt" with
  A <- C;
  B <- D;
```

The last code block imports the hyperedge `H` and re-names all occurrences of node `A` by node `C` and node `B` by node `D`. It is therefore equivalent to defining:

```
#HYPEREDGE H_1
#CONSTRAINTS
C.x[t] + D.x[t] >= 6;
```

CHAPTER

SEVEN

USEFUL IDIOMS

Several modeling needs may be readily addressed by using particular idioms that may not be immediately apparent from the GBOML syntax rules. Such modeling needs are discussed and appropriate idioms to address them are discussed next.

7.1 Repeating Data

In some cases, the modeling task calls for repeating data. For example, a model may cover a time horizon of multiple days but the behavior of certain model components may be the same for each day. This case is illustrated in the following example, along with the appropriate idiom for encoding the repeating model behavior.

```
#TIMEHORIZON
T = 5*24; // 5 days with hourly resolution
#NODE factory
#PARAMETERS
production = import "production.csv"; // 24 values for hourly production
#VARIABLES
external : outflow[T];
#CONSTRAINTS
outflow[t] == production[mod(t,24)];
#OBJECTIVES
// objective definitions
```

7.2 Round Down Integer Division

Integer division is not natively implemented in GBOML but can be a very useful tool for many modeling tasks. To illustrate this, let us consider a model whose time horizon spans several days and some of whose components involve indices that correspond to hours and days, respectively. An example of such problems is given below with the appropriate idiom for encoding the different index behaviors.

```
#TIMEHORIZON
T = 365*24; // 365 days with hourly resolution

#GLOBAL
days = T/24;

#NODE bank
#PARAMETERS
```

(continues on next page)

(continued from previous page)

```
interest_rate = import "interest_rate.csv"; // 365 values for daily interest rates
mean_interest = sum(interest_rate[i] for i in [0:global.days-1])/global.days; // mean␣
↪interest rate
#VARIABLES
internal : investment_interest[T];
internal : investment[T];
#CONSTRAINTS
investment_interest[i] == interest_rate[(i-mod(i,24))/24]*investment[i] for i in [0:T-1];
```

CHAPTER

EIGHT

HOW TO USE

Two interfaces to GBOML exist, namely a command-line interface and a Python API. They are described in the sections listed below:

8.1 Command Line Interface

The GBOML parser can be called from the command line by typing the following commands in a terminal window:

```
gboml <file> <options>
```

where `<file>` is the name of the file to be considered and `<options>` corresponds to one or several optional flags that can be activated.

The options are the following :

- *Print tokens*: the tokens output by the lexer can be printed with:

```
--lex
```

- *Print the syntax tree*: the syntax tree generated by the parser can be printed with:

```
--parser
```

- *Print the matrices*: the coefficient, right-hand side and objective matrices and vectors can be printed with:

```
--matrix
```

- *Linprog*: the linprog solver can be used with:

```
--linprog
```

The linprog solver for (continuous) linear programming comes with scipy and therefore does not require any installation or license. It is much less powerful than other solver options and is therefore only recommended for testing purposes (e.g., make sure that GBOML was properly installed).

- *Gurobi*: Gurobi can be invoked with:

```
--gurobi
```

- *CPLEX*: CPLEX can be used with:

```
--cplex
```

- *CPLEX Benders*: CPLEX Benders can be used with:

```
--cplex_benders
```

- *Xpress*: Xpress can be used with:

```
--xpress
```

- *Highs*: Highs can be used with:

```
--highs
```

- *Clp*: Clp can be used with:

```
--clp
```

It interfaces with CLP and CBC via CyLP.

- *Cbc*: Cbc can be used with:

```
--cbc
```

It interfaces with CLP and CBC via a custom made experimental interface (it allows to set model parameters)

- *DSP Dantzig-Wolfe*: the DSP implementation (experimental) of the Dantzig-Wolfe algorithm can be invoked with:

```
--dsp_dw
```

- *DSP Extensive form*: the DSP Extensive Form (i.e., flattened model) algorithm can be used with:

```
--dsp_de
```

- *Solver option parameters*: solver parameters can be set via a ".opt" file with:

```
--opt <opt_file>
```

where <opt_file> is the name of the file containing the solver parameters. If no parameters are provided, the default solver parameters are used.

- *Solver library path*: solver library path for DSP, CBC and HiGHS can be set by with:

```
--solver_lib <path_to_library>
```

where <path_to_library> is the library path to read. If solver_lib is not set, the default solver on the library PATH is used.

- *CSV*: the solution can be printed to a CSV file on a row basis (e.g., one variable per row):

```
--row_csv
```

- *Transposed CSV*: the solution can be printed to a CSV file on a column basis (e.g., one variable per column):

```
--col_csv
```

- *JSON*: the solution can be printed to a JSON file with:

```
--json
```

- *Detailed*: the solution provided by the solver along with auxiliary information (e.g., dual variables, slacks or basis ranges) can be printed to a JSON or CSV file with:

```
--detailed
```

- *Multi-processing*: the number of processes used for model generation can be controlled via:

```
--nb_processes <number>
```

where <number> is an integer, whose default value is 1.

- *Output*: the name of the output file can be defined with:

