The Journal of Open Source Software

GBOML: Graph-Based Optimization Modeling Language

Bardhyl Miftari[1*¶], **Mathias Berger**[1*¶], **Hatim Djelassi**[1], **and Damien Ernst**[1,2]

1 Department of Electrical Engineering and Computer Science, University of Liège **2** LTCI, Telecom Paris, Institut Polytechnique de Paris ¶ Corresponding author * These authors contributed equally.

DOI: 10.21105/joss.04158

Software

- Review
- Repository
- Archive

Editor: Frauke Wiese

Reviewers:

- @odow
- @leonardgoeke

Submitted: 28 January 2022
Published: 22 April 2022

Summary

The Graph-Based Optimization Modeling Language (GBOML) is a modeling language for mathematical programming enabling 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. It also directly interfaces with a variety of open-source and commercial solvers, including structure-exploiting ones.

Statement of need

Many planning and control problems (e.g., in the field of energy systems) can be formulated as mathematical programs. Notably, many models of practical interest can be viewed as collections of smaller models with some form of coupling between them. Such models display a natural block structure, where each block represents a small model or component.

Broadly speaking, two classes of tools can be used to implement such models, namely algebraic modeling languages (AMLs) and so-called object-oriented modeling environments (OOMEs). In short, AMLs typically allow users to compactly encode broad classes of optimization problems (e.g., mixed-integer nonlinear programs) while decoupling models and solvers. AMLs can be either stand-alone (e.g., GAMS (Bussieck & Meeraus, 2004) or AMPL (Fourer et al., 1990)) or embedded in general-purpose programming languages, such as JuMP (Dunning et al., 2017) (in Julia), Pyomo (Hart et al., 2011) and PuLP (Mitchell et al., 2011) (in Python). They may also be open-source (e.g., JuMP, Pyomo and PuLP). On the other hand, OOMEs usually make it easy to construct instances of specific problems from pre-defined components. For instance, in the field of energy systems, established OOMEs include PyPSA (power flow calculations and capacity expansion planning) (Brown et al., 2018), Calliope (multi-energy carrier capacity expansion planning) (Pfenninger & Pickering, 2018) or Balmorel (long-term planning and short-term operational analyses) (Wiese et al., 2018).

Unfortunately, both AMLs and OOMEs suffer from several drawbacks. More precisely, AMLs rarely provide language constructs to represent the block structure in a model and typically fail to take advantage of it, although this may be used to simplify problem encoding (e.g., by treating blocks as independent objects), enable model re-use (e.g., by allowing the import of model components), speed up model generation (e.g., by parallelising block generation)

and facilitate the use of structure-exploiting solution algorithms. Extensions to established AMLs were proposed to address some of these concerns, such as StructJuMP (Huchette & Developers, 2021), PySP (Watson et al., 2012) and SML (Colombo et al., 2009) (extending JuMP, Pyomo and AMPL, respectively). However, these extensions were usually designed with the primary aim of handling very specific problem structures (e.g., dual block angular structures found in stochastic linear programming models). On the other hand, OOMEs typically lack expressiveness and adding components is often cumbersome. Furthermore, they very often rely on established AMLs themselves and thus automatically inherit any shortcomings of the AML on top of which they are built (e.g., it may not be open-source).

The tools that appear closest in spirit to our own are the recent SMS++ modeling framework (Frangioni et al., 2021) and Plasmo.jl (Jalving et al., 2020) (an extension of JuMP). The former makes it possible to implement fairly general block-structured models and aims to ease the development and deployment of advanced structure-exploiting algorithms. The latter relies on a graph abstraction of optimization models to enable modular model construction, partitioning and the use of structure-exploiting algorithms.

Against this backdrop, GBOML was designed to blend and natively support some key features of AMLs and OOMEs. Notably, it was built with the following goals in mind:

- being open-source and stand-alone
- allowing any mixed-integer linear program to be represented
- enabling any hierarchical block structure to be exposed and exploited
- facilitating the encoding and construction of time-indexed models
- allowing low-level model encoding to be close to mathematical notation
- making it easy to re-use and combine components and models
- interfacing with commercial and open-source solvers, including structure-exploiting ones

An early version of the tool was used in a research article studying the economics of carbon-neutral fuel production in remote areas where renewable resources are abundant (Berger et al., 2021). The tool is also used in the context of a research project focusing on the design of the future Belgian energy system.

Example

Next, we describe a short example illustrating how GBOML works. First, a model must be encoded in a GBOML input file. An input file implementing a stylised microgrid investment planning problem is shown below.

```
#TIMEHORIZON
T = 8760; // planning horizon (hours)

#NODE SOLAR_PV
#PARAMETERS
capex = 600; // annualised capital expenditure per unit capacity
capacity_factor = import "pv_gen.csv"; // normalised generation profile
#VARIABLES
internal: capacity; // capacity of solar PV plant
external: power[T]; // power output of solar PV plant
#CONSTRAINTS
capacity >= 0;
power[t] >= 0;
power[t] <= capacity_factor[t] * capacity;
#OBJECTIVES
min: capex * capacity;

#NODE BATTERY
```

```
#PARAMETERS
capex = 150; // annualised capital expenditure per unit capacity
#VARIABLES
internal: capacity; // energy capacity of battery storage system
internal: energy[T]; // energy stored in battery storage system
external: power[T]; // power flow in/out of battery storage system
#CONSTRAINTS
capacity >= 0;
energy[t] >= 0;
energy[t] <= capacity;
energy[t+1] == energy[t] + power[t];
#OBJECTIVES
min: capex * capacity;

#HYPEREDGE POWER_BALANCE
#PARAMETERS
electrical_load = import "electrical_load.csv";
#CONSTRAINTS
SOLAR_PV.power[t] == electrical_load[t] + BATTERY.power[t];
```

The optimization horizon is defined at the beginning of the GBOML input file. Two node blocks then define the solar photovoltaic (PV) and battery storage system models, respectively. Each node has its own parameters, variables, constraints and objectives. Finally, a hyperedge block is used to ensure that power flows in the microgrid are balanced and the electricity demand is satisfied.

