Vilnius University
Institute of Data Science and Digital Technologies
L I T H U A N I A

INFORMATICS (N009)

RESEARCH AND DEVELOPMENT OF AN OPEN SOURCE GLOBAL OPTIMIZATION SYSTEM

Vaidas Jusevičius

October 2019

Technical Report DMSTI-DS-N009-19-09

VU Institute of Data Science and Digital Technologies, Akademijos str. 4, Vilnius LT-08412, Lithuania
www.mii.lt

Abstract

We consider a research and development of an open source global optimization system. In this work, we investigate the current state of optimization modeling systems by selecting four of the most prominent algebraic modeling languages (`AMPL`, `AIMMS`, `GAMS`, `Pyomo`) and modeling systems supporting them in order to perform an extensive theoretical and experimental analysis of their characteristics. In our theoretical comparison, we evaluate how the features of reviewed languages match with requirements for modern AMLs, while in the experimental analysis we create automated tools to generate a test model library and tools to perform extensive benchmarks using the library created. We determine the best performing AMLs by comparing the time needed to create model instance for specific type of optimization problem and analyze the impact that the presolve procedures performed by AML have on the actual problem solving. Lastly, we provide insights on which AMLs performed best and features that we deem important in the current landscape of mathematical optimization.

Keywords: Algebraic modeling languages, optimization, AMPL, AIMMS, GAMS, Pyomo

Contents

1 Introduction

Many real-world problems are routinely solved using modern optimization tools [FG02]. Internally, these tools use the combination of a mathematical model with an appropriate solution algorithm to solve the problem at hand. Thus, the way mathematical models are formulated is critical to the impact of optimization in real life.

Mathematical modeling is the process of translating real-world problems into mathematical formulations whose theoretical and numerical analysis can provide insight, answers, and guidance beneficial for the originating application [Kal04]. Algebraic modeling languages (AMLs) are declarative optimization modeling languages, which bridge the gap between model formulation and the proper solution technique [FG02]. They enable the formulation of a mathematical model as a human-readable set of equations. Typically an AML does not specify how the described model is solved.

The similarity of the model written in an algebraic modeling language to the mathematical formulation of a problem is an essential aspect which distinguishes algebraic modeling languages from other types of modeling languages, like object-oriented (e.g., `OptimJ`), solver specific (e.g., `LINGO`), general purpose (e.g., `TOMLAB`) modeling languages. This algebraic design approach allows practitioners without specific programming or modeling knowledge to be efficient in describing the problems to be solved.

It is also important to note that the algebraic modeling language is then responsible for creating a problem instance that a solution algorithm can tackle [Kal04]. Since the majority of algebraic modeling languages are integral part of a specific modeling system, it is important to isolate the responsibilities of a modeling language from an overall modeling system.

In general, AMLs are sophisticated software packages that provide a key link between a mathematical concept of an optimization model and the complex algorithmic routines that compute optimal solutions. AML software automatically reads a model and data, generates an instance, and conveys it to a solver in the required form [Fou13].

From the late 1970s many AMLs were created (e.g. `GAMS` [MMvdE+16], `AMPL` [Fou03]) and are still actively developed and used today. Lately new open-source competitors to the traditional AMLs started to emerge (e.g. `Pyomo` [HLW+17, HWW11], `JuMP` [DHL17, LD15]). Therefore we feel that a review and comparison of the traditional and emerging AMLs is needed to examine what newcomers are bringing to the competition.

The remainder of the paper is organized as follows. In Section 2 we review basic characteristics of algebraic modeling languages and motivate our selection of AMLs for the current review. In Section 3 we investigate how the requirements for a modern AML are met within each of the chosen languages. In Section 4 we examine characteristics of AMLs using an extensive benchmark. Finally, we conclude the paper in Section 5.

2 Algebraic modeling languages

In the late 1970s, when the first algebraic modeling languages were developed, they were game changers as they allowed separating model formulation from the implementation details [Kal04] while keeping notation close to the mathematical formulation of the problem [FG02]. Since the data appears to be more volatile than the problem structure, most modeling languages designers insist on data and model structure being separated [Ḧ99]. Therefore, the central idea in modern algebraic modeling languages is the differentiation between abstract models and concrete problem instances [HWW11]. A specific model instance is generated from an abstract model using data. This way, model and data together specify a particular instance of an optimization problem for which a solution can be sought. This is realized by replicating every entity of an abstract model over the different elements of the data set, and often is referred to as a set-indexing ability of the AML [FG02].

Essential characteristics of a modern AML could be defined in the following way [Kal04]:

- problems are represented in a declarative way;
- there is a clear separation between problem definition and the solution process;
- there is a clear separation between the problem structure and its data.

Support for mathematical expressions and operations needed for describing nonlinear models is considered an important feature of an AML [Kal04].

