Proposal for MOSDEX, an Alternative to MPS for Data Exchange with Optimization Solvers

Dr. Jeremy A. Bloom
jeremyblmca@gmail.com
December 23, 2018

Abstract

MPS format, which has become the standard for data exchange with mathematical optimization solvers, has serious deficiencies when it comes to supporting modern decision applications based on optimization. This paper proposes a new standard called MOSDEX to overcome them. MOSDEX is based on several principles: independence from and support for multiple optimization solvers and their APIs and for multiple algebraic modeling languages, model-data separation, relational data modeling, and incorporation of standard optimization modeling objects. MOSDEX uses the widely adopted JSON data format standard to take advantage of JSON support in a variety of programming languages including Java, C++, Python, and Julia. The paper demonstrates the principles of MOSDEX through examples taken from well-known optimization problems. The paper concludes with recommendations for next steps in developing MOSDEX.

1. Introduction and Basic Rationale

The MPS format "has emerged as a de facto standard ASCII medium among most of the commercial LP solvers." (Wikipedia, 2018). Although its usage has declined with the widespread adoption of algebraic modeling languages such as OPL, AMPL, and GAMS, the need for a non-proprietary standard that can support multiple solvers remains strong. However, the limitations of the MPS format are becoming increasingly apparent, and therefore in this document, we want to propose an alternative called **MOSDEX** (**M**athematical **O**ptimization **S**olver **D**ata **Ex**change), which hopefully can become a new standard to supersede the older MPS standard.

Before addressing the deficiencies we hope to remedy with an alternative, let us first consider the advantages that MPS brings:

1. **Sparsity**: MPS supports a sparse data format that requires specifying only non-zero elements of the data. Since most real-world optimization problems are highly sparse (1-10% non-zeros), this feature results in substantial reductions data volume and computational effort. In addition, all professional-grade solvers use sparse data structures internally.
2. **Text-based**: MPS files are ordinary text files and are reasonably easy for humans to read.
3. **Non-proprietary**: Although MPS began is a proprietary standard for an IBM solver, it became widely adopted, and today almost all solvers support it.

However, despite these advantages, MPS suffers from a number of shortcomings. Among these, its fixed-column format, while archaic, is perhaps the least important, since that aspect is rather easily overcome and many solvers that accept MPS do not enforce it. More important are the following:

a. **Lack of an output standard**: MPS is a format for input and there exists no corresponding standard for output from an optimization solver.

b. **Lack of model-data separation**: This aspect means that the model (variables, constraints, objective) is intertwined with the data that populates it rather than specified independently of the data populating any particular instance. It is considered a best practice the separate them, for a number of reasons discussed below. Furthermore, the widely used modeling languages are designed for model-data separation.
c. **Difficulty in scaling:** One reason for specifying the model and data separately is that it allows for scalability – the model remains unchanged as the size of the data changes. In practice, an optimization application often consists of a family of related instances in which a model, representing for instance a distribution network, does not change but the data represents different numbers of entities, such as warehouses, stores, and routes between them.
d. **Lack of indexing**: One way to achieve scalability is to use indexing to represent groups of related variables or constraints, a common practice in mathematics. For example, instead of giving each variable a distinct name, such as *x, y, z*, one uses an index like this x_1, x_2, x_3. In MPS, each variable is represented as a column and thus there is a column record for each. With indexing, each family of variables could be represented by a single record (although the use of column records to specify matrix data would also need to be revised as discussed in the next item).
e. **Column orientation**: In MPS, the matrix elements (i.e. the coefficients of each variable in the constraints) are specified in the column records. (This aspect is probably an historical artifact of the way that a sparse matrix was specified in the original IBM solver.) While column orientation is appropriate for some optimization models (for example a model constructed using Dantzig-Wolfe decomposition), for others a row orientation is more appropriate. Indeed, in mathematical notation, an optimization model is usually represented by its constraints, and most of the widely adopted modeling languages favor specification by constraint (row orientation) over specification by variable (column orientation).
f. **Extensions beyond linear models**: Originally, MPS was intended to represent purely linear optimization problems. As solvers began to support integer and mixed-integer problems, their developers extended the MPS format to accommodate them, although no standard for such extensions emerged. Furthermore, solvers began to exploit special model structures in their algorithms, such as special-ordered sets or indicator constraints, again extending the MPS format to represent them. A replacement for MPS needs to support these extensions in a standard way.