```
--output <output_filename>
```

where <output_filename> is the output filename without the extension (CSV or JSON). The default output name is the name of the GBOML file with the date and chosen extension.

8.2 Python Interface

class gboml.**GbomlGraph**(*timehorizon=1*)

GbomlGraph makes it possible to define and solve a GBOML model.

The GbomlGraph class enables the construction of GBOML models by importing nodes and hyperedges from a GBOML file. It also possesses a set of functions for updating the imported nodes and hyperedges (e.g., re-defining parameters or the type of variables). Nodes and hyperedges can be added to a GbomlGraph instance, from which the model can be solved and generated.

Parameters

timehorizon (*int*) – length of optimization horizon considered

Variables

- **list_nodes** – nodes included in model
- **list_hyperedges** – hyperedges included in model
- **timehorizon** – optimization horizon object
- **node_hyperedge_dict** – dictionary of all nodes and hyperedges
- **program** – Program class of generated model (= None if model not generated)
- **matrix_a** – constraint matrix A in sparse COO format (= None if model not generated)
- **matrix_b** – upper bound on each row in constraint matrix (i.e., right-hand side coefficients, = None if model not generated)
- **vector_c** – vector of objective coefficients (= None if model not generated)
- **indep_term_c** – objective offset (i.e., constant term in the objective, = None if model not generated)

add_global_parameter(*identifier*, *value*)

Add one global parameter objects to the graph

Parameters

- **identifier** (*str*) – parameter name
- **value** (*int* | *float* | *str*) – value associated to the parameter (if string it expects a filename to read from)

Returns:

add_global_parameters(*global_parameters*)

Add global parameters objects to the graph

Parameters
global_parameters (*list*) – list of Parameter object or tuples of <parameter_name, values> where values can be an int/float/list <int/float> or a string to import from.

Returns:

add_global_parameters_objects(*global_parameters*)

Add global parameters objects from list of parameters objects Warning this function will be removed in future release ! Use add_global_parameters instead

Parameters
global_parameters (*list*) – list of global parameters

Returns:

add_hyperedges_in_model(**hyperedges*)

bound method adding hyperedges to a GbomlGraph instance

Parameters
hyperedges (*list <Hyperedge>*) – list of hyperedge objects to be added

add_nodes_in_model(**nodes*)

bound method adding nodes to a GbomlGraph instance

Parameters
nodes (*list <Nodes>*) – list of node objects to be added

static add_sub_hyperedge(*hyperedge_to_add*, *in_node*)

static method adding a sub-hyperedge to a given node

Parameters

- **hyperedge_to_add** (*Hyperedge*) – sub-hyperedge to add
- **in_node** (*Node*) – node to which sub-hyperedge should be added

static add_sub_node(*node_to_add*, *in_node*)

static method adding a child node to a given node

Parameters

- **node_to_add** (*Node*) – sub-node to add
- **in_node** (*Node*) – node to which sub-node should be added

build_model(*nb_processes: int = 1*)

bound method generating the matrices of the optimization model

Parameters
nb_processes (*int*) – number of processes used for model generation

static change_node_name_in_hyperedge(*hyperedge*, *old_node_name*, *new_node_name*)

static method changing the name of a node appearing in the constraints of a given hyperedge

Parameters

- **hyperedge** (*Hyperedge*) – Hyperedge in which node names should be changed
- **old_node_name** (*str*) – previous node name
- **new_node_name** (*str*) – new node name

static change_type_variable_in_node(*node*, *variable_name: str*, *variable_type*)

static method changing the type of a variable

Parameters

- **node** (*Node*) – node to which variable that should be modified belongs
- **variable_name** (*str*) – variable name
- **variable_type** (*VariableType*) – new variable type (either External or Internal)

static create_parameter(*parameter_name*, *value*)

static method that returns a parameter whose name is given by parameter_name and expression by value

Parameters

- **parameter_name** (*str*) – parameter name
- **value** (*float/int/list<int/float>*) – value of parameter

Returns

param (*Parameter*) – parameter created

static get_object_in_node(*in_node*, **node_identifier: str*, *wanted_type=None*)

static method returning a node or a hyperedge given an ancestor node

Parameters

- **in_node** (*Node*) – node to which the sub-node is expected to belong
- **node_identifier** (*list <str>*) – list of ancestor node names, with first name corresponding to first sub-node and last name corresponding to node to retrieve
- **wanted_type** (*Class Type*) – either Node or Hyperedge depending on the type of the object considered

Returns

retrieved_object (*Node/Hyperedge*) – retrieved node or hyperedge

get_timehorizon()

bound method returning the value of the time horizon

Returns

value (*int*) – length of time horizon considered

static import_all_nodes_and_edges(*filename*, *cache=True*)

static method importing all nodes and hyperedges contained in a file

Parameters

- **filename** (*str*) – path to GBOML input file
- **cache** (*bool*) – activate caching all the hypergraphs read during import by default set to true

Returns

- **all_nodes** (*list*) – list of nodes contained in file
- **all_hyperedges** (*list*) – list of hyperedges contained in file
- **all_global_param** (*list*) – list of global parameters in file

static import_hyperedge(*filename: str*, **imported_hyperedge_identifier: str*, *new_hyperedge_name: str = ''*, *copy=True*, *cache=True*)

static method importing a hyperedge from a GBOML input file

Parameters

- **filename** (*str*) – path to GBOML input file
- **imported_hyperedge_identifier** (*list <str>*) – list of ancestor node names and hyperedge name (used for depth- first traversal)
- **new_hyperedge_name** (*str*) – new hyperedge identifier (for re-naming purposes, optional)
- **copy** (*bool*) – keyword argument defining whether a shallow or deep copy of the imported node is created (defaults to True, which produces a deepcopy)
- **cache** (*bool*) – activate caching all the hypergraphs read during import by default set to true

Returns

imported_hyperedge (*Hyperedge*) – imported hyperedge

static import_node(*filename: str*, **imported_node_identifier: str*, *new_node_name: str = ''*, *copy=True*, *cache=True*)

static method importing a node from a GBOML input file

Parameters

- **filename** (*str*) – path to GBOML input file
- **imported_node_identifier** (*list <str>*) – list of ancestor node names and node name (used for depth-first traversal)
- **new_node_name** (*str*) – new identifier of node (for re-naming purposes, optional)
- **copy** (*bool*) – keyword argument defining whether a shallow or deep copy of the imported node is created (defaults to True, which produces a deepcopy)
- **cache** (*bool*) – activate caching all the hypergraphs read during import by default set to true

Returns

imported_node (*Node*) – imported node

static modify_parameter_value(*parameter*, *value*)

Modify the value of parameter

Parameters

- **parameter** (*Parameter*) – parameter to modify
- **value** (*int|float|list<float/int>*) – value associated to the parameter

Returns:

static redefine_parameters_from_keywords(*node_or_hyperedge*, ***kwargs*)

static method re-defining parameter values from keyword arguments

Parameters

- **node_or_hyperedge** (*Node/Hyperedge*) – Node/Hyperedge in which parameters should be re-defined
- **kwargs** (*tuple <str, value>*) – tuple of parameters name, value

static redefine_parameters_from_list(*node_or_hyperedge*, *list_parameters: list*, *list_values: list*)

static method re-defining parameter values from a list

Parameters

- **node_or_hyperedge** (*Node/Hyperedge*) – Node/Hyperedge in which parameters should be re-defined
- **list_parameters** (*list <str>*) – list of parameter names
- **list_values** (*list <float> | list <float> | <str>*) – list of parameter values

static remove_constraint(*node_or_hyperedge*, **to_delete_constraints_names*)

static method removing constraints from a node/hyperedge

Parameters

- **node_or_hyperedge** (*Node/Hyperedge*) – Node/Hyperedge from which constraints should be removed
- **to_delete_constraints_names** (*list <str>*) – names of constraints to remove

static remove_objective_in_node(*node*, **to_delete_objectives_names*)

static method removing objectives from a node

Parameters

- **node** (*Node/Hyperedge*) – node from which objectives should be removed
- **to_delete_objectives_names** (*list <str>*) – names of objectives to remove

static rename(*node_or_hyperedge*, *new_name*)

static method re-naming a node or hyperedge

Parameters

- **node_or_hyperedge** (*Node/Hyperedge*) – node or hyperedge to be re-named
- **new_name** (*str*) – new name

static set_parsing_cache_limit(*size*)

sets a limit to the global cache.

Parameters
size – The cache size

set_timehorizon(*value*)

bound method setting the time horizon to a specified value

Parameters