Also, it is worth to observe that most interpreters included in today's algebraic modeling languages are based on automatic differentiation [FG02], a process in which the modeling language can compute derivatives of problems from the model description without the assistance of the user [Kal04]. This motivates us to include automatic differentiation as an important feature of a modern AML too.

The algebraic expressions are useful not only in describing individual models but also for describing manipulations on models and transformations of data. Thus almost as soon as AML became available, users started finding ways to adapt model notations to implement sophisticated solution strategies and iterative schemes. These efforts stimulated the evolution within AMLs of scripting features, which include statements for looping, testing, and assignment [Fou13]. Therefore, scripting capabilities is another aspect which differentiates AMLs.

2.1 Reviewed AMLs

For this review, we have chosen four AMLs: `AIMMS` [BR19], `AMPL`, `GAMS`, and `Pyomo`. The selection was based on the following criteria:

- AMLs which won 2012 INFORMS Impact Prize award[1] [INF12];
- popularity of an AML based on NEOS Server[2] [NEO19] model input statistics;
- an emerging open-source alternative `Pyomo` was added to the list, since it may be attractive for situations where budgets are tight or where the greatest degree of flexibility is required – such as when new or customized algorithmic ideas are being investigated [Fou17].

3 Comparative analysis of AMLs characteristics

In the following section, we investigate how the requirements for a modern AML defined in the previous section are met by each of the chosen languages. The websites of the AMLs and vendor documentation are used for this comparison. Any support of the identified features and capabilities are validated against the documentation the suppliers of the AMLs provide. In addition, an in-depth survey concluded by Robert Fourer in Linear Programming Software Survey [Fou17] is also used as a reference.

The following AML characteristics are reviewed:

- are problems represented in a declarative way?
- does a clear separation between problem definition and the solution process exist?
- does a clear separation between the problem structure and its data exist?

Later on a more practical comparison of AML characteristics is conducted to identify potential easy of use of AML in daily work.

3.1 Criteria of the practical comparison

For the practical comparison of the selected AMLs, a classical Dantzig Transportation Problem was chosen [Dan63]. In this problem, we are given the supplies at the factories and the demands at the markets for a single commodity. We have also given the unit costs of shipping the product from factories to the markets. Then, the goal is to find the least cost shipping schedule that meets requirements at markets and supplies at factories.

The transportation problem formulated as a model in all considered AML is compared based on the following criteria:

- model size in bytes;
- model size in number of code lines;
- model size in number of language primitives used;

[1] prize awarded to the originators of the five most important algebraic modeling languages
[2] free internet-based service for solving the numerical optimization problem

- basic model instance creation time.

Since transportation problem is a linear programming (LP) type of problem we have chosen to measure model instance creation time as the time needed to export concrete model instance to `MPS` [lps19] format supported by most LP solvers.

For the first comparison, sample implementations of the transportation problem for the AMLs under consideration were provided by the following sources:

- AIMMS Wikipedia page [Wik19]
- AMPL model in GNU Linear Programming Kit [LL14]
- GAMS Model Library [GAM19b]
- Pyomo Gallery [Pyo19]

Transportation problem models in different AMLs can be seen in Appendix Nr. 2..

It should be noted that the textual representation of an `AIMMS` model presents the model as a tree of attributed identifier nodes. It reflects how the model is given to the modeler in the `AIMMS` IDE and is typically generated by the `AIMMS` IDE. Also, it is worth to note that for the sake of simplicity, problem model samples are concrete models, i.e., data of the model instance is described alongside with model structure.

3.2 Findings

In all of the reviewed algebraic modeling languages problems are represented in a declarative way. Furthermore, since all of them are part of a specific modeling system, a clear separation between problem definition and the solution process in the context of the modeling system exists. The separation between the problem structure and it's data is supported in all of the reviewed languages. It should be noted that `Pyomo` and `GAMS` also allow initiating data structures during their declaration while `AIMMS` and `AMPL` only support it as a separate step in the model instance building process. However, while it might be convenient for building a simple model, we do not consider the lack of direct data structure initiation as an advantage since, in real-world cases, it is rarely needed. Therefore, we can conclude that all of the reviewed languages fulfill the basic characteristics of a modern algebraic modeling language as defined in the previous Section 2.

It is important to note that `Pyomo` is a Python-based AML. Being built on top general purpose programming language makes it fundamentally different from the competitors. This allows researchers familiar with Python to learn, improve and use `Pyomo` much easier while it is practically impossible to introduce improvements to the commercial counterparts.