Then, the command-line interface or the Python API may be used to generate the model and solve it. Model generation can be parallelised based on the block structure provided by the user and models can be passed to open-source or commercial solvers. Direct access to several solver APIs is provided, allowing users to tune algorithm parameters and query complementary information (e.g., dual variables, slacks or basis ranges, when available).

Finally, results are retrieved and can be either printed to file or used directly in Python. Two file formats are supported at the time of writing, namely CSV and JSON.

The full GBOML syntax is described in the online documentation, along with advanced features such as model imports and hierarchical model construction as well as solver options. The files required to run the example above are available in the GBOML repository.

Acknowledgements

The authors gratefully acknowledge the support of the Federal Government of Belgium through its Energy Transition Fund and the INTEGRATION project. The authors would also like to thank Adrien Bolland for his help with an early version of the documentation, Jocelyn Mbenoun for testing some early features of the tool, as well as Ghislain Detienne and Thierry Deschuyteneer for constructive discussions.

References

Berger, M., Radu, D., Detienne, G., Deschuyteneer, T., Richel, A., & Ernst, D. (2021). Remote Renewable Hubs for Carbon-Neutral Synthetic Fuel Production. *Frontiers in Energy Research*, *9*, 200. https://doi.org/10.3389/fenrg.2021.671279

Brown, T., Horsch, J., & Schlachtberger, D. (2018). PyPSA: Python for Power System Analysis. *Journal of Open Research Software*, *6*(3). https://doi.org/10.5334/jors.188

Bussieck, M. R., & Meeraus, A. (2004). General Algebraic Modeling System (GAMS). In J. Kallrath (Ed.), *Modeling Languages in Mathematical Optimization* (Vol. 88, pp. 137–157). Springer. https://doi.org/10.1007/978-1-4613-0215-5_8

Colombo, M., Grothey, A., Hogg, J., Woodsend, K., & Gondzio, J. (2009). A Structure-Conveying Modelling Language for Mathematical and Stochastic Programming. *Mathematical Programming Computation*, *1*(4), 223–247. https://doi.org/10.1007/s12532-009-0008-2

Dunning, I., Huchette, J., & Lubin, M. (2017). JuMP: A Modeling Language for Mathematical Optimization. *SIAM Review*, *59*(2), 295–320. https://doi.org/10.1137/15M1020575

Fourer, R., Gay, D. M., & Kernighan, B. W. (1990). A Modeling Language for Mathematical Programming. *Management Science*, *36*(5), 519–554. https://doi.org/10.1287/mnsc.36.5.519

Frangioni, A., Iardella, N., & Lobato, R. D. (2021). The SMS++ Project: A Structured Modelling System for Mathematical Models. In *Gitlab Repository*. Gitlab. https://smspp.gitlab.io/

Hart, W. E., Watson, J.-P., & Woodruff, D. L. (2011). Pyomo: Modeling and Solving Mathematical Programs in Python. *Mathematical Programming Computation*, *3*(3), 219–260. https://doi.org/10.1007/s12532-011-0026-8

Huchette, J., & Developers, S. (2021). StructJuMP: A Block-Structured Optimization Framework for JuMP. In *GitHub Repository*. GitHub. https://github.com/StructJuMP/StructJuMP.jl

Jalving, J., Shin, S., & Zavala, V. M. (2020). *A Graph-Based Modeling Abstraction for Optimization: Concepts and Implementation in Plasmo.jl*. https://arxiv.org/abs/2006.05378

Mitchell, S., O'Sullivan, M., & Dunning, I. (2011). *PuLP: A Linear Programming Toolkit for Python*. http://www.optimization-online.org/DB_FILE/2011/09/3178.pdf

Pfenninger, S., & Pickering, B. (2018). Calliope: A Multi-Scale Energy Systems Modelling Framework. *Journal of Open Source Software*, *3*(29), 825. https://doi.org/10.21105/joss.00825

Watson, J.-P., Woodruff, D. L., & Hart, W. E. (2012). PySP: Modeling and Solving Stochastic Programs in Python. *Mathematical Programming Computation*, *4*(2), 109–149. https://doi.org/10.1007/s12532-012-0036-1

Wiese, F., Bramstoft, R., Koduvere, H., Pizarro Alonso, A., Balyk, O., Kirkerud, J. G., Tveten, Å. G., Bolkesjø, T. F., Münster, M., & Ravn, H. (2018). Balmorel Open Source Energy System Model. *Energy Strategy Reviews*, *20*, 26–34. https://doi.org/10.1016/j.esr.2018.01.003