Experimental Analysis of Algebraic Modelling Languages for Mathematical Optimization

Vaidas JUSEVIČIUS¹, Richard OBERDIECK², Remigijus PAULAVIČIUS¹,*

¹ Vilnius University Institute of Data Science and Digital Technologies, Akademijos 4, LT-08663 Vilnius, Lithuania
² Gurobi Optimization, LLC, USA
e-mail: vaidas.jusevicius@mif.vu.lt, oberdieck@gurobi.com, remigijus.paulavicius@mif.vu.lt

Received: October 2020; accepted: March 2021

Abstract. In this work, we perform an extensive theoretical and experimental analysis of the characteristics of five of the most prominent algebraic modelling languages (AMPL, AIMMS, GAMS, JuMP, and Pyomo) and modelling systems supporting them. In our theoretical comparison, we evaluate how the reviewed modern algebraic modelling languages match the current requirements. In the experimental analysis, we use a purpose-built test model library to perform extensive benchmarks. We provide insights on which algebraic modelling languages performed the best and the features that we deem essential in the current mathematical optimization landscape. Finally, we highlight possible future research directions for this work.

Key words: algebraic modelling languages, optimization, AMPL, AIMMS, GAMS, JuMP, Pyomo.

1. Introduction

Many real-world problems are routinely solved using modern optimization tools (e.g. Abhishek et al., 2010; Fragniere and Gondzio, 2002; Groër et al., 2011; Paulavičius and Žilinskas, 2014; Paulavičius et al., 2020a, 2020b; Pistikopoulos et al., 2015). Internally, these tools use the combination of a mathematical model with an appropriate solution algorithm (e.g. Cosma et al., 2020; Fernández et al., 2020; Gómez et al., 2019; Lee et al., 2019; Paulavičius and Žilinskas, 2014; Paulavičius et al., 2014; Paulavičius and Adjiman, 2020; Stripinis et al., 2019, 2021) to solve the problem at hand. Thus, the way mathematical models are formulated is critical to the impact of optimization in real life.

Mathematical modelling is the process of translating real-world business problems into mathematical formulations whose theoretical and numerical analysis can provide insight, answers, and guidance beneficial for the originating application (Kallrath, 2004), including the current Covid-19 pandemic (Rothberg, 2020). Algebraic modelling languages (AMLs) are declarative optimization modelling languages, which bridge the gap between
model formulation and the proper solution technique (Fragniere and Gondzio, 2002). They enable the formulation of a mathematical model as a human-readable set of equations while not requiring to specify how the described model should be solved or what specific solver should be used.

Models written in an AML are known for the high degree of similarity to the mathematical formulation. This aspect distinguishes AMLs from other types of modelling languages, like object-oriented (e.g. OptimJ), solver specific (e.g. LINGO), or general-purpose (e.g. TOMLAB) modelling languages. Such an algebraic design approach allows practitioners without specific programming or modelling knowledge to be efficient in describing the problems to be solved. It is also important to note that AML is then responsible for creating a problem instance that a solution algorithm can tackle (Kallrath, 2004). Since many AMLs are integral parts of a specific modelling system, it is essential to isolate a modelling language’s responsibilities from the overall system. In general, AMLs are sophisticated software packages that provide a crucial link between an optimization model’s mathematical concept and the complex algorithmic routines that compute optimal solutions. Typically, AML software automatically reads a model and data, generates an instance, and conveys it to a solver in the required form (Fourer, 2013).

From the late 1970s, many AMLs were created (e.g., GAMS, McCarl et al., 2016, AMPL, Fourer, 2003) and are still actively developed and used today. Lately, new open-source competitors to the traditional AMLs started to emerge (e.g., Pyomo, Hart et al., 2017, 2011, JuMP, Dunning et al., 2017; Lubin and Dunning, 2015). Therefore, we feel that a review and comparison of the traditional and emerging AMLs are needed to examine how the current landscape of AMLs looks.

The remainder of the paper is organized as follows. In Section 2, we review the essential characteristics of AMLs 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 the characteristics of AMLs using an extensive benchmark. In Section 5, we investigate the presolve impact on solving. Finally, we conclude the paper in Section 6.

2. Algebraic Modelling Languages

The first algebraic modelling languages, developed in the late 1970s, were game-changers. They allowed separating the model formulation from the implementation details (Kallrath, 2004) while keeping the notation close to the problem’s mathematical formulation (Fragniere and Gondzio, 2002). Since the data appears to be more volatile than the problem structure, most modelling language designers insist on the data and model structure being separated (Hürlimann, 1999). Therefore, the central idea in modern AMLs is the differentiation between abstract models and concrete problem instances (Hart et al., 2011). A specific model instance is generated from an abstract model using data. This way, the 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. Such a feature often is referred to as a set-indexing ability of the AML (Fragniere and Gondzio, 2002).