Comparison of the characteristics for the sample Transportation Problem model implemented in all the reviewed algebraic modeling languages can be seen in Table 1. To have a more concise view, the simplification of model implementations provided in the literature sources is made in the following way:

- all the optional comments, explanatory texts, and documentation are removed;
- all empty lines are excluded;
- parts of the code responsible for calling the solver and displaying results are omitted;
- while counting AML primitives generic functions (*sum*, *for*), data loading directives (*data*), arithmetical and logical operators are excluded.

Table 1: Comparison of Transportation Problem models

Criteria	**AIMMS**	**AMPL**	**GAMS**	**Pyomo**
size in bytes	2229	683	652	1207
lines of code	68	24	31	28
primitives used	9	5	8	6

As we see from Table 1 models implemented in both `AMPL` and `GAMS` are the most compact ones, while model written in `AIMMS` is much more verbose and `Pyomo` is somewhere in the middle. The reason for `AIMMS` model being much more verbose is in the nature of `AIMMS` modeling system, which propagates model creation using graphical user interface (GUI) while keeping the source code of the model hidden from a modeler. Naturally, there is not that much of the focus on how the model is stored. We can argue that while the GUI based approach might be convenient to some of the modelers, it enforces greater vendor lock-in and makes extensibility and maintainability of the model harder.

While comparing a number of language primitives required to create a model, `AMPL` and `Pyomo` showed best results which allows us to predict that these modeling languages might have a more gentle learning curve.

Therefore, we can conclude that in the context of reviewed algebraic modeling languages, `AMPL` allows formulating an optimization problem in the shortest way.

The creation time of the transportation problem model instance defined in each AMLs was used as a measure for a model loading. The process was done in the following steps:

1. loading model instance from a problem definition written in the native AML;
2. exporting model instance to MPS format;
3. measuring total execution time;
4. investigating characteristics of an instance model.

Since creators of `AIMMS` system did not respond to the request for an academic license, we were not able to include `AIMMS` into the benchmark. Generated model instances in MPS format can be found in `models` directory of our GitHub repository [JP19].

Characteristics of the created model instances can be seen in Table 2. We can conclude that all of the modeling languages have created a model instance using the same amount of variables and constraints, however, the definition of non zero elements is different between `GAMS` and other modeling systems.

Table 2: Characteristics of the created transportation model instances

Characteristic	**AMPL**	**GAMS**	**Pyomo**
Constraints	6	6	6
Non zero elements	13	19	13
Variables	7	7	7

In Table 3 model instance creation time benchmark results are provided. We have tried to run multiple consecutive model instance creations (10 runs, 100 runs) in order to identify if any caching is being used by the modeling system. We can exhibit that `AMPL` showed significantly better results compared to others. This allows concluding that `AMPL` is the most optimized from a performance point of view. On the other hand, `Pyomo` coming in last is shows the potential language still has for performance improvements. We can also conclude that no caching of previous runs took place since in none of the cases of multiple consecutive model instance creations lead to noticeable performance improvement.

Table 3: Total time of consecutive transportation model instance creation runs

AML	1 run	10 runs	100 runs
AMPL	30 ms	220 ms	2130 ms
GAMS	170 ms	1730 ms	16490 ms
Pyomo	720 ms	7280 ms	79600 ms

4 Performance benchmark of AMLs

All of the examined AMLs support the same types of optimization problems, however it is unclear how efficiently each AML is capable of handling large model loading and what optimizations are applied during model instance creation. It would also be of a great value to analyze how each of the modeling languages performs within an area of the specific type of optimization problems (linear, quadratic, nonlinear, mixed-integer, etc.). To give such a comparison and thoroughly examine characteristics of AMLs, a more extensive benchmark involving much larger optimization problem models is needed. Therefore a large and extensive library of sample optimization problems for the analyzed AMLs has to be used.

4.1 AMLs testing library

We have chosen GAMS Model Library [GAM19b] as a reference for creating such a sample optimization problem suite against which future research will be done. Automated shell script `gamslib-convert.sh` was created to build such a library. It can be found in the `tools` directory of our GitHub repository [JP19]. Detailed explanation on how the test library creation tool works and issues identified in the GAMS Library are provided in Appendix Nr. 1..

