**MOSDEX Examples**

Dr. Jeremy A. Bloom  
[jeremyblmca@gmail.com](mailto:jeremyblmca@gmail.com)  
February 3, 2019

The paper “Proposal for MOSDEX, an Alternative to MPS for Data Exchange with Optimization Solvers” (), presented a new syntax for specifying data and models for use in mathematical optimization. The current paper elaborates on the original and extends the syntax though a series of examples.

1. **MOSDEX Syntax**

JSON has three basic syntactic units

* An Object , enclosed in braces “{“ and “}”which consists of a series of named fields, and
* An Array or List, enclosed in brackets “[“ and “]”which consists of a list of unlabeled fields
* A field which consists of a label (if a component of an object), a quoted string followed by a colon, and a value, which is either a primitive (a quoted string or an unquoted number) or a nested object or array.

A MOSDEX file is structured as a JSON object (within braces) followed by the following fields

1. Problem – an object with specific defined primitive fields
2. Input\_Data\_Model – an object with user-defined object fields
3. Output\_Data\_Model – an object with user-defined object fields
4. Data – an object with user-defined object fields conforming to the Input\_Data\_Model
5. Variables – an array of specific defined fields, mostly primitives but occasionally objects
6. Constraints – an array of specific defined fields, mostly primitives but occasionally objects
7. Decision\_Expressions – an array of specific defined fields, mostly primitives but occasionally objects
8. Coefficients – an array of specific defined fields, mostly primitives but occasionally objects

All but the Data object should reside in a single file, but it is often useful for the Data to reside in a separate file, not only because of model/data separation, but also because the other objects may require a different type of parser. Parsing the Data parser often will use a streaming parser for reasons of efficiency with large data sets, whereas the other objects will often use a parser that creates objects in the host language.

The actual configurations of the objects other than the input and output data models and the data are still being defined (indeed, one of the purposes of the examples developed in this paper is to determine what information about them is needed). The examples themselves are currently the best explanation of these parts of the syntax.

Any identifier that starts with a capital letter represents a key word in the MOSDEX syntax; and any that does not represents a user-definable identifier. We are generally using “Snake\_Case” for key words and “camelCase” for user defined identifiers. Other conventions could be chosen to indicate key words, for instance using “ALL\_CAPS” style. Of course the list of keywords is still fluid at this point.

1. **The Volsay Example**

Volsay is a small, very simple linear programming problem for chemical production found at <https://www.ibm.com/support/knowledgecenter/en/SSSA5P_12.5.0/ilog.odms.ide.help/OPL_Studio/opllanguser/topics/opl_languser_shortTour_LP_prodplanning.html>; it has only two variables and three constraints, one of which is merely an upper bound on one of the variables. MOSDEX is certainly overkill for such a small problem, since the volsay.json file is many times larger than the OPL equivalent or its representation in MPS. In fact, it illustrates the misconceptions about mathematical optimization that can arise when applied to trivial, class-room style examples. It has no indexing, which is a critical feature of most realistic optimization models. It is inherently not scalable; consider how much work would be required to another variable, say one called “soda” to it. Model/data separation serves only to make the model file much larger and harder to read.

However, we chose Volsay to begin our exposition of MOSDEX for two reasons: 1) to show the overall structure of a MOSDEX file and 2) to illustrate several features of the syntax.

Since the instance data in this example is directly embedded in the model file (specifically in the Coefficients objects), there is no need for the input and output data models nor for the data object, all of which are empty. The two variables are defined in the Variables objects. Notice that their indexes are specified by the key word “Self”; this term denotes that a variable is its own index, that is, it is a singleton. Similarly, the three constraints are defined in the Constraints objects (even the bound, which could have been included in the Bounds field of the corresponding variable object). Finally, the mapping of constraints (i.e. rows) and variables (i.e. columns) to entries in the matrix is defined in the Coefficients objects. As we shall see below, in general each Coefficients object represents a block in the matrix, but in this simple example, each block is simply a single coefficient.