Below, we will explain in detail MOSDEX, the proposed new standard. However, here is a brief summary:

a. **Represent the data in relational form**: Use the well-known structure of relational data bases, that is, a set of 2-dimensional tables each consisting of a fixed column schema and an indeterminate number of rows.
b. **Use the JSON (JavaScript Object Notation) standard**: Tie the standard format for optimization problems to a widely used data format standard to take advantage of support for the standard in various programming languages. We discuss below the reasons for choosing JSON over XML, the other widely adopted data format standard.
c. **When necessary, augment the data representation with mathematical modeling objects (variables, constraints, objectives, etc.) in the new standard**: When using a modeling

language, the data representation should be sufficient to fully specify an optimization problem. However, a full replacement of MPS would also need to specify the modeling objects. The proposed MOSDEX standard would support linear, integer and mixed-integer linear models, and various special structures. It would also be possible to extend MOSDEX to accommodate quadratic formulations. However, MOSDEX does not support general nonlinear formulations, which increase complexity enormously due to the need to represent more general mathematical expressions. (Note however, that the relational data representation would also support nonlinear formulations.)

2. The Relational Data Model

As discussed in my paper (Bloom, 2017), there is a deep relationship between the structure of the data used in optimization modeling and the constructs (variables, constraints) of an optimization model. In fact, as demonstrated in that paper, one can view the modeling layer of an optimization application as transforming the data from its external form in some sort of data store into its internal form in the solver's data structures. These transformations naturally take the form of SELECT queries in SQL. Given the widespread adoption of optimization domain-specific languages such as OPL, AMPL, and GAMS and of solvers' own modeling APIs for programming languages such as Python, a standard for representing the data in an optimization problem would accomplish almost all of the value in replacing the MPS standard.

The central object of the relational data model is the *table*, which is a two-dimensional array with a fixed number of columns and an indeterminate number of rows. The columns are defined by a *schema*, which is an ordered set of *fields*, each consisting of a column name and a data type. For the purposes of optimization modeling, only three primitive data types are used: string, integer (including binary), or floating point number (usually double precision); composite data types such as arrays are not allowed (however, arrays can be represented by tables that include key/value pairs). A set of one or more columns that uniquely defines each row in the table is called the table's *key*; key columns must use one of the discrete data types, string or integer. Non-key columns are used for numerical data as coefficients for constraints or objectives or as right-hand sides for constraints and thus must have a numeric type integer or floating point. A row in the table is called a *tuple*, and all rows must have the same schema. The components of a row tuple are called its data *items*.

The totality of the application data that populates an optimization problem instance, encompassing all of its tables, is called the *collector*, and the collection of their schemas is called the *application data model*. Specifying the application data model is one of the key steps in defining an optimization application. Generally, the inputs to and outputs from the solver should be specified separately, so there are actually two collectors and two corresponding application data models in an optimization application.

The table schemas play two roles in the architecture of an optimization application. First, they are used as part of the validation process to assure the data extracted from the source data systems conforms to the specification for input to the optimization model (validation includes other steps as well, such as checking for missing values or values out of range). Second, they are used to create the parser that reads the data files and translates them to the internal objects of the solver. Strictly speaking, this second use applies mostly to programming languages such as Java and C++ that are statically typed.

Dynamically typed languages such as JavaScript, Python, and Julia can often infer the schema from the data in order to create the necessary objects.

There are numerous ways to represent a schema. The following illustrates two of them. The first figure is the representation in OPL and the second is the corresponding representation in JSON (as created by Apache Spark):

Figure 1: Representation of the warehouses schema in OPL

```
tuple Warehouse {
 key string location;
 float fixedCost;    // $/yr
 float capacityCost; // $/pallet/yr
 }
```

Figure 2: Representation of the warehouses schema in JSON

```
{
  "type" : "struct",
  "fields" : [ {
    "name" : "location",
    "type" : "string",
    "nullable" : false,
    "metadata" : { }
  }, {
    "name" : "fixedCost",
    "type" : "double",
    "nullable" : false,
    "metadata" : { }
  }, {
    "name" : "capacityCost",
    "type" : "double",
    "nullable" : false,
    "metadata" : { }
  } ]
}
```

3. Representing the Data in Relational Form

Using the relational model, the data for an optimization application can be represented by a three-level JSON object:

1. Collector: object
2. Table: array
3. Tuple: object

Figure 2 illustrates the structure using JSON.