value (*int*) – length of time horizon considered

`solve_cbc`(*opt_file=None*, *opt_dict=None*)

bound method solving the flattened optimization problem with Cbc

Parameters

- **opt_file** (*str*) – filename of file containing the optimization parameters
- **opt_dict** (*dict*) – dictionary containing the optimization parameters the key must be the parameter to tune the value a tuple of the <type, value> example: {"gap": ["double", 0.5]

Returns

- **solution** (*ndarray*) – flattened solution
- **objective** (*flat*) – float of the objective value
- **status** (*str*) – solver exit status
- **solver_info** (*dict*) – dictionary storing solver information

`solve_clp`()

bound method solving the flattened optimization problem with Clp

Returns

- **solution** (*ndarray*) – flattened solution
- **objective** (*flat*) – float of the objective value
- **status** (*str*) – solver exit status
- **solver_info** (*dict*) – dictionary storing solver information

`solve_cplex`(*opt_file: str = None*, *details=False*, *opt_dict=None*)

bound method solving the flattened optimization model with CPLEX

Parameters

- **opt_file** (*str*) – path to an optimization parameters file
- **details** (*bool*) – get variables and constraints information
- **opt_dict** (*dict*) – dictionary containing the optimization parameters

Returns

- **solution** (*ndarray*) – flattened solution
- **objective** (*float*) – objective value
- **status** (*str*) – solver exit status
- **solver_info** (*dict*) – dictionary storing solver information
- **constraints_information** (*dict*) – dict of additional information concerning constraints
- **variables_information** (*dict*) – dict of additional information concerning variables

`solve_dsp`(*algorithm='dw'*)

bound method solving the optimization model with DSP

Parameters

algorithm (*str*) – algorithm selected ("dw" for Dantzig-Wolfe and "de" for extensive form solve)

Returns

- **solution** (*ndarray*) – flattened solution

- **objective** (*float*) – objective value
- **status** (*str*) – solver exit status
- **solver_info** (*dict*) – dictionary of solver information

`solve_gurobi`(*opt_file: str = None*, *details=False*)

bound method solving the flattened optimization model with Gurobi

Parameters

- **opt_file** (*str*) – path to an optimization parameters file
- **details** (*bool*) – get variables and constraints information

Returns

- **solution** (*ndarray*) – flattened solution
- **objective** (*float*) – objective value
- **status** (*str*) – solver exit status
- **solver_info** (*dict*) – dictionary storing solver information
- **constraints_information** (*dict*) – dict of additional information concerning constraints
- **variables_information** (*dict*) – dict of additional information concerning variables

`solve_highs`()

bound method solving the flattened optimization problem with Highs

Returns

- **solution** (*ndarray*) – flattened solution
- **objective** (*flat*) – float of the objective value
- **status** (*str*) – solver exit status
- **solver_info** (*dict*) – dictionary storing solver information

`solve_xpress`(*opt_file: str = None*, *details=False*)

bound method solving the flattened optimization model with Xpress

Parameters

- **opt_file** (*str*) – path to an optimization parameters file
- **details** (*bool*) – get variables and constraints information

Returns

- **solution** (*ndarray*) – flattened solution
- **objective** (*float*) – objective value
- **status** (*str*) – solver exit status
- **solver_info** (*dict*) – dictionary storing solver information
- **constraints_information** (*dict*) – dict of additional information concerning constraints
- **variables_information** (*dict*) – dict of additional information concerning variables

turn_solution_to_dictionary(*solver_data*, *status*, *solution*, *objective*, *constraint_info=None*, *variables_info=None*)

bound method converting the flat solution to a structured dictionary

Parameters

- **solver_data** (*dict*) – dictionary of solver information
- **status** (*str*) – solver exit status
- **solution** (*ndarray*) – flattened solution
- **objective** (*float*) – objective value
- **constraint_info** (*dict*) – dict of additional information concerning constraints
- **variables_info** (*dict*) – dict of additional information concerning variables

Returns

gathered_data (*dict*) – structured dictionary containing all the solution information

turn_solution_to_list(*solution*, *constraints_info=None*)

bound method converting the flat solution to a list of <name, value> tuples

Parameters

- **solution** (*ndarray*) – flattened solution
- **constraints_info** (*dict*) – dict of additional information concerning constraints

Returns

output_list (*list*) – list of <name, value> tuples

8.3 Solver APIs

The GBOML parser interfaces with a variety of open source and commercial solvers in order to solve optimization models. Direct access to their API is provided for several solvers, allowing users to tune algorithm parameters and query complementary information (e.g., dual variables, slacks or basis ranges, when available).

Solver parameters must be placed in file named *solver_name.opt*. This file must be passed via the *–opt* option of the command-line interface or directly as an argument in the Python API. More information about solver parameters can typically be found on the website of the respective solver (e.g., for Gurobi).

The list of attributes that may be queried from the different solvers can be found below:

- Gurobi: the following constraint attributes can be queried from the solver: *Pi* (dual variables), *Slack* (slack variables), *CBasis* (whether slack variable is in simplex basis), *SARHSLow* (right-hand side basis sensitivity information), *SARHSUp* (right-hand side basis sensitivity information). In addition, the following variable attributes can be queried (most of them useful for sensitivity analyses): *RC* (reduced cost), *VBasis*, *SAObjLow*, *SAObjUp*, *SALBLow*, *SALBUp*, *SAUBLow*, *SAUBUp*. More details can be found on the Gurobi website.
- CPLEX: the following constraint attributes can be queried: dual and slack variables. The following variable attributes can be queried: basis information and dual norms.
- Xpress: the following constraint attributes can be queried: dual and slack variables. The following variable attributes can be queried: reduced cost.
- Cbc/Clp: the Cbc/Clp API is still under development with a new interface where multiple parameters can be passed to the solver but no variable attributes can be queried yet.
- Highs: the Highs API is still under development with a new interface where multiple parameters can be passed to the solver but no variable attributes can be queried yet.

Note that all of these attributes are automatically queried and printed in the detailed JSON/CSV file, when requested.

CHAPTER

NINE

EXAMPLES

This section is divided in two parts. The first part is dedicated to three examples. The first example deals with a microgrid system design problem and illustrates the basic features of GBOML. The second example is more sophisticated and focuses on a remote carbon-neutral fuel supply chain planning problem. Finally, the third example is based on a hypothetical problem and illustrates the Python API. The second part provides a list of reference papers and models that use GBOML.

9.1 Microgrid Example

9.1.1 Problem Description

A grid-connected microgrid is a small-scale and (ideally) self-sufficient electric power system. It consists of an interconnection of electric generators (e.g., solar panels or fossil fuel generators) and loads (the set of electricity consumers). An electrical storage system is often added to the system in order to balance electricity production and consumption in time while limiting the dependence on the distribution network. The configuration of the microgrid system is shown in Fig. 9.1.

In this section, we study the problem of sizing an electric microgrid similar to the one shown in Fig. 9.1. The aim of the sizing problem is to determine the amount of solar panel and battery storage capacity required to minimize the cost of serving pre-specified electricity demand levels over the lifetime of the system. Hence, both investment and operating costs are taken into account. In addition, the electricity consumed in the microgrid and the solar irradiation of the panels are assumed known for a typical representative day.

9.1.2 GBOML Implementation

The system is composed of four nodes implementing the behavior of the elements of the microgrid. The first node corresponds to solar panels. The second node represents the dynamics and costs of a battery based on the power flows that charge and discharge it. The third node models the consumer load. The last node represents the electricity distribution network. The power balance of the system is represented via a hyperedge. A skeleton of the optimization problem expressed in the language is provided below. First, the horizon T is defined as the number of hours over the lifetime of the system, which is assumed to be twenty years. Then, the four nodes are implemented. Finally, these nodes are linked via a hyperedge.