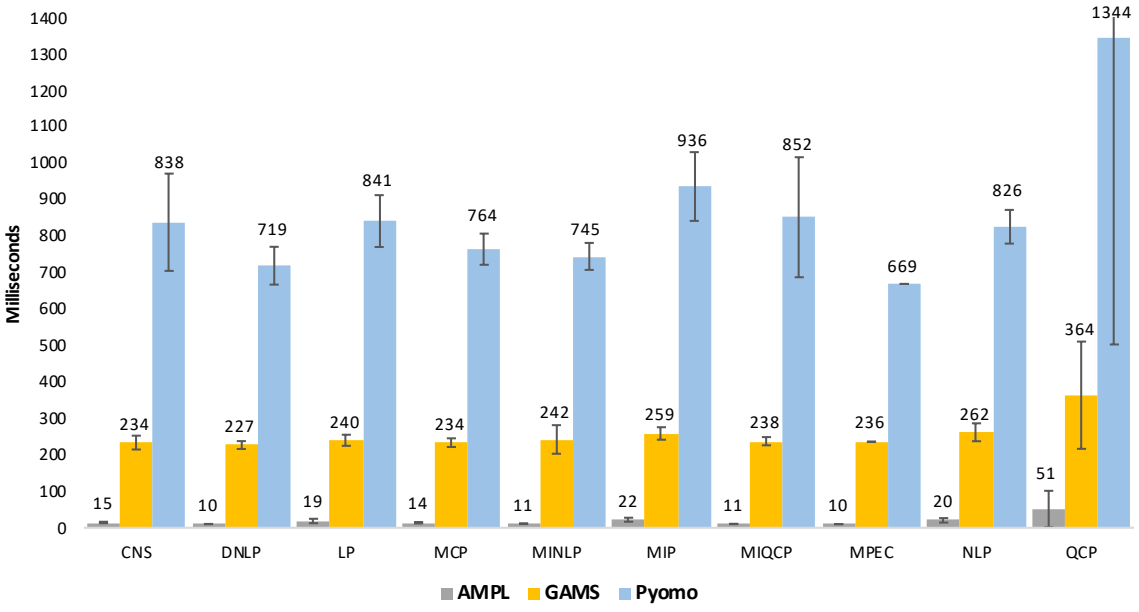
As a result of the transformation, we compiled a library consisting of 298 sample problems in `AMPL`, `GAMS`, and `Pyomo` scalar model formats.

4.2 Model instance creation time

The generated library was used to determine the amount of time each modeling system requires to create problem instance of a particular problem. For that, we decided to write `load-benchmark.sh` shell script available in `tools` directory of our GitHub repository [JP19] which loads each model into the particular modeling system and then exports it to the format understandable by the solvers. We have chosen *.nl* [Gay05] or *.nlc* formats as target formats acceptable by the solvers. The benchmark measures the time modeling system took to perform both model instance creation and export operations.

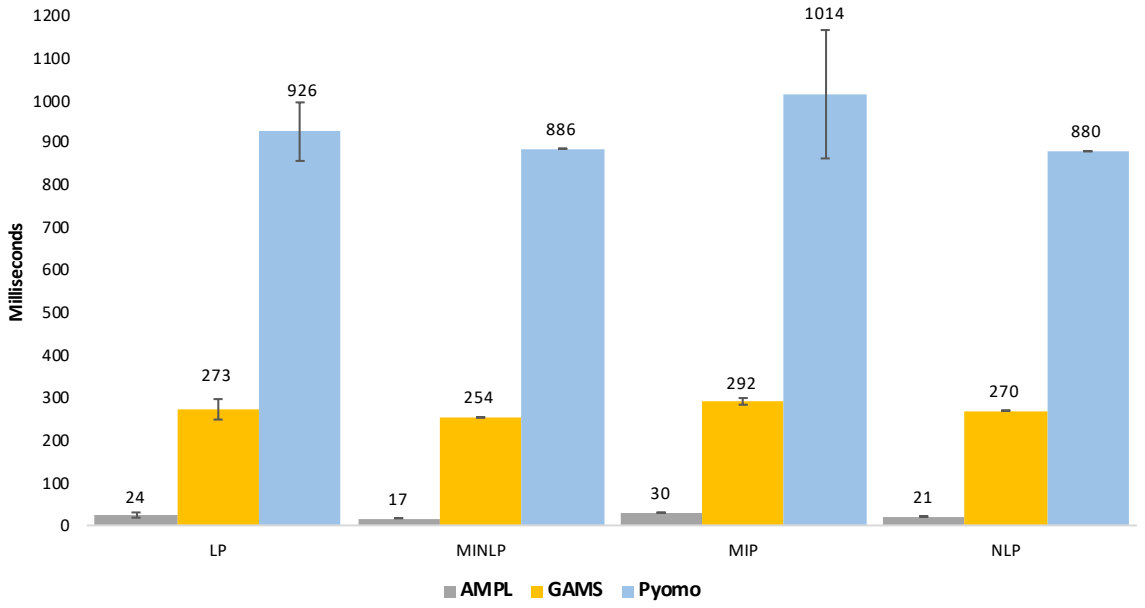
We have chosen to exclude sample problems which had conversion errors from the benchmark (more information about them in Appendix Nr. 1.), meaning only the models which were successfully processed by all modeling systems were compared. This reduced the scope of our benchmark to 269 models.

Figure 1: Average model instance creation time



Detailed benchmark results can be found in `model-loading-times.xlsx` workbook in the benchmark section of our GitHub repository [JP19] while the summary of average model instance creation time split by the problem type can be seen in Figure 1.