Figure 2: Relational Model for Optimization Data

```
{                                          //Start Collector Object
 "firstTableName" : [                      //Start Table Array
```



```
      {                                    //Start first Tuple Object
       "firstFieldName" : "firstFieldValue",//Tuple Field
        ...                                //More Tuple Fields
       "lastFieldName" : "lastFieldValue"   //Tuple Field
      },                                   //End first Tuple Object
      ...                                  //More Tuples
      {                                    //Start last Tuple Object
       "firstFieldName" : "firstFieldValue",//Tuple Field
        ...                                //More Tuple Fields
       "lastFieldName" : "lastFieldValue"   //Tuple Field
      }                                    //End last Tuple Object
    ],                                     //End Table Array
      ...                                  //More Tables
 "lastTableName" : [                       //Start Table Array
      {                                    //Start first Tuple Object
       "firstFieldName" : "firstFieldValue",//Tuple Field
        ...                                //More Tuple Fields
       "lastFieldName" : "lastFieldValue"   //Tuple Field
      },                                   //End first Tuple Object
      ...                                  //More Tuples
      {                                    //Start last Tuple Object
       "firstFieldName" : "firstFieldValue",//Tuple Field
        ...                                //More Tuple Fields
       "lastFieldName" : "lastFieldValue"   //Tuple Field
      }                                    //End last Tuple Object
    ]                                      //End Table Array
}                                          //End Collector Object
```

A specific example for a warehouse location problem is found in my paper (Bloom, 2017).

4. Use of JavaScript Object Notation (JSON)

We want to tie the MOSDEX standard format for optimization problems to a widely used data format standard to take advantage of support for the standard in various programming languages. There are two obvious candidates, JSON and XML. Of the two, JSON has a less complex syntax and is somewhat easier to construct and read by humans, while XML has greater expressive capabilities. The distinction between the two is that JSON is primarily a data format which parses text into data objects. On the other hand, XML is a markup language, which identifies elements in a text file with tags that can be read by a specialized parser. We recommend using JSON although the basic principles of MOSDEX could just as easily be implemented in XML. We note that an analogous standard for machine learning exists in XML called PMML (**P**redictive **M**odeling **M**arkup **L**anguage) (Data Mining Group) which may serve as a template for MOSDEX.

In dynamically typed languages such as JavaScript, Python, and Julia, objects can be created on-the-fly by parsing JSON. In contrast, statically typed languages such as Java and C++ require predefinition of the classes into which the JSON data will be read. These classes constitute the Domain Object Model (DOM) of the application, which is analogous to the application data model described above. Having created the DOM, a JSON parser can automatically instantiate objects of the DOM classes. One such parser is FasterXML, also known as Jackson (Faster:XML) which also supports JSON. Developing the DOM is a straightforward, if tedious process, for a human developer, but tools exist to automate the process

given the schemas of the application data model. The paper (Bloom, 2017) shows an alternative, using Apache Spark datasets rather than DOM objects to hold the application data.

It is probably worth comparing the MOSDEX proposal with the Optimization Services proposal (Gassmann, H., Ma, J., and Martin, K., 2016) and the related OSiL (Fourer, R., Ma, J., and Martin, K. , 2010). These are based on XML and have a broader purpose than MOSDEX to address the needs of cloud-based optimization applications. However, they have a complex syntax, and more importantly, they do not appear to observe model-data separation nor to use indexing beyond simple integer sequences.

5. Optimization Modeling

On many optimization platforms, the MOSDEX data standard described above would be sufficient. Domain-specific languages such as OPL, AMPL, and GAMS provide the capability to translate an algebraic representation of an optimization model into internal objects used by a solver. Similarly, many of the popular solvers, including CPLEX and Gurobi provide modeling APIs that serve the same purpose. In either case, a standard data format would provide a non-proprietary means to populate those objects with data.

However, since MPS also provides a low-level capability to specify the essential objects of a mathematical optimization model, a superseding standard should perhaps also offer that capability. Bear in mind however, that a descriptive format such as MPS or its successor has limitations that can only be overcome through use of a domain-specific language or an API, because the latter involve actual computations beyond the delivery of data to the solver.

There are several possibilities that we discuss below: model specification through SQL or model specification through generic constraints. Before turning to these proposals, however, we first address the common part of model specification.

A. Indexing with tuples

In mathematical optimization, use of vector and matrix notation is pervasive, and most solvers have a low-level interface that enables populating the solver's internal data structures with data in vector and matrix form. These interfaces have typically used MPS format, which also as a vector/matrix oriented format. However, developers of optimization models have long recognized the inadequacy of 1- and 2-dimensional representations of the multidimensional entities that arise in many optimization applications. Thus one of the key steps in developing an optimization model is *encoding* the multidimensional indices into one or two dimensional indices; indeed automatic encoding is one of the main reasons for using an algebraic modeling language.