Essential characteristics of a modern AML could be defined in the following way (Kallrath, 2004):

1. problems are represented in a declarative1 way;
2. there is a clear separation between problem definition and the solution process;
3. there is a clear separation between the problem structure and its data.

Besides, the support for mathematical expressions and operations needed for describing non-linear models is often considered an important feature of an AML (Kallrath, 2004). Moreover, it is worth observing that most interpreters included in today’s AMLs are based on automatic differentiation (Fragniere and Gondzio, 2002), a process in which the modeling language can compute the derivatives of problems from the model description without the assistance of the user (Kallrath, 2004). This motivates us to include automatic differentiation as an additional, important feature of a modern AML.

The algebraic expressions are useful in describing individual models and 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, including statements for looping, testing, and assignment (Fourer, 2013). Therefore, scripting capabilities are an integral part of AMLs.

For this review, we have chosen five AMLs: AIMMS, AMPL, GAMS, JuMP, and Pyomo. The selection was based on the following criteria:

- AMLs which won 2012 INFORMS Impact Prize award2 dedicated to the originators of the five most important algebraic modelling languages: AIMMS, AMPL, GAMS, LINDO/LINGO, and MPL;
- the popularity of AMLs based on NEOS Server model input statistics for the year 2020;3
- open-source options that are attractive for the academic society or in situations where budgets are tight.

We have chosen to include GAMS and AMPL based on NEOS Server popularity, respectively, with 49% and 47% share of jobs executed via the NEOS platform in 2020. AIMMS was added as an example of AML with a graphical application development environment. JuMP and Pyomo were included as the most prominent open-source AMLs. We have decided to exclude MPL since it has not been updated for the last five years. We have also excluded LINGO as a solver specific modelling language.

---

1 Specifying the problem’s properties: space, set of constraints and optimality requirements.
3. Comparative Analysis of AMLs Characteristics

In the following section, we investigate how each of the chosen languages meets the requirements for a modern AML defined in the previous section. 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. Besides, an in-depth survey concluded by Robert Fourer in Linear Programming Software Survey (Fourer, 2017) is also used as a reference. Later on, a more practical comparison of AML characteristics is conducted to identify the potential ease of use of AML in daily work.

3.1. Comparative Analysis of the Features

We start by analysing how selected AMLs satisfy the three essential characteristics defined in the previous Section 2. In all reviewed AMLs, optimization problems are represented in a declarative way. Furthermore, since all of them are part of a specific modelling system, a clear separation between problem definition and the solution process in the context of the modelling system exists. The separation between the problem structure and its data is supported in all reviewed languages. It should be noted that GAMS, JuMP, and Pyomo 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 reviewed languages fulfill the essential characteristics of modern AMLs.

Next, in Table 1, we provide an overview of the key features each AML supports. For creating such a summary, we used the information provided by the AML vendors on their websites. All reviewed AMLs allow modelling problems in a solver independent manner. Additionally, AIMMS, JuMP, and Pyomo provide a more powerful way to define advanced algorithms using R, Julia, or Python programming languages. The ease of data input for the model differs among AMLs. While all of them support input from a flat file, some more advanced scenarios such as reading data from relational databases are more straightforward in AIMMS, JuMP, or Pyomo. AMPL and GAMS require a complicated setup instead (e.g. using ODBC drivers) to access the database. Wherein JuMP or Pyomo, a standard Julia or Python driver could be used to get data from relational and any other type of database supported by Python or Julia. Manipulation (e.g. transformation) of data is only supported by AIMMS, JuMP, and Pyomo.