Figure 2: Average large model instance creation time



We can see the trend exhibited in the transportation problem model benchmark persists. `AMPL` is still a definite top performer while `Pyomo` performs the worst. There are no significant variations between different optimization problem types. However, the `Pyomo` model instance creation time tends to vary more once working with different types of problems. The average difference between `AMPL` and other contenders increases when models become larger. For example, for the transportation problem, `GAMS` took 5.5 times more to create the model instance than `AMPL`, and `Pyomo` took 24 times more.

Comparing instance creation times of large models (models having more than 500 equations, 8 such models in the testing library) reveals 11 times the difference between `AMPL` and `GAMS`, and 38 times the difference between `AMPL` and `Pyomo`. The difference between `GAMS` and `Pyomo` stayed roughly the same - around 3.5 times. Summary of large models instance creation time can be seen in Figure 2.

We can conclude that out of reviewed AMLs `AMPL` is a clear top performing AML when it comes to model instances creation time.

4.3 Presolving in AMLs

Another performance-related feature of algebraic modeling languages is the ability to presolve problem before providing it to the solver. The presolver can preprocess problems and simplify, i.e. reduce the problem size or determine the problem to be unfeasible.

Only two of the reviewed algebraic modeling languages provide presolving capabili-

Table 4: AMPL model presolving

Type	Models Count	Presolved	Presolved (%)	Not Feasible	Constraints reduced	Variables reduced
CNS	4	4	100.00%	0	14.63%	31.39%
DNLP	5	1	20.00%	0	0.00%	7.41%
LP	57	21	36.84%	0	17.81%	9.66%
MCP	19	17	89.47%	0	47.00%	8.56%
MINLP	21	13	61.90%	1	16.32%	9.30%
MIP	61	37	60.66%	0	19.06%	11.50%
MIQCP	5	3	60.00%	2	0.00%	2.38%
MPEC	1	1	100.00%	0	50.00%	0.00%
NLP	101	48	47.52%	2	9.71%	11.55%
QCP	10	6	60.00%	0	7.10%	2.55%
RMIQCP	2	0	0.00%	0	0.00%	0.00%
Total	286	151	52.80%	5	18.42%	10.73%

ties - `AMPL` [Fou03] and `AIMMS` [AIM19]. Since we did not have the opportunity to evaluate `AIMMS` modeling language practically, we were only able to examine `AMPL` presolving capabilities. In order to evaluate `AMPL` presolving performance, we gathered presolving characteristics while performing model instance creation time benchmark. We have used 286 models which were successfully converted from `GAMS` original model to `AMPL` scalar model. A detailed report of the presolving applied to the specific model can be seen in the benchmark section of our GitHub repository [JP19] while the summary of it can be found in Table 4.

We observed that `AMPL` presolver managed to simplify models in 52.8% of the cases, out of which 5 times it was able to determine that the problem solution is not feasible, thus even not requiring to call the solver. On average, once applied `AMPL` presolver managed to reduce the model size by removing 18.42% of constraints and 10.73% of variables.

We can conclude that `AMPL` presolver is an efficient way to simplify larger problems which might lead to improved solution finding performance once invoking a solver with an already reduced problem model instance. Also, the ability to determine not feasible models can help modelers in the problem definition process to debug and find errors in the model definition. This allows us to argue that presolving is an important capability of any modern algebraic modeling language.

4.4 Presolve impact on solving

In order to evaluate if `AMPL` presolving has a positive impact on problem solving an additional benchmark was conducted. The benchmark included 146 out of 151 models to which `AMPL` has applied presolve in the model instance creation benchmark. Five models which `AMPL` presolve determined to be not feasible were excluded from the benchmark. Shell script `solve-benchmark.sh` provided in tools directory of our GitHub repos-

itory [JP19] was created for executing such a benchmark. The script solves each model using one of the solvers and gathers output statistics to a report file.

We have chosen to solve models using `Gurobi` [Gur19] and BARON [Sah19] [TS05] solvers. `Gurobi Optimizer (v.8.1.0)` was chosen for solving `LP, MIP, QCP` and `MIQCP` type of problems. While `BARON (v.18.11.12)` global solver was chosen for solving `NLP, MINLP, MCP, MPEC, CNS` and `DNLP problems`. The choice of the solvers was motivated by the support for particular problem types [Gur19] [Sah19] and the popularity of solvers based on NEOS Server statistics [NEO19].

Two attempts to solve each model were made. One with `AMPL` presolver turned on (default setting) and second one with `AMPL` presolver turned off. After each run solvers statistics including iterations count, solve time (pure solve phase execution time) and objective were gathered.