```
#TIMEHORIZON
T = 20 * 365 * 24; // number of hours in twenty years

#NODE SOLAR_PV
// Implementation of solar panel node
```

(continues on next page)

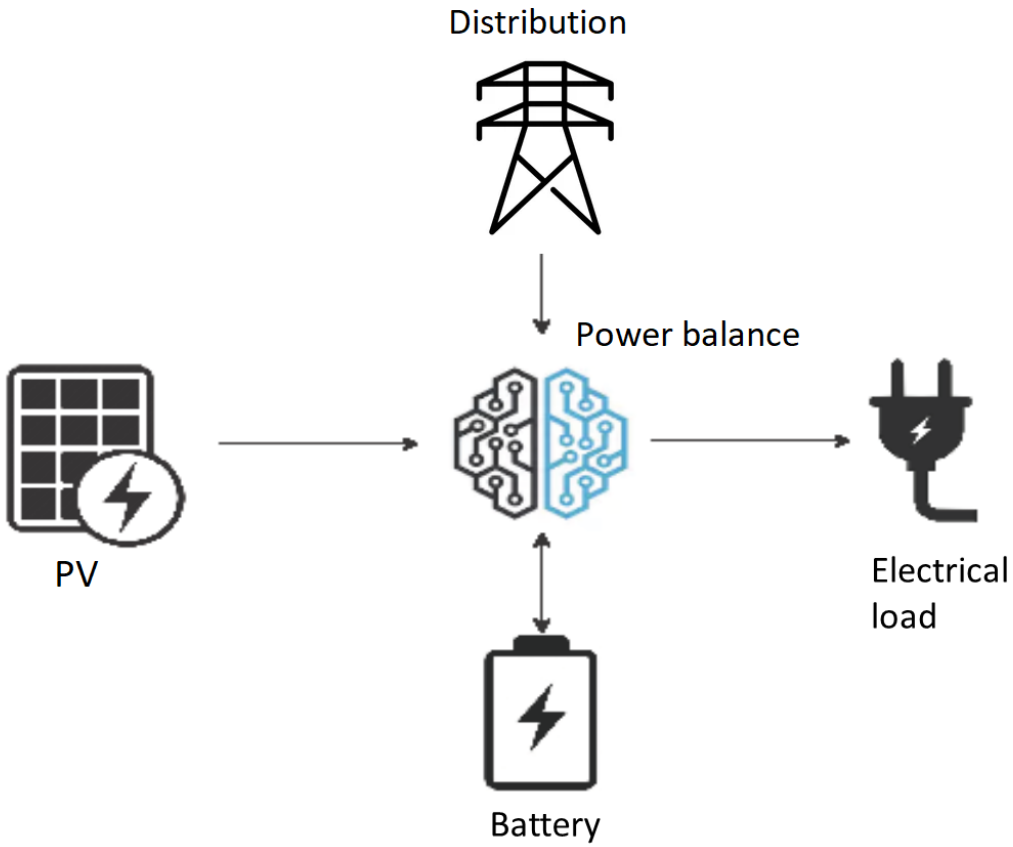

Fig. 9.1: Microgrid system configuration.

(continued from previous page)