When it comes to solver support, AMPL is the one supporting the most. However, it should be noticed that the categorization of solvers by supported problem types is different among vendors. Thus, in this comparison, we have reflected the information available from vendors harmonizing it across all of them. Solvers supported by JuMP and Pyomo require additional explanation. First, both AMLs support solvers compatible with AMPL (via AmpINLWriter package or ASL interface). Therefore, any solver that is equipped with
Table 1
Overview of AMLs features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>AIMMS</th>
<th>AMPL</th>
<th>GAMS</th>
<th>JuMP</th>
<th>Pyomo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modelling</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Scripting</td>
<td>Yes</td>
<td>Limited</td>
<td>Limited</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data</td>
<td>Yes</td>
<td>Limited</td>
<td>Limited</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Manipulation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Solvers</td>
<td>Total</td>
<td>13</td>
<td>47</td>
<td>35</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Global</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>8</td>
<td>17</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>MCP</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MINLP</td>
<td>3</td>
<td>6</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>MIP</td>
<td>5</td>
<td>14</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>MIQCP</td>
<td>5</td>
<td>5</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>NLP</td>
<td>6</td>
<td>19</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>QCP</td>
<td>6</td>
<td>9</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>Presolving</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Visualization</td>
<td>General</td>
<td>Paid</td>
<td>Paid</td>
<td>Paid</td>
<td>Free</td>
</tr>
<tr>
<td>License</td>
<td>Academic</td>
<td>Paid</td>
<td>Free</td>
<td>Free</td>
<td>Free</td>
</tr>
</tbody>
</table>

an AMPL interface can be used by JuMP or Pyomo. This could allow us to state that JuMP and Pyomo support all AMPL solvers. However, we have excluded solvers supported via the AMPL interface. It might be needed for some commercial solvers to request a particular version from the solver’s vendor that comes with the AMPL interface. Second, since both AMLs are open-source, multiple third-party packages add support for specific solvers for each of AMLs. In Section 1, we counted only the solvers mentioned on the official JuMP and Pyomo websites.

Presolving capabilities are only available in AIMMS and AMPL. JuMP and Pyomo have programming interfaces for creating custom presolvers, however, none of them are provided out of the box. Only AIMMS provides a visualization of the solver results out of the box. Using Python or Julia libraries, it is possible to visualize the results produced by Pyomo and JuMP. However, it requires custom development, and none of the standard JuMP or Pyomo libraries are supporting that.

It is important to conclude that JuMP and Pyomo are open-source AMLs built on top of general-purpose programming languages, making them fundamentally different from the competitors. This allows researchers familiar with Julia or Python to learn, improve, and use JuMP or Pyomo much more comfortably. At the same time, it is practically impossible to introduce improvements to commercial counterparts.

3.2. Practical Comparison of AMLs

For the first practical comparison of the selected AMLs, a classical Dantzig Transportation Problem was chosen (Dantzig, 1963). In this problem, we are given the factories’ supplies and the markets’ demands for a single commodity. We have also given the unit costs of
shipping the product from factories to the markets. The goal is to find the least costly shipping schedule that meets the requirements at markets and supplies at factories.

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

- model size in bytes;
- model size in the number of code lines;
- model size in the number of language primitives used;
- model instance creation time.

Since the transportation problem is a linear programming (LP) type of problem, we have chosen to measure the model instance creation time as the time needed to export a concrete model instance to MPS\(^4\) format supported by most LP solvers. The following sources provided sample implementations of the transportation problem for the AMLs under consideration:

- AIMMS Wikipedia page;\(^5\)
- AMPL model in GNU Linear Programming Kit;\(^6\)
- GAMS Model Library;\(^7\)
- JuMP Examples;\(^8\)
- Pyomo Gallery.\(^9\)

Transportation problem models in all five AMLs are given in Appendix B, Listings 3–6. 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 modeller in the AIMMS IDE and is typically generated by the AIMMS IDE. Moreover, it is worth noting that for the sake of simplicity, the problem model samples are concrete models, i.e. data of the model instance is described alongside the model structure.

A comparison of the sample Transportation Problem model’s characteristics in all reviewed AMLs is given in Table 2. The simplification of the model implementations provided in the literature sources is made in the following way:

- all optional comments, explanatory texts, and documentation are removed;

---

\(^4\)http://lpsolve.sourceforge.net/5.5/mps-format.htm.
\(^7\)https://www.gams.com/latest/gamslib_ml/libhtml/index.html.
\(^8\)https://github.com/jump-dev/JuMP.jl/tree/master/examples.
\(^9\)https://github.com/Pyomo/PyomoGallery.
Table 3

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>AMPL</th>
<th>GAMS</th>
<th>JuMP</th>
<th>Pyomo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Non zero elements</td>
<td>13</td>
<td>19</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Variables</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

- 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.

It can be seen from Table 2 that models implemented in AMPL, GAMS, and JuMP are the most compact ones, while the model written in AIMMS is much more verbose, and Pyomo lies in the middle. AIMMS propagates the creation of models using a graphical user interface (GUI) while keeping the model’s source code hidden from the modeller. Naturally, there is not much focus on how the model is stored. We can argue that while the GUI-based approach might be convenient to some modellers, it enforces greater vendor lock-in and makes the model’s extensibility and maintainability harder.