Optimization model developers have increasingly moved to a more general notion of indexing that overcomes the limitations of the vector/matrix formulations, namely using tuples as indices. More precisely, the key fields of a tuple, which are unique within a table of such tuples, represent the index. This approach has two advantages. First, a key can be composed of an arbitrary number of fields (although as a practical matter, using more than 3 fields is rare). Second, filtering (that is, selecting a subset of the tuples that meet certain criteria) is straightforward both notationally and computationally.

Here is an example of tuple indexing, derived from the capacity constraint in a warehouse location model written in OPL (see the paper (Bloom, 2017)).

```
 forall(w in warehouses)
//   Cannot ship more out of a warehouse than its capacity
   ctCapacity[w]: capacity[w] >= sum(r in routes: r.location==w.location)
demand[r]*ship[r];
```

We suggest that specifying the objects composing an optimization model in this proposed MOSDEX standard be based on tuple indexing. In the specification below, we denote a table whose tuples include the key fields as an index_set (which may also have non-key data fields), and a specific item within a tuple as an index_set.component.

B. Specifying variables, constraints, and objectives

Specification of a variable, constraint, or the objective uses a JSON object with various named fields as shown below. The sets of variables and constraints are JSON arrays of the relevant objects. We denote a table whose tuple keys constitute the index as an index_set, and a specific item within a tuple as an index_set.component. A constraint whose sense is labeled "none" is simply an expression among the variables without a required bound; it can serve as the objective or a subexpression of the objective.

1. **Variables**:

```
variables: [
          {"name" : String,
           "index" : index_set,
           "type" : continuous or integer or binary
           "lower_bound" : index_set.component or number literal or "+/-infinity",
           "upper_bound" : index_set.component or number literal or "+/-infinity"
          }
          …
]
```

2. **Constraints**:

```
constraints: [
          {"name" : String,
           "index" : index_set,
           "direction" : "LE" or "GE" or "EQ" or "none",
           "right-hand_side" : index_set.component or number literal,
           "type" : linear or indicator or SOS or …
          }
          …
]
```

3. **Objective**:

```
objective: {  "name" : String,
              "row" : reference to a row among the constraints,
              "sense" : "max" or "min" }
```

C. Specifying coefficients with SQL

In my paper (Bloom, 2017), I demonstrate how the coefficients of an optimization model can be constructed using SQL queries on the application data tables. Hence, one way to specify the coefficients in MOSDEX is to use those SQL statements. An obvious objection to this approach is that optimization solvers generally do not execute SQL. However, in an optimization-based application, there is often a relational database component of the architecture that can be used for this purpose; indeed, the referenced paper (Bloom, 2017) demonstrates the use of Apache Spark. The input to the solver then uses its low-level vector/matrix API to read the results of the queries into its internal data structures.

Transformation of an optimization problem instance into a form acceptable to a solver's low-level API has three steps, each of which corresponds to a type of SQL query:

1. mapping the indices of the decision variables and constraints to columns and rows
2. encoding the data associated with the decision variables and constraints
3. encoding the coefficient data of the variables in the constraints as matrix entries

The paper (Bloom, 2017) illustrates each of these steps with an example.

MOSDEX represents each query as a JSON object and the entire set of queries as an array of these objects:

```
coefficients: [
          {“name” : String,
           “constraint”: reference to a constraint among the constraints,
           “variable”: reference to a variable among the variables,
           “query” : String
          }
          …
]
```

The queries themselves might be coded by hand by the model developer, a straightforward, if tedious process that requires an understanding of the SQL query language. Alternatively, they could be generated from the more familiar syntax of an optimization domain-specific language; that would of course require the language provider to augment their product to do so.

D. Specifying coefficients with generic constraints

The use of SQL as described in the previous section is completely general, meaning that any optimization problem could be specified by appropriate queries; however, as noted above, that leaves a gap in the optimization application development process when it comes to generating the relevant SQL. As an alternative, one could specify a type of generic constraint that covers a significant majority of the cases seen in actual practice. Consider the form of general linear constraints in an optimization problem:

for all (c in Constraint_Index_Set)
*sum(v in Variable_Index_Set, d in Data_Set where condition(c, v, d) is true) d.value * x[v] <= c.right-hand_side*

There may be multiple such constraints in a model, corresponding to different index sets for the constraints and variables.

The boolean function *condition* forms the WHERE clause of the SQL SELECT query that generates the coefficients, denoted here by *d.value*, a value field of the *Data_Set* table. Because an SQL engine is equipped to evaluate complex expressions, the condition function could be almost any syntactically correct expression. However, in an optimization model, the condition function is most often a set of key matchings, e.g.

d.key1==v.key1 & d.key2==c.key2 &…

Thus, a *generic* constraint in an optimization model would take the form

for all (c in Constraint_Index_Set)
*sum(v in Variable_Index_Set, d in Data_Set where d matches c & d matches v) d.value * x[v] <= c.right-hand_side*

The Boolean function *matches* is defined as follows:

d matches v is **true** if the tuples *d* and *v* have at least one key field in common and the corresponding values of the common fields are equal in both tuples, and otherwise **false**.