It is important to note that both `BARON` and `Gurobi` solvers have their own presolve mechanisms so the provided model is simplified by the solver too. This might result in very similar models being solved by the solver in spite of the `AMPL` presolve being turned on or off. However, the focus was on estimating `AMPL` presolve impact in real life situations, so full benchmark was executed without changing default solver behaviour. Later on an additional benchmark was made to estimate what is the impact of `AMPL` presolve once solver presolve functionality is turned off.

Detailed `AMPL` presolve impact on solving report can be found in our GitHub repository's [JP19] directory benchmark file ampl-solving-times.xlsx sheet Benchmark 1. While here in Table 5 we summarize the positive and negative impact `AMPL` presolve had on solving problems iteration and time wise. Positive impact means fewer iterations or time was needed to solve a problem once the presolve was turned on. Negative impact means the opposite that more iterations or time was required.

Table 5: Summary of AMPL presolve impact on solving

	Iterations wise	Time wise	Iterations wise %	Time wise %
Positive	37	67	26.43%	47.86%
Neutral	74	40	52.86%	28.57%
Negative	29	33	20.71%	23.57%

During the benchmark 6 models failed to be solved due to solver limitations. A detailed explanation of limitations faced is provided in the benchmark report. Two models deemed to be not feasible and two were solved during `AMPL` presolve phase. Solver's were capable to solve 41 model during solver's presolve phase. And for six models mismatching objective was found with `AMPL` presolve turned on and off. Overall `AMPL` presolve had a positive impact in 26.43% of the cases iteration wise and 47.86% time wise. However it had a negative impact in 20.71% of cases iteration wise and 23.57% time wise.

As mentioned earlier both `BARON` and `Gurobi` solvers have their own presolve mechanisms. In order to test what would be an impact of `AMPL` pesolve if the solver does

not attempt to presolve a problem on its own an additional benchmark was made. Since only Gurobi allows to disable presolve functionality a subset of models previously solved with Gurobi was chosen for the benchmark. Detailed benchmark results can be seen in our GitHub repository's [JP19] directory benchmark file ampl-solving-times.xlsx sheet Benchmark 2. The summary of the benchmark is provided in Table 6 and Table 7. Gurobi was not capable to solve two MIP problems (clad and mws) in reasonable time once Gurobi presolve functionality was turned off. Those models were excluded from the benchmark.

Table 6: AMPL presolve impact with Gurobi presolve on

	Iterations wise	Time wise	Iterations wise %	Time wise %
Positive	18	39	28.57%	61.90%
Neutral	34	0	53.97%	0.00%
Negative	11	24	17.46%	38.10%

Table 7: AMPL presolve impact with Gurobi presolve off

	Iterations wise	Time wise	Iterations wise %	Time wise %
Positive	33	44	54.10%	72.13%
Neutral	10	0	16.39%	0.00%
Negative	18	17	29.51%	27.87%

AMPL presolve had a greater positive effect both iteration wise (+22.4%) and time wise (+10.2%) once Gurobi presolve was turned off. AMPL presolve also had less neutral impact once solver presolving was off, thus leading to a conclusion that during first benchmark some models were simplified to a very similar ones before actually solving them.

As we can see from the benchmarks presolving done by AML has inconclusive effects on the actual problem solving both iterations and time wise. However, positive impact is always greater than the negative one and it especially becomes evident once solver does not have or use it's own problem presolving mechanisms. This allows us to conclude that presolving capabilities of AML is an important feature of a modern algebraic modeling language. We can also advice to choose AML having presolving capabilities in cases the solver used to solve the problem does not have it's own presolving mechanism.

5 Conclusions and future work

From the research, we can conclude that AMPL allows to formulate an optimization problem in the shortest and potentially easiest way while also providing the best performance in model instance loading times. It also leverages the power of model presolving, which is helpful for the modelers in both problem definition and efficient solution finding processes. GAMS is a strong runner up providing a very similar to AMPL problem formula-

tion capabilities though running behind in the model instance creation time. `AIMMS` can be considered as being on its own class of modeling languages as it has taken a purely graphical user interface based approach. Since we were not able to examine the performance characteristics of `AIMMS` due to lack of academic license, the performance aspect remains unclear. Open source alternative `Pyomo` is on par with commercial competitors in problem definition process, however Pyomo's performance of model instance creation is a bit behind once compared to the competitors.

In the future we would like to continue our research by expanding test models library with more and larger models, re-running performance and presolve benchmarks and introducing new AMLs like emerging JuMP [DHL17] or ones we had no means to fully review like AIMMS [BR19].

References

[AIM19] AIMMS B.V. The AIMMS Presolver, 2019.

[BR19] Johannes Bisschop and Marcel Roelofs. AIMMS-The User's Guide, 2019.