```
#NODE BATTERY
// Implementation of battery node

#NODE DEMAND
// Implementation of demand node

#NODE DISTRIBUTION
// Implementation of distribution network node

#HYPEREDGE POWER_BALANCE
// Implementation of power balance hyperedge
```

Let us now discuss the implementation of the four nodes one by one. In what follows, Latin letters denote optimization variables, while Greek letters are used for parameters.

A solar panel is a technology that harnesses solar radiation for electricity production. The installed capacity of solars panels $\bar{P}^{PV} \in \mathbb{R}^+$ (in watt) determines the maximum amount of power that they may produce at any point in time. The investment cost $I^{PV} \in \mathbb{R}^+$ (in some currency) can be calculated as the product of the installed capacity and the capital expenditure (CAPEX) $\iota^{PV} \in \mathbb{R}_+$ associated with the deployment of one unit of capacity (in currency/watt):

$$I^{PV} = \iota^{PV} \cdot \bar{P}^{PV}.$$

At time t, the maximum power that may be generated by the solar panels is equal to the product of the installed capacity $\bar{P}^{PV}$ and a dimensionless capacity factor parameter $\pi_t^{PV} \in [0, 1]$, which is computed based on the irradiance at time t and the PV technology at hand. In addition, the power can be curtailed so that the actual power production $p_t^{PV} \in \mathbb{R}^+$ (in watt) can be smaller than the maximum power that may be generated by the panels. The latter is captured by the following inequality constraint involving the power injected into the microgrid and the maximal power generated by the solar panels, the constraint being tight when no curtailment occurs:

$$p_t^{PV} \leq \pi_t^{PV} \cdot \bar{P}^{PV}, \; t = 0, \ldots, T-1.$$

The node describing the solar panels is implemented below. This node has two scalar internal variables: one for the capacity $\bar{P}^{PV}$ and one for the investment cost I^{PV}. A time-dependent external variable implements the power generated by the panels p_t^{PV}. Constraints are used to define the investment cost and the power output of the solar panels and enforce the nonnegativity of optimization variables. Finally, the objective to be minimized is the investment cost I^{PV}.

```
#NODE SOLAR_PV
#PARAMETERS
capex = 600; // capital expenditure per unit capacity
capacity_factor = import "pv_gen.csv";
#VARIABLES
internal: capacity;
internal: investment_cost;
external: electricity[T];
#CONSTRAINTS
capacity >= 0;
electricity[t] >= 0;
electricity[t] <= capacity_factor[mod(t, 24)] * capacity;
investment_cost == capex * capacity;
#OBJECTIVES
min: investment_cost;
```

A battery is an electrical device that can store energy. The installed capacity $\bar{E}^B \in \mathbb{R}^+$ (in watt-hour) defines the maximum amount of energy that may be stored in the battery. Similarly to the solar panels, a capital expenditure ι^B (in

currency/watt-hour) is associated with the deployment of one unit of battery storage capacity, such that the investment cost $I^{PV} \in \mathbb{R}^+$ is computed as follows:

$$I^B = \iota^B \cdot \bar{E}^B.$$

Energy can be charged or discharged from the battery by letting power flow in or out of the battery. The charging power and discharging power are denoted by $p_t^{B+} \in \mathbb{R}^+$ (in watt) and $p_t^{B-} \in \mathbb{R}^+$ (in watt), respectively. The energy stored in the battery $e_t^B \in \mathbb{R}^+$ (in watt-hour), which is sometimes referred to as the *state of charge* of the battery, is upper-bounded by the installed capacity:

$$e_t^B \leq \bar{E}^B, \ t = 0, \ldots, T-1.$$

In addition, the state of charge is linked to the power flowing in and out of the battery through the following constraint:

$$e_{t+1}^B = e_t^B + \eta \cdot p_t^{B+} - \frac{p_t^{B-}}{\eta}, \ t = 0, \ldots, T-2,$$

where $\eta \in [0, 1]$ is the efficiency of the battery, which is a parameter quantifying the energy lost when charging and discharging the battery. Finally, it is common to impose that the energy stored in the battery at the beginning of the time horizon is equal to the energy stored in the battery at the end of it, in order to avoid spurious transient effects in storage operation close to the beginning and the end of the time horizon:

$$e_0^B = e_{T-1}^B.$$

In addition to two scalar internal variables representing the installed capacity $\bar{E}^B$ and the investment cost I^B, a time-dependent internal variable is used for the energy stored in the battery e_t^B. Furthermore, the charge p_t^{B+} and discharge p_t^{B-} power flows are defined as time-dependent external variables of the battery node. Finally, the investment cost I^B is minimized. The implementation is provided below.

```
#NODE BATTERY
#PARAMETERS
capex = 150; // capital expenditure per unit capacity
efficiency = 0.75;
#VARIABLES
internal: capacity;
internal: investment_cost;
internal: energy[T];
external: charge[T];
external: discharge[T];
#CONSTRAINTS
capacity >= 0;
energy[t] >= 0;
charge[t] >= 0;
discharge[t] >= 0;
energy[t] <= capacity;
energy[t+1] == energy[t] + efficiency * charge[t] - discharge[t] / efficiency;
energy[0] == energy[T-1];
investment_cost == capex * capacity;
#OBJECTIVES
min: investment_cost;
```

In the demand node shown below, the electrical consumption $p_t^C \in \mathbb{R}^+$ (in watt) is computed for each time t based on a time series provided as a parameter and giving the typical consumption for the 24 hours of a representative day. No objective is required.

```
#NODE DEMAND
#PARAMETERS
demand = import "demand.csv";
#VARIABLES
external: consumption[T];
#CONSTRAINTS
consumption[t] == demand[mod(t, 24)];
```

The distribution node represents the distribution network to which the microgrid is connected. It is possible to buy power $p_t^D \in \mathbb{R}^+$ (in watt) from the grid to make up for any power shortage that may occur in the microgrid at time t. This power is bought at a marginal price θ^D (in currency/watt) such that the operating cost $o_t^D \in \mathbb{R}_+$ at time t is given by:

$$o_t^D = \theta^D \cdot p_t^D,\ t = 0, \ldots, T-1.$$

In this node, the total operating cost $O^D \in \mathbb{R}_+$ over the lifetime of the system is minimized, and the objective function is thus the following:

$$O^D = \sum_{t=0}^{T-1} o_t^D.$$

This node is implemented below. The imported power p_t^D is modeled by a time-dependent external variable. Moreover, the operating cost o_t^D is computed using a time-dependent internal variable and the total operating cost O^D is minimized.