Note that the *matches* function is not transitive because the key fields in the expression *d matches c & d matches v* may differ among *c, v,* and *d*.

The pair of matches links the constraint and variable through the data, which must have common keys with both. By restricting the *condition* function to a pair of key matches, we can specify a generic constraint with a simple JSON object and require of the solver only the capability to evaluate the matches, which many do in their APIs. Representation of generic constraints in MOSDEX would then look something like this:

```
coefficients: [
          {“name” : String,
           “constraint” : reference to a constraint record including its index set,
           “variable” : reference to a variable record including its index set,
           “data” : reference to a data table including its index set,
          }
          …
]
```

In this object, the matching condition is implicit in the various index sets. Note that the formulation is more general than the generic constraint stated above, since each constraint can sum over multiple variables, each with its own index set.

6. Conclusion and Next Steps

This paper has proposed a new standard called MOSDEX to overcome deficiencies in the MPS standard for data exchange with optimization solvers. MOSDEX is based on several principles: independence from and support for multiple optimization solvers and their APIs and for multiple algebraic modeling languages, model-data separation, relational data modeling, and incorporation of standard optimization modeling objects. MOSDEX uses the widely adopted JSON data format standard to take advantage of JSON support in a variety of programming languages including Java, C++, Python, and Julia. The paper

has demonstrated the principles of MOSDEX through examples taken from well-known optimization problems.

The following steps are suggested for further development of MOSDEX:

1. Review and debate the appropriateness of MOSDEX vs. other potential replacements for MPS.
2. Develop examples of MOSDEX specification for widely understood optimization problems with a view towards testing and extending MOSDEX where necessary. Examples should include (but not be limited to) network models, time-staged models involving lagged variables (e.g. production/inventory problems), and stochastic programs.
3. Draft more rigorous syntax specifications for MOSDEX objects.
4. Propose MOSDEX extensions for special structures in optimization models (e.g. special ordered sets and indicator constraints).
5. Test how MOSDEX would interact with different solvers' APIs (e.g. CPLEX and Gurobi) and with different optimization domain-specific languages (e.g. OPL, AMPL, and GAMS).
6. Code parsers for reading and writing MOSDEX files in various languages, especially Java, C++, Python, and Julia.

References

Bloom, J. A. (2017). *Optimization Modeling and Relational Data.* Retrieved from https://github.com/JeremyBloom/Optimization---Sample-Notebooks/blob/master/Optimization%2BModeling%2Band%2BRelational%2BData%2Bpub.ipynb

Data Mining Group. (n.d.). *Predictive Model Markup Language.* Retrieved from http://dmg.org/

Faster:XML. (n.d.). *FasterXML.* Retrieved from http://fasterxml.com/

Fourer, R., Ma, J., and Martin, K. (2010). OSiL: An Instance Language for Optimization. *Comput. Optim. Appl., 45*(1), 181–203.

Gassmann, H., Ma, J., and Martin, K. (2016). Communication protocols for options and results. *Math. Prog. Comp., 8*, 161–189.

Wikipedia. (2018). *MPS (format).* Retrieved from https://en.wikipedia.org/wiki/MPS_(format)

About the Author

Dr. Jeremy A. Bloom retired in 2017 after a 40-year career in operations research. Most recently, he was responsible for IBM's Decision Optimization Center product, an application development and deployment platform using IBM's CPLEX optimization solver and its OPL algebraic modeling language. Prior to joining IBM, he worked in technical sales and product marketing at ILOG before its acquisition by IBM. Prior to joining ILOG, Dr. Bloom managed programs at the Electric Power Research Institute in power delivery asset management, retail market analysis and resource management for the restructured power industry, distributed energy resources, and integrated resource planning. While at EPRI, he was part of the leadership team of a spin-out providing information and market research for retail energy markets, and he was responsible for technical leadership of a proposal to manage

California's energy efficiency market transformation programs. Earlier, he spent a significant part of his career at General Public Utilities, where he was responsible for resource planning and demand-side management, including leading the company's first efforts to procure demand-side resources through competitive bidding. He began his career teaching operations research at Cornell University. Dr. Bloom received his undergraduate degree in electrical engineering at Carnegie-Mellon University and his graduate degrees in operations research from the Massachusetts Institute of Technology.