[Dan63] George B Dantzig. The Classical Transportation Problem. In *Linear Programming and Extensions*, pages 299–315. Princeton University Press, 1963.

[DHL17] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[FG02] Emmanuel Fragniere and Jacek Gondzio. Optimization modeling languages. *Handbook of Applied Optimization*, pages 993–1007, 2002.

[Fou03] Robert Fourer. *AMPL : a modeling language for mathematical programming*. Thomson/Brooks/Cole, Pacific Grove, CA, 2003.

[Fou13] Robert Fourer. Algebraic modeling languages for optimization. In *Encyclopedia of Operations Research and Management Science*, pages 43–51. Springer, 2013.

[Fou17] Robert Fourer. Linear Programming: Software Survey. *OR/MS Today*, 44(3), June 2017.

[GAM19a] GAMS Development Corporation. GAMS Convert, 2019.

[GAM19b] GAMS Development Corporation. GAMS Model Library, 2019.

[Gay05] David M Gay. Writing .nl files. *Optimization and Uncertainty Estimation*, 2005.

[Gur19] Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2019.

[HÖ9] Tony Hürlimann. *Mathematical Modeling and Optimization*, volume 31 of *Applied Optimization*. Springer US, Boston, MA, 1999.

[HLW+17] William E. Hart, Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, second edition, 2017.

[HWW11] William E Hart, Jean-Paul Watson, and David L Woodruff. Pyomo: modeling and solving mathematical programs in python. *Mathematical Programming Computation*, 3(3):219–260, 2011.

[INF12] INFORMS. INFORMS Impact Prize 2012, 2012.

[JP19] Vaidas Jusevičius and Remigijus Paulavičius. Algebraic modeling language benchmark, 2019.

[Kal04] Josef Kallrath. *Modeling languages in mathematical optimization*, volume 88. Springer Science & Business Media, 2004.

[LD15] Miles Lubin and Iain Dunning. Computing in operations research using julia. *INFORMS Journal on Computing*, 27(2):238–248, 2015.

[LL14] Lopaka Lee and Louis Luangkesorn. GNU Linear Programming Kit, 2014.

[lps19] lpsolve developers. MPS file format, 2019.

[MMvdE+16] Bruce A McCarl, Alex Meeraus, Paul van der Eijk, Michael Bussieck, Steven Dirkse, and Franz Nelissen. *McCarl Expanded GAMS user guide*. Citeseer, 2016.

[NEO19] NEOS Server. Neos Solver Access Statistics, 2019.

[Pyo19] Pyomo. Pyomo Gallery, 2019.

[Sah19] N. V. Sahinidis. BARON 19.7.13: Global Optimization of Mixed-Integer Nonlinear Programs, *User's Manual*, 2019.

[TS05] M. Tawarmalani and N. V. Sahinidis. A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming*, 103:225–249, 2005.

[Wik19] Wikipedia contributors. AIMMS — Wikipedia, The Free Encyclopedia, 2019.

Appendixes

Appendix Nr. 1.
Creation of AMLs testing library

The automated shell script `gamslib-convert.sh` availble in the `tools` directory of our GitHub repository [JP19] was created to generate AMLs testing library. The script uses `GAMS Convert tool v.25` [GAM19a] to convert model in `GAMS` proprietary format to a scalar model in the `AMPL`, `GAMS` and `Pyomo` formats. Characteristics of a sample problem models (number of equations, variables, discrete variables, non-zero elements, non-zero nonlinear elements) are automatically extracted and noted. Sample problems are also grouped based on optimization problem types.

The script has two execution modes - one for converting single model and another for converting all the models in GAMS Library. An example of how transportation problem available in GAMS Library [GAM19b] looks converted to `GAMS` scalar format can be seen in Listing 1.

Listing 1 Transportation problem converted to GAMS scalar model

```
Variables  x1,x2,x3,x4,x5,x6,x7;
Positive Variables  x1,x2,x3,x4,x5,x6;
Equations  e1,e2,e3,e4,e5,e6;
e1..  - 0.225*x1 - 0.153*x2 - 0.162*x3 - 0.225*x4 - 0.162*x5 - 0.126*x6 + x7
     =E= 0;
e2..    x1 + x2 + x3 =L= 350;
e3..    x4 + x5 + x6 =L= 600;
e4..    x1 + x4 =G= 325;
e5..    x2 + x5 =G= 300;
e6..    x3 + x6 =G= 275;
Model m / all /;
m.limrow=0; m.limcol=0;
Solve m using LP minimizing x7;
```

At the time of writing, there were 423 models in the GAMS Model Library. Out of them, we eliminated 66 models which are using `GAMS` proprietary modeling techniques (e.g., `MPSGE`, `BCH Facility`), 20 using general-purpose programming languages features (e.g. cycles), four models tightly coupled to `CPLEX` and `DECIS` solvers.