```
#NODE DISTRIBUTION
#PARAMETERS
electricity_price = 0.05;
#VARIABLES
internal: operating_cost[T];
external: power_import[T];
#CONSTRAINTS
power_import[t] >= 0;
operating_cost[t] == electricity_price * power_import[t];
#OBJECTIVES
min: operating_cost[t];
```

All nodes are connected via a hyperedge implementing an equality constraint that represents the balance between electricity production and consumption in the microgrid. Hence, the sum of the solar production p_t^{PV}, the power discharged from the battery p_t^{B-} and the power bought from the distribution network p_t^D must be equal to the sum of the power charged in the battery p_t^{B+} and the power consumed p_t^C by loads and appliances. In other words, the following constraint enforces the power balance in the microgrid:

$$p_t^{PV} + p_t^{B-} + p_t^D = p_t^{B+} + p_t^C,\ t = 0, \ldots, T-1.$$

This hyperedge can be implemented as follows:

```
#HYPEREDGE POWER_BALANCE
#CONSTRAINTS
SOLAR_PV.electricity[t] + BATTERY.discharge[t] + DISTRIBUTION.power_import[t] == BATTERY.
↪charge[t] + DEMAND.consumption[t];
```

Finally, the complete model is obtained by substituting the code blocks of all nodes in the skeleton code introduced earlier. The model is then translated using the GBOML compiler and solved with Gurobi. For the optimal configuration,

the objective function is such that:

$$\min \underbrace{\iota^{PV} \cdot \bar{P}^{PV}}_{\text{Investment PV}} + \underbrace{\iota^{B} \cdot \bar{E}^{B}}_{\text{Investment battery}} + \underbrace{\sum_{t=0}^{T-1} \theta^{D} \cdot p_t^{D}}_{\text{Power Imports}} \approx 5.6 \times 10^4.$$

9.1.3 Running the Example

There are two ways of running the microgrid example:

- From the command line: first, you need to go to the GBOML directory, open a terminal window and type the following commands,

```
gboml examples/microgrid/microgrid.txt --cplex --json --output microgrid_example
```

This will solve the microgrid problem using CPLEX and save the solution in "examples/microgrid/microgrid_example.json".

- From Python: execute the following Python code,

```
from gboml import GbomlGraph

gboml_model = GbomlGraph(24*365)
nodes, edges, _ = gboml_model.import_all_nodes_and_edges("path_to_GBOML_directory/
↪examples/microgrid/microgrid.txt")
gboml_model.add_nodes_in_model(*nodes)
gboml_model.add_hyperedges_in_model(*edges)
gboml_model.build_model()
solution = gboml_model.solve_cplex()
print(solution)
```

The solution of this example is printed in the terminal.

9.2 Remote Hub Example

9.2.1 Problem Description

The production of carbon-neutral synthetic fuels in remote areas where high-quality renewable resources are abundant has long been viewed as a means of developing a cost-effective, decarbonised energy supply for countries with limited local renewable potential (see for instance Hashimoto et al.). The synthesis of carbon-neutral fuels relies on a set of tightly-integrated technologies implementing various chemical processes. In order to properly estimate the cost of the final product, the entire supply chain must be modelled in an integrated fashion, from remote electricity production to product delivery at the destination.

From a modeling perspective, remote carbon-neutral fuel supply chains can be naturally represented as hypergraphs where each node models a technology or process and has a low degree (i.e., each technology only interacts with a small subset of all technologies and processes), as depicted below for the particular case of synthetic methane. Moreover, for the purpose of strategic techno-economic analyses, each process can be described using one of only two simple generic nodes, namely *conversion* and *storage* nodes. Roughly speaking, conversion nodes represent technologies implementing physical processes that enable the transformation of commodities, while storage nodes model technologies that can hold commodities over time and restore them when needed. On the other hand, the relationship between nodes

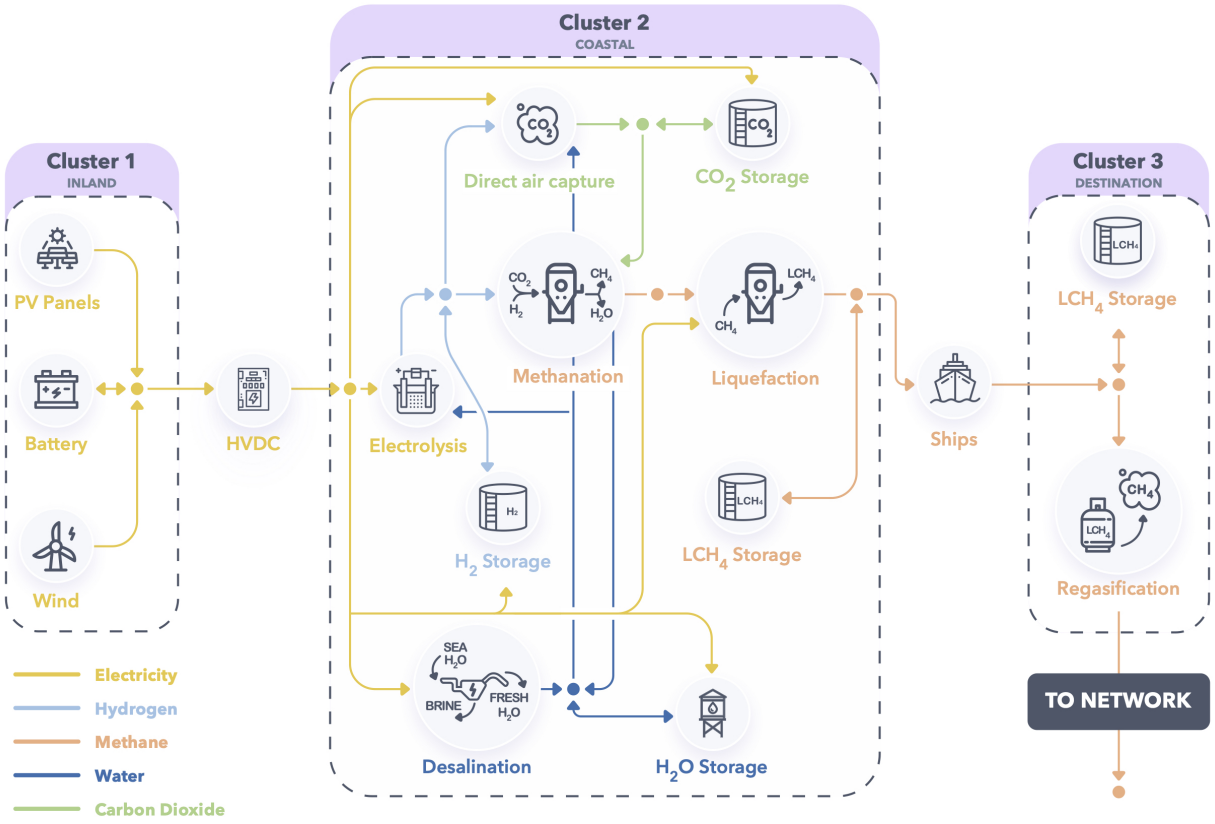

Fig. 9.2: Remote hub system configuration.

is expressed via so-called *conservation* hyperedges that enforce the conservation of flows of commodities between conversion and storage nodes. These nodes and hyperedges are simple to encode in GBOML, which therefore provides a convenient way of building integrated carbon-neutral fuel supply chain models.

The modeling framework was leveraged in a recent paper to study the economics of producing carbon-neutral synthetic methane from solar and wind energy in North Africa and exporting it to Northwestern Europe. The supply chain schematically represented in Fig. 9.2 was modelled in an integrated fashion, and each technology in the supply chain was sized based on an operational horizon of five years with hourly resolution in order to minimize total system cost. The full supply chain model is described in detail in the paper, along with the data required to instantiate it. Results suggest that total system costs would be around 1.5 BEUR/year (over the lifetime of the system) by 2030 for systems producing 10 TWh (higher heating value) of synthetic methane annually using a combination of solar and wind power plants (assuming a weighted average cost of capital of 7%), respectively, resulting in carbon-neutral synthetic methane costs around 150 EUR/MWh. A comprehensive sensitivity analysis is also carried out in order to assess the impact of various techno-economic parameters and assumptions on synthetic methane cost, including the availability of wind power plants, the investment costs of electrolysis, methanation and direct air capture plants, their operational flexibility, the energy consumption of direct air capture plants, and financing costs. The most expensive configuration (around 200 EUR/MWh) relies on solar photovoltaic power plants alone, while the cheapest configuration (around 88 EUR/MWh) makes use of a combination of solar PV and wind power plants and is obtained when financing costs are set to zero. The input files encoding the model in GBOML and enabling the replication of these results are available in the GBOML repository.

9.2.2 Running the Example

There are two ways of running the remote hub example:

- From the command line: first, you need to go to the GBOML directory, open a terminal and write the following,