While comparing the number of language primitives required to create a model, JuMP and AMPL showed the best results, which allows us to predict that these modelling languages might have a more gentle learning curve. Therefore, we can conclude that in the context of the reviewed algebraic modelling languages, JuMP allows formulating an optimization problem most concisely.

The creation time of the transportation problem model instance defined in each AMLs was used to measure 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 the AIMMS system’s creators did not respond to an academic license request, we could not include AIMMS in the benchmark. Generated model instances in MPS format can be found in our GitHub repository’s models directory (Jusevičius and Paulavičius, 2019).

The characteristics of the created model instances can be seen in Table 3. We can conclude that all modelling languages have created a model instance using the same amount of variables and constraints. However, the definition of nonzero elements is different between GAMS and other modelling systems.

In Table 4, the benchmark results of model instance creation time are provided. We have tried to run multiple consecutive model instance creations (10 runs, 100 runs) to identify if the modelling system uses any caching. 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, the poor JuMP
performance confirms the Dunning et al. (2017) statement that JuMP has a noticeable start-up cost\(^\text{10}\) of a few seconds even for the smallest instances. In our case, only the initialization of the JuMP package took around 7 seconds. We also observed a significant speed-up in multiple consecutive model instances creation, which also confirms (Dunning et al., 2017) results. When a family of models is solved multiple times within a single session, this compilation cost is only paid for the first time that an instance is solved.

### 4. Performance Benchmark of AMLs

All examined AMLs support all types of traditional optimization problems; however, it is unclear how efficiently each AML can handle large model loading and what optimizations are applied during model instance creation. It would also be of great value to analyse how each of the modelling languages performs within an area of the specific type of optimization problems (e.g. linear, quadratic, nonlinear, mixed-integer). 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 analysed AMLs has to be used.

#### 4.1. AMLs Testing Library

We have chosen the GAMS Model Library\(^\text{11}\) 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 (Jusevičius and Paulavičius, 2019). A detailed explanation of how the test library creation tool works and the issues identified in the GAMS Library are provided in Appendix A. As a result of the transformation, we compiled a library consisting of 296 sample problems in AMPL, GAMS, JuMP, and Pyomo scalar model formats.

#### 4.2. Model Instance Creation Time

The generated library was used to determine the amount of time each modelling system requires to create a model instance of a particular problem. We wrote load-

---

\(^{10}\)Start-up cost consists of the precompilation and caching time required to prepare JuMP environment.

benchmark.sh shell script available in the tools directory of our GitHub repository (Jusevičius and Paulavičius, 2019), which loads each model into the particular modelling system and then exports it to the format understandable by the solvers. We have chosen the .nl (Gay, 2005) format as the target format acceptable by the solvers, as .nl supports a wide range of optimization problem types. The benchmark measures the time the modelling system takes to perform both model instance creation and export operations.

We have chosen to exclude sample problems with conversion errors from the benchmark (more information about them in Appendix A). Only the models that were successfully processed by all modelling systems were compared. This reduced the scope of our benchmark to 268 models.

Benchmark methodology, hardware, and software specifications can be found in our GitHub repository (Jusevičius and Paulavičius, 2019). Detailed results are available in the model-loading-times.xlsx workbook in the benchmark section of our GitHub repository (Jusevičius and Paulavičius, 2019). We have provided a summary of the average model instance creation time split by the problem type in Fig. 1. We can see the trend exhibited in the transportation problem model benchmark persists. AMPL is still a definite top performer, while JuMP and Pyomo perform the worst. There are no significant variations between different optimization problem types except for JuMP, where the model instance creation time tends to vary significantly while working with different types of problems. Moreover, as confidence intervals show, the variation between different models of the same type is also more significant once using JuMP. We tend to believe this is caused by Julia’s dynamic nature and the mix of run time compilation and caching of similar JuMP models.

We have observed that the average difference between AMPL and other contenders increases when the models become larger. 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, 38 times the difference between AMPL and Pyomo, and close to 100 times the difference between AMPL and JuMP. The difference between GAMS and Pyomo stayed roughly the same – around 3.5 times. The summary of the large model instance creation time can be seen in Fig. 2.

Thus, we can conclude that out of the reviewed AMLs, AMPL is a clear top-performing AML when it comes to the model instance creation time.

4.3. JuMP Benchmark

A similar time benchmark of the model instance creation has already been conducted (Dunning et al., 2017), where a smaller set of large models is used. While some of the trends exhibited in our benchmark persist (AMPL is the fastest, GAMS comes second), JuMP performance in our and Dunning et al. (2017) benchmarks differs significantly. This leads us to compare the benchmark methodology and results by conducting the benchmark described by Dunning et al. (2017).