We feel it is important to note that 35 models failed to be loaded by a fully licensed `GAMS Convert` tool due to execution or compilation errors. Thus meaning some models in the GAMS Library are not compatible with `GAMS` modeling system itself.

While performing the model instance creation benchmark, we have identified that 12 `AMPL` models and 29 `Pyomo` models generated by `GAMS Convert` tool had errors in them.

Listing 2 Example of a GAMS Convert error

```
# GAMS Convert generated Pyomo suffix syntax
suffix ref integer IN;
# Correct Pyomo suffix syntax
ref = Suffix(direction=Suffix.EXPORT, datatype=Suffix.INT)
```

Most of the `Pyomo` errors were caused by an incorrect `GAMS Convert` tool behavior where the definition

of the `Suffix` primitive uses `AMPL` but not `Pyomo` semantics. Example of what `GAMS Convert` generates and the correct `Pyomo` syntax can be seen in Listing 2.

Appendix Nr. 2.
Transportation problem models

Listing 3 Transportation problem defined in AMPL format

```
set I;

set J;

param a{i in I};

param b{j in J};

param d{i in I, j in J};

param f;

param c{i in I, j in J} := f * d[i,j] / 1000;

var x{i in I, j in J} >= 0;

minimize cost: sum{i in I, j in J} c[i,j] * x[i,j];

s.t. supply{i in I}: sum{j in J} x[i,j] <= a[i];

s.t. demand{j in J}: sum{i in I} x[i,j] >= b[j];

data;

set I := Seattle San-Diego;

set J := New-York Chicago Topeka;
param a := Seattle     350
           San-Diego   600;

param b := New-York    325
           Chicago     300
           Topeka      275;

param d :              New-York   Chicago   Topeka :=
           Seattle     2.5        1.7       1.8
           San-Diego   2.5        1.8       1.4  ;

param f := 90;

end;
```

Listing 4 Transportation problem defined in GAMS format

```
Set
   i 'canning plants' / seattle,  san-diego /
   j 'markets'        / new-york, chicago, topeka /;

Parameter
   a(i) 'capacity of plant i in cases'
        / seattle    350
          san-diego  600 /

   b(j) 'demand at market j in cases'
        / new-york   325
          chicago    300
          topeka     275 /;

Table d(i,j) 'distance in thousands of miles'
              new-york  chicago  topeka
   seattle         2.5      1.7     1.8
   san-diego       2.5      1.8     1.4;

Scalar f 'freight in dollars per case per thousand miles' / 90 /;

Parameter c(i,j) 'transport cost in thousands of dollars per case';
c(i,j) = f*d(i,j)/1000;

Variable
   x(i,j) 'shipment quantities in cases'
   z      'total transportation costs in thousands of dollars';

Positive Variable x;

Equation
   cost      'define objective function'
   supply(i) 'observe supply limit at plant i'
   demand(j) 'satisfy demand at market j';

cost..      z =e= sum((i,j), c(i,j)*x(i,j));

supply(i).. sum(j, x(i,j)) =l= a(i);

demand(j).. sum(i, x(i,j)) =g= b(j);

Model transport / all /;

solve transport using lp minimizing z;
```

Listing 5 Transportation problem defined in Pyomo format

```
from pyomo.environ import *

model = ConcreteModel()

model.i = Set(initialize=['seattle','san-diego'])
model.j = Set(initialize=['new-york','chicago', 'topeka'])

model.a = Param(model.i, initialize={'seattle':350,'san-diego':600})
model.b = Param(model.j, initialize={'new-york':325,'chicago':300,'topeka':275})

dtab = {
    ('seattle',  'new-york') : 2.5,
    ('seattle',  'chicago')  : 1.7,
    ('seattle',  'topeka')   : 1.8,
    ('san-diego','new-york') : 2.5,
    ('san-diego','chicago')  : 1.8,
    ('san-diego','topeka')   : 1.4,
    }
model.d = Param(model.i, model.j, initialize=dtab)
model.f = Param(initialize=90)
def c_init(model, i, j):
  return model.f * model.d[i,j] / 1000
model.c = Param(model.i, model.j, initialize=c_init)

model.x = Var(model.i, model.j, bounds=(0.0,None))

def supply_rule(model, i):
  return sum(model.x[i,j] for j in model.j) <= model.a[i]
model.supply = Constraint(model.i, rule=supply_rule)
def demand_rule(model, j):
  return sum(model.x[i,j] for i in model.i) >= model.b[j]
model.demand = Constraint(model.j, rule=demand_rule)

def objective_rule(model):
  return sum(model.c[i,j]*model.x[i,j] for i in model.i for j in model.j)
model.objective = Objective(rule=objective_rule, sense=minimize)
```