```
gboml examples/remote_energy_supply_chain/remote_hub_nowacc.txt --cplex --json --output␣
↪remote_hub_example
```

This will solve the remote hub problem using CPLEX and save the solution in "examples/remote_energy_supply_chain/remote_hub_example.json".

- From Python: execute the following Python code,

```
from gboml import GbomlGraph

gboml_model = GbomlGraph(24*365)
nodes, edges, _ = gboml_model.import_all_nodes_and_edges("path_to_GBOML_directory/
↪examples/remote_energy_supply_chain/remote_hub_nowacc.txt")
gboml_model.add_nodes_in_model(*nodes)
gboml_model.add_hyperedges_in_model(*edges)
gboml_model.build_model()
solution = gboml_model.solve_cplex()
print(solution)
```

The solution of this example is printed in the terminal.

9.3 Python API Example

The following example illustrates various functions defined in the Python API and its *GbomlGraph* class.

First, the *GbomlGraph* class can be imported from the *gboml* package as follows:

```
from gboml import GbomlGraph
```

Second, an instance of the *GbomlGraph* class with a time horizon of 3 can be created:

```
timehorizon = 3
gboml_model = GbomlGraph(timehorizon)
```

Third, all the nodes defined in the microgrid example can be imported:

```
nodes, edges, global_param = gboml_model.import_all_nodes_and_edges("examples/microgrid/
↪microgrid.txt")
```

Then, the nodes and hyperedges may be re-named by adding "new_" to the original names of all nodes and hyperedges,

```
old_names = []
for node in nodes:
   old_names.append(node.get_name())
   gboml_model.rename(node, "new_"+node.get_name())

for hyperedge in edges:
   gboml_model.rename(hyperedge, "new_"+hyperedge.get_name())
   for i, node in enumerate(nodes):
      gboml_model.change_node_name_in_hyperedge(hyperedge, old_names[i], node.get_name())
```

Let us assume that a node named `N` exists in a GBOML file called *test6.txt*. In addition, let us assume that a variable `x[T]`, a parameter `b=4`, a constraint `x[t]>= b` and an objective `min : x[t]` are defined in this node. Then, this node can be imported into a new node and the full microgrid problem can be encapsulated inside of it in order to create a hierarchy:

```
parent = gboml_model.import_node("test/test6.txt", "N")
for node in nodes:
   gboml_model.add_sub_node(node, parent)

for edge in edges:
   gboml_model.add_sub_hyperedge(edge, parent)
```

Note that at this stage, the *parent* node has not yet been added to the model and is currently a stand-alone imported Node object with sub-nodes and sub-hyperedges.

The value of the *parent* node parameter `b` can be updated as follows:

```
gboml_model.redefine_parameters_from_keywords(parent, b=6)
```

Finally, the *parent* node can be added to the model. The latter can then be generated and solved with CPLEX:

```
gboml_model.add_nodes_in_model(parent)
gboml_model.build_model()
solution = gboml_model.solve_cplex()
```

To recap, the full code reads:

```
from gboml import GbomlGraph

timehorizon = 3
gboml_model = GbomlGraph(timehorizon)
nodes, edges, global_param = gboml_model.import_all_nodes_and_edges("examples/microgrid/
↪microgrid.txt")
old_names = []
for node in nodes:
   old_names.append(node.get_name())
   gboml_model.rename(node, "new_"+node.get_name())

for hyperedge in edges:
   gboml_model.rename(hyperedge, "new_"+hyperedge.get_name())
   for i, node in enumerate(nodes):
      gboml_model.change_node_name_in_hyperedge(hyperedge, old_names[i], node.get_name())

parent = gboml_model.import_node("test/test6.txt", "H")
for node in nodes:
   gboml_model.add_sub_node(node, parent)

for edge in edges:
   gboml_model.add_sub_hyperedge(edge, parent)

gboml_model.redefine_parameters_from_keywords(parent, b=6)
gboml_model.add_nodes_in_model(parent)
gboml_model.build_model()
solution = gboml_model.solve_cplex()
```

9.4 Models and papers that use GBOML

- Berger et al. 2021, Remote Renewable Hubs for Carbon-Neutral Synthetic Fuel Production, https://www.frontiersin.org/articles/10.3389/fenrg.2021.671279/full. The model can be found at : https://gitlab.uliege.be/smart_grids/public/gboml/-/tree/master/examples/remote_energy_supply_chain.
- Cauz et al. 2023, Reinforcement Learning for Joint Design and Control of Battery-PV Systems, https://arxiv.org/pdf/2307.04244.pdf.
- Dachet et al. 2023, Towards CO2 valorization in a multi remote renewable energy hub framework, https://arxiv.org/pdf/2303.09454.pdf. The model can be found at : https://gitlab.uliege.be/smart_grids/public/gboml/-/tree/master/examples/towards_co2_valorization.
- Fonder et al. 2023, Synthetic methane for closing the carbon loop: Comparative study of three carbon sources for remote carbon-neutral fuel synthetization, https://arxiv.org/pdf/2310.01964.pdf. The model can be found at : https://gitlab.uliege.be/smart_grids/public/gboml/-/tree/master/examples/synthetic_methane_morocco/gboml_models.

If you have a model that you would like to add to the list. Please contact us on gitlab: https://gitlab.uliege.be/smart_grids/public/gboml.

CHAPTER

TEN

CITING GBOML

An early version of the GBOML framework was introduced in a paper: https://www.frontiersin.org/articles/10.3389/fenrg.2021.671279/full. GBOML software has been published in the Journal of Open-Source Software : https://joss.theoj.org/papers/10.21105/joss.04158. The inner workings of GBOML and a benchmark have been discussed in a recent paper published in Optimization Methods and Software : https://www.tandfonline.com/doi/full/10.1080/10556788.2023.2246169.

To cite the software :

```
@article{Miftari2023,
    author = {Bardhyl Miftari, Mathias Berger, Guillaume Derval, Quentin Louveaux and␣
↪Damien Ernst},
    title = {GBOML: a structure-exploiting optimization modelling language in Python},
    journal = {Optimization Methods and Software},
    volume = {0},
    number = {0},
    pages = {1-30},
    year = {2023},
    publisher = {Taylor & Francis},
    doi = {10.1080/10556788.2023.2246169},
    URL = {https://doi.org/10.1080/10556788.2023.2246169},
    eprint = {https://doi.org/10.1080/10556788.2023.2246169}
}
```

```
@article{Miftari2022,
  doi = {10.21105/joss.04158},
  url = {https://doi.org/10.21105/joss.04158},
  year = {2022},
  publisher = {The Open Journal},
  volume = {7},
  number = {72},
  pages = {4158},
  author = {Bardhyl Miftari and Mathias Berger and Hatim Djelassi and Damien Ernst},
  title = {GBOML: Graph-Based Optimization Modeling Language},
  journal = {Journal of Open Source Software}
}
```

A PDF version of this documentation can be found here.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

add_global_parameter() (*gboml.GbomlGraph method*), 31
add_global_parameters() (*gboml.GbomlGraph method*), 32
add_global_parameters_objects() (*gboml.GbomlGraph method*), 32
add_hyperedges_in_model() (*gboml.GbomlGraph method*), 32
add_nodes_in_model() (*gboml.GbomlGraph method*), 32
add_sub_hyperedge() (*gboml.GbomlGraph static method*), 32
add_sub_node() (*gboml.GbomlGraph static method*), 32

B

build_model() (*gboml.GbomlGraph method*), 32

C

change_node_name_in_hyperedge() (*gboml.GbomlGraph static method*), 32
change_type_variable_in_node() (*gboml.GbomlGraph static method*), 33
create_parameter() (*gboml.GbomlGraph static method*), 33

G

GbomlGraph (*class in gboml*), 31
get_object_in_node() (*gboml.GbomlGraph static method*), 33
get_timehorizon() (*gboml.GbomlGraph method*), 33

I

import_all_nodes_and_edges() (*gboml.GbomlGraph static method*), 33
import_hyperedge() (*gboml.GbomlGraph static method*), 34
import_node() (*gboml.GbomlGraph static method*), 34

M

modify_parameter_value() (*gboml.GbomlGraph static method*), 34

R

redefine_parameters_from_keywords() (*gboml.GbomlGraph static method*), 34
redefine_parameters_from_list() (*gboml.GbomlGraph static method*), 35
remove_constraint() (*gboml.GbomlGraph static method*), 35
remove_objective_in_node() (*gboml.GbomlGraph static method*), 35
rename() (*gboml.GbomlGraph static method*), 35

S

set_parsing_cache_limit() (*gboml.GbomlGraph static method*), 35
set_timehorizon() (*gboml.GbomlGraph method*), 35
solve_cbc() (*gboml.GbomlGraph method*), 35
solve_clp() (*gboml.GbomlGraph method*), 36
solve_cplex() (*gboml.GbomlGraph method*), 36
solve_dsp() (*gboml.GbomlGraph method*), 36
solve_gurobi() (*gboml.GbomlGraph method*), 37
solve_highs() (*gboml.GbomlGraph method*), 37
solve_xpress() (*gboml.GbomlGraph method*), 37

T

turn_solution_to_dictionary() (*gboml.GbomlGraph method*), 37
turn_solution_to_list() (*gboml.GbomlGraph method*), 38