First of all, our and their time benchmark methodologies differ. In comparison, we are trying to be solver independent and instruct AML to export the generated model instance to NL file, Dunning et al. (2017) attempt to solve the model using Gurobi solver and measure the time until Gurobi reports model characteristics. We believe that while our approach can be impacted by the system’s input/output performance, using a specific solver heavily depends on how the solver interface is implemented for a particular AML.

In the following, we have conducted two benchmarks – one as described in the original article and the second one using our method of exporting to a NL file. Results of the benchmarks can be seen in Tables 5, 6.
Table 5

<table>
<thead>
<tr>
<th>Model</th>
<th>AMPL</th>
<th>GAMS</th>
<th>Pyomo</th>
<th>JuMP (DIRECT)</th>
<th>JuMP (CACHE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lqcp-500</td>
<td>2093</td>
<td>2271</td>
<td>17000</td>
<td>17388</td>
<td>37317</td>
</tr>
<tr>
<td>lqcp-1000</td>
<td>8075</td>
<td>11995</td>
<td>139201</td>
<td>24590</td>
<td>44575</td>
</tr>
<tr>
<td>lqcp-1500</td>
<td>18222</td>
<td>38813</td>
<td>322604</td>
<td>39370</td>
<td>66566</td>
</tr>
<tr>
<td>lqcp-2000</td>
<td>32615</td>
<td>93586</td>
<td>575406</td>
<td>57597</td>
<td>88833</td>
</tr>
<tr>
<td>fac-25</td>
<td>407</td>
<td>480</td>
<td>7442</td>
<td>17517</td>
<td>39245</td>
</tr>
<tr>
<td>fac-50</td>
<td>2732</td>
<td>2884</td>
<td>43106</td>
<td>21331</td>
<td>47735</td>
</tr>
<tr>
<td>fac-75</td>
<td>9052</td>
<td>12422</td>
<td>150550</td>
<td>31582</td>
<td>57432</td>
</tr>
<tr>
<td>fac-100</td>
<td>20998</td>
<td>29144</td>
<td>393200</td>
<td>61326</td>
<td>93129</td>
</tr>
</tbody>
</table>

Before running the benchmarks, we had to rewrite some parts of the sample lqcp and facility JuMP models since syntax changes were introduced between JuMP v0.12 (used by Dunning et al. (2017)) and JuMP v0.21.5 (used by us, the latest version at the time of writing). Our benchmark was also conducted using newer versions of other AMLs—AMPL Version 20190207, GAMS v.32, Pyomo 5.7 (Python 3.8.3), Gurobi 9.0.

Additionally, we wanted to test JuMP’s new abstraction layer’s performance for working with solvers called MathOptInterface.jl (MOI). Therefore, we have tried both CachingOptimizer and DIRECT modes. As seen in Table 5, the DIRECT mode performed much better than the CachingOptimizer mode for both lqcp and facility models. An average difference in the instance creation time is close to two times, which leads us to suggest modellers evaluate MOI type choice based on specific use cases carefully.

Overall, both benchmarks confirmed our observation that JuMP suffers from the long warm-up time required to pre-compile JuMP libraries. Results were also consistent with the patterns exhibited during the full gamslib benchmark performed earlier. We could not reproduce the JuMP performance metrics reported by Dunning et al. (2017), where JuMP always outperforms Pyomo. Using the original benchmark method, JuMP outperformed Pyomo only once the model size increased. However, while using our export to NL file method, JuMP, on the contrary, started to fall behind Pyomo once model size increased.

The reported differences between our and the original benchmark (Dunning et al., 2017) might be caused by multiple factors such as different JuMP versions used, improved
Table 7

<table>
<thead>
<tr>
<th>Type</th>
<th># models</th>
<th># infeasible</th>
<th>Presolved (%)</th>
<th>Constraints reduced (%)</th>
<th>Variables reduced (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNS</td>
<td>4</td>
<td>0</td>
<td>100.00%</td>
<td>14.63%</td>
<td>31.39%</td>
</tr>
<tr>
<td>DNLP</td>
<td>5</td>
<td>0</td>
<td>20.00%</td>
<td>0.00%</td>
<td>7.41%</td>
</tr>
<tr>
<td>LP</td>
<td>57</td>
<td>0</td>
<td>36.84%</td>
<td>17.81%</td>
<td>9.66%</td>
</tr>
<tr>
<td>MCP</td>
<td>19</td>
<td>0</td>
<td>89.47%</td>
<td>47.00%</td>
<td>8.56%</td>
</tr>
<tr>
<td>MINLP</td>
<td>21</td>
<td>1</td>
<td>61.90%</td>
<td>16.32%</td>
<td>9.30%</td>
</tr>
<tr>
<td>MIP</td>
<td>61</td>
<td>0</td>
<td>60.66%</td>
<td>19.06%</td>
<td>11.50%</td>
</tr>
<tr>
<td>MIQCP</td>
<td>5</td>
<td>2</td>
<td>60.00%</td>
<td>0.00%</td>
<td>2.38%</td>
</tr>
<tr>
<td>MPEC</td>
<td>1</td>
<td>0</td>
<td>100.00%</td>
<td>50.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>NLP</td>
<td>101</td>
<td>2</td>
<td>47.52%</td>
<td>9.71%</td>
<td>11.55%</td>
</tr>
<tr>
<td>QCP</td>
<td>10</td>
<td>0</td>
<td>60.00%</td>
<td>7.10%</td>
<td>2.55%</td>
</tr>
<tr>
<td>RMIQCP</td>
<td>2</td>
<td>0</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Total</td>
<td>286</td>
<td>5</td>
<td>52.80%</td>
<td>18.42%</td>
<td>10.73%</td>
</tr>
</tbody>
</table>

Pyomo performance, or different Gurobi versions. It is important to stress that JuMP is a very actively developed AML that underwent significant changes during the last years. We think that it could be valuable to explore why the performance could have degraded and the reasons for such slow I/O operations performance revealed during writing to a NL file benchmark.

5. Presolving Benchmark

Another performance-related feature of AMLs is the ability to presolve a problem before providing it to the solver. The presolver can preprocess problems and simplify, i.e. reduce the problem size or determine the infeasible problem. Only two of the reviewed algebraic modelling languages provide presolving capabilities – AMPL (Fourer, 2003) and AIMMS. Since we did not have the opportunity to evaluate the AIMMS modelling language practically, we could only examine AMPL presolving capabilities.

5.1. Presolving in AMPL

To assess AMPL’s presolving performance, we gathered presolving characteristics while performing the model instance creation time benchmark. We have used 286 models that were successfully converted from GAMS original model to the 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 (Jusevičius and Paulavičius, 2019), while the summary of it can be found in Table 7. We observed that AMPL presolver managed to simplify the models in 52.8% of the cases, out of which 5 times it could determine that the problem solution is not feasible, thus not even requiring to call the solver. On average, once applied, the 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, leading to improved solution finding performance once invoking a solver with an already reduced problem model instance. Moreover, determining not feasible models can help modellers debug the problem definition process and find errors in the model definition. This allows us to argue that presolving is an important capability of any modern AML.

5.2. Presolve Impact on Solving

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 that AMPL presolve determined to be not feasible were excluded from the benchmark. Shell script solve-benchmark.sh provided in the tools directory of our GitHub repository (Jusevičius and Paulavičius, 2019) 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 the models using Gurobi and BARON solvers. Gurobi Optimizer (v.8.1.0) was chosen for solving LP, MIP, QCP, and MIQCP type of problems. At the same time, BARON (v.18.11.12) global solver was chosen for solving NLP, MINLP, MCP, MPEC, CNS, and DNLP problems. The solvers’ choice was motivated by the support for particular problem types and the popularity of solvers based on NEOS Server statistics. Two attempts to solve each model were made. One with AMPL presolver turned on (default setting), and the 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 presolve mechanisms (Puranik and Sahinidis, 2017; Achterberg et al., 2019). Thus the provided model is simplified by the solver too. This might result in very similar models being solved by the solver despite the AMPL presolve being turned on or off. However, the focus was on estimating AMPL presolve impact in real-life situations; therefore, a full benchmark was executed without changing the default solver behaviour. Later on, an additional benchmark was made to estimate 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 (Jusevičius and Paulavičius, 2019) file ampl-solving-times.xlsx sheet Benchmark 1. Table 8 summarizes the positive and negative impact AMPL presolve had on solving the problems iteration and time-wise. Positive impact means fewer iterations or time was needed to solve a problem once the presolve was turned on. A negative impact means the opposite that more iterations or time was required. The impact is considered neutral if the number of iterations did not change or the required time was within the one-second tolerance level.

---

Table 8
Summary of AMPL presolve impact on solving.

<table>
<thead>
<tr>
<th></th>
<th>Iteration-wise</th>
<th>Time-wise</th>
<th>Iteration-wise (%)</th>
<th>Time-wise (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>37</td>
<td>67</td>
<td>26.43%</td>
<td>47.86%</td>
</tr>
<tr>
<td>Neutral</td>
<td>74</td>
<td>40</td>
<td>52.86%</td>
<td>28.57%</td>
</tr>
<tr>
<td>Negative</td>
<td>29</td>
<td>33</td>
<td>20.71%</td>
<td>23.57%</td>
</tr>
</tbody>
</table>

Table 9
AMPL presolve impact with Gurobi presolve on.

<table>
<thead>
<tr>
<th></th>
<th>Iteration-wise</th>
<th>Time-wise</th>
<th>Iteration-wise (%)</th>
<th>Time-wise (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>18</td>
<td>39</td>
<td>28.57%</td>
<td>61.90%</td>
</tr>
<tr>
<td>Neutral</td>
<td>34</td>
<td>0</td>
<td>53.97%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Negative</td>
<td>11</td>
<td>24</td>
<td>17.46%</td>
<td>38.10%</td>
</tr>
</tbody>
</table>

Table 10
AMPL presolve impact with Gurobi presolve off.

<table>
<thead>
<tr>
<th></th>
<th>Iteration-wise</th>
<th>Time-wise</th>
<th>Iteration-wise (%)</th>
<th>Time-wise (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>33</td>
<td>44</td>
<td>54.10%</td>
<td>72.13%</td>
</tr>
<tr>
<td>Neutral</td>
<td>10</td>
<td>0</td>
<td>16.39%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Negative</td>
<td>18</td>
<td>17</td>
<td>29.51%</td>
<td>27.87%</td>
</tr>
</tbody>
</table>

During this benchmark, 6 models failed to be solved due to solver limitations. Two models were deemed to be not feasible, and two were solved during the AMPL presolve phase. Solvers were capable of solving 41 models during the solver’s presolve phase. Moreover, for six models, the mismatching objective was found with AMPL presolve turned on and off. Overall, AMPL presolve positively impacted 26.43% of the cases iteration-wise and 47.86% time-wise. However, it hurt 20.71% of cases iteration-wise and 23.57% time-wise.

As mentioned earlier, both BARON and Gurobi solvers have their presolve mechanisms. An additional benchmark was made to test the impact of AMPL presolve with disabled solvers presolving. Since only Gurobi allows the user to disable presolve functionality, a subset of models previously solved with Gurobi was chosen. Detailed benchmark results can be seen in our GitHub repository’s (Jusevičius and Paulavičius, 2019) file ampl-solving-times.xlsx sheet Benchmark 2. The summary of the benchmark is provided in Tables 9, 10. Gurobi could not solve two MIP problems (clad and mws) in a reasonable time once Gurobi’s presolve functionality was turned off. Those models were excluded from the benchmark.

As seen once comparing these results in Tables 9 and 10, the AMPL presolve had a greater positive effect both iteration-wise (+25.5%) and time-wise (+10.2%) once Gurobi presolve was turned off. AMPL presolve also had a less neutral impact once the solver presolving was off, thus leading to the conclusion that during the first benchmark, some models were simplified to 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, a positive impact
is always more significant than the negative one, and it especially becomes evident once the solver does not have or use its problem presolving mechanisms. This allows us to conclude that the presolving capability of AML is an important feature of a modern algebraic modelling language. We can also advise choosing AML having presolving capabilities in cases the solver used to solve the problem does not have its presolving mechanism.

6. Conclusions and Future Work

From the research, we can conclude that AMPL allows us 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 helps the modellers in both problem definition and efficient solution finding processes. GAMS is a powerful runner-up providing very similar to AMPL problem formulation capabilities although running behind in the model instance creation time. AIMMS can be considered as being in the class of its own as it has taken a purely graphical user interface based approach. Since we could not examine the performance characteristics of AIMMS due to a lack of academic license, the performance aspect remains unclear. Open-source alternatives JuMP and Pyomo are on par with commercial competitors in the problem definition process. However, the performance of model instance creation is a bit behind compared to its competitors. JuMP suffers from noticeable environment start-up costs, while Pyomo performance tends to downgrade once the model’s size increases.

We plan to continue our research in this area by including performance comparison on automatic differentiation, adding even more large problems to our test library, and exploring the potential of parallel model instance creation support by AMLs.

Data Access Statement

Data underlying this article can be accessed on Zenodo at https://zenodo.org/record/4106728, and used under the Creative Commons Attribution license.

A. Creation of AMLs Testing Library

The automated shell script gamslib-convert.sh is available in the tools directory of our GitHub repository (Jusevičius and Paulavičius, 2019) was created to generate the AMLs testing library. The script uses GAMS Convert tool v.32\textsuperscript{15} to convert the GAMS proprietary format model to a scalar model in AMPL, GAMS, JuMP, and Pyomo formats. Characteristics of the sample problem models (number of equations, variables, discrete variables, non-zero elements, and non-zero nonlinear elements) are automatically

\textsuperscript{15}https://www.gams.com/latest/docs/S_CONVERT.html.
extracted and noted. Sample problems are also grouped based on optimization problem types.

The script has two execution modes – one for converting a single model and another for converting all GAMS Library models. An example of the transportation problem from GAMS Model Library\(^\text{16}\) converted to GAMS scalar format is shown in Listing 1.

At the time of writing, there were 423 models in the GAMS Model Library. Out of them, we eliminated 66 models using GAMS proprietary modeling techniques (e.g. MPSGE, BCH Facility), 20 using general-purpose programming language features (e.g. cycles), four models tightly coupled to CPLEX and DECIS solvers. 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. This means, some models in the GAMS Library are not compatible with the GAMS modeling system itself. While performing the model instance creation benchmark, we have identified that 12 AMPL, 11 JuMP, and 29 Pyomo models generated by the GAMS Convert tool had errors in them. Most of the Pyomo errors were caused by an incorrect GAMS Convert tool behaviour where the definition of the Suffix primitive uses AMPL but not Pyomo semantics. Similar issues were observed in some of the JuMP models. An example of what GAMS Convert generates and the correct Pyomo syntax can be seen in Listing 2.


---

### Listing 1. Transportation problem converted to GAMS scalar model.

```gams
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;
```

### Listing 2. Example of a GAMS Convert error.

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

---
B. Transportation Problem Models

Listing 3. Transportation problem defined in AMPL format.

```ampl
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];
subject to supply{i in I}: sum{j in J} x[i,j] <= a[i];
subject to 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 3, Transportation problem defined in AMPL 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 4. Transportation problem defined in GAMS 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)

Listing 5. Transportation problem defined in Pyomo format.
using JuMP
ORIG = ["Seattle", "San-Diego"]
DEST = ["New-York", "Chicago", "Topeka"]
supply = [350, 600]
demand = [325, 300, 275]
cost = [2.5 1.7 1.8;
2.5 1.8 1.4]

F = 50

cost_f = [F * cost[i,j] / 1000
   for i in 1:length(ORIG), j in 1:length(DEST)]

model = Model()

@variable(model, trans[1:length(ORIG), 1:length(DEST)] >= 0)
@objective(model, Min, sum(cost_f[i, j] * trans[i, j]
   for i in 1:length(ORIG), j in 1:length(DEST)))
@constraint(model, [i in 1:length(ORIG)],
   sum(trans[i, j] for j in 1:length(DEST)) <= supply[i])
@constraint(model, [j in 1:length(DEST)],
   sum(trans[i, j] for i in 1:length(ORIG)) >= demand[j])

Listing 6. Transportation problem defined in JuMP format.

References


V. Jusevičius received the master’s degree in software engineering from Vilnius University, Vilnius, Lithuania, in 2011. In 2017 he started PhD studies in computer science at Vilnius University, Institute of Data Science and Digital Technologies. His thesis title “Research and Development of an Open Source System for Algebraic Modeling Languages”. He is working as a partnership associate professor in Vilnius University, Institute of Computer Science, and as a Chief Software Architect for Danske Bank A/S.

R. Oberdieck obtained his bachelor and MSc degrees from ETH Zurich in Switzerland (2009–2013), before pursuing a PhD in Chemical Engineering at Imperial College London, UK, which he completed in 2017. After using his knowledge in mathematical modelling and optimization in the space of renewable energies at the world leader in offshore wind energy, Ørsted A/S, he is now helping companies around the world to unlock business value through mathematical optimization as a Technical Account Manager for Gurobi Optimization, LLC.

R. Paulavičius received the PhD degree in computer science from Vytautas Magnus University, Kaunas, Lithuania, in 2010. He was a postdoctoral researcher with Vilnius University, Vilnius, Lithuania, and a research associate with Imperial College London, London, UK. He is currently a professor and the Head of the Blockchain Technologies Group, Institute of Data Science and Digital Technologies, Vilnius University. His research interests include optimization, distributed ledger technologies, parallel and distributed computing, machine learning, and the development and application of various operation research techniques.