1. **Generic Constraint**

More complex models, such as the examples that follow, need to represent a diversity of constraint relationships among the decision variables. MOSDEX uses a template for those relationships called the *generic constraint*. The generic constraint must be configured through data alone, in order to eliminate creating objects that require customized parsing and compilation. The general form of the generic constraint is

forall (c in Constraints)

forall (i in c.Index])

sum(v in Variables) (

sum(j in v.Index, a in Coefficients[c,v] where condition(i, j, a) is true)

a.value \* x[j] <= i.RHS)

The essence of the generic constraint is the definition of the boolean condition function. To avoid customized parsing and compilation in MOSDEX, the condition function is limited to certain specific types of terms. Most commonly, it is a matching condition of the form

a matches i & a matches j

where the boolean function *matches* is defined as follows:

*a matches i* evaluates as **true** if the tuples *a* and *i* have at least one key field in common and the corresponding values of the common fields are equal in both tuples, and otherwise it evaluates as **false**.

Examples below will indicate other potential types of conditions that can be represented.

1. **The Warehousing Example**

The warehousing example, extensively discussed in <https://github.com/JeremyBloom/Optimization---Sample-Notebooks/blob/master/Optimization%2BModeling%2Band%2BRelational%2BData%2Bpub.ipynb>, illustrates most of the key features of the MOSDEX syntax.

The input and output data model objects, called collectors, each consist of a sequence of table schema objects, each of which is an ordered set of fields consisting of a name and a data type. In MOSDEX, only three primitive data types are used: String, Integer (including binary), or Double (a floating point number); objects or arrays, composite data types, are not allowed. A table’s key, the set of one or more fields that uniquely defines each row in the table, are denoted by the asterisk (\*) at the beginning of the name, which should be stripped by the parser.

The Data itself is represented as a collector object consisting of a sequence of table objects, each with its own schema as defined in the input data model. The tables in turn are represented as arrays of tuple objects. Tuples are represented as sequences of named data items as specified by the schema. The tables in this example have been abbreviated by using the ellipsis (…) as a comment (comments are officially outlawed in JSON, although many parsers allow them either as balance of a line (//) or embedded (/\*…\*/)). In practice, the data would be provided as a separate file, due to its length and to adherence to model/data separation.

Next, the Variables array consists of a list of variable objects, each specifying the name, type, index and bounds of a variable. The index is a reference to a table, whose key uniquely identifies the variable; that is, a variable declaration actually compactly represents a whole set of individual decision variables. The dimension of the set is determined by the number of fields in the table’s key; for instance, the open variable has one dimension while routes has two. The variable’s bounds are represented as a two-item object; while in this example, the bounds are numeric literals, they could also refer to fields in a tuple with a key matching the variable index. Alternatively, the bounds can be specified by a predefined key word such as “Non-negative”.

Next, the Constraints array is defined analogously. Each constraint object specifies the name, type, index, sense, and right-hand side of a constraint. The index operates analogously as in a variable object. The sense represents the direction of the inequality between the left-hand side expression and the constant on the right-hand side (RHS). Expressions involving decision variables are not allowed in MOSDEX, nor are constants allowed in left-hand side expressions. The right-hand side may either be a numeric literal or a reference to a tuple field.

Next, the Decision\_Expressions array is defined in the same manner as the Constraints. The decision expressions represent components of the objective function. In general, the sense field should be “None” except for the actual objective, which can have a “Maximize” or “Minimize” sense. Note that a general decision expression can have an index, but the actual objective is a singleton with a “Self” index. Decision expressions might also be equivalently defined as constraints with equality as the sense and a variable to capture its computed value; then the objective would be defined as minimizing or maximizing one of those variables. We have chosen not to go this route so as to not overly constrain model formulation, but it may actually simplify MOSDEX to use it.

The final array is Coefficients, comprising the objects that represent blocks of coefficients in a matrix representation of the optimization problem. Each coefficients object represents the intersection of a set of constraints with a set of variables. The entries field represents that data that fills the block, which may be a numeric literal or a tuple field reference. The coefficients object implicitly imposes the matching condition of the generic constraint

a matches i & a matches j

where *i* is the index of the constraints, *j* is the index of thevariables, and *a* is the tuple of coefficient data which includes keys matching the constraint and variable indices. Alternative, MOSDEX could make the condition explicit, which would require, at a minimum, the ability to parse a condition expression.

We choose to separate the coefficients objects from the corresponding constraints/decision expressions and variables. This separation enables MOSDEX to avoid the row-orientation vs. column-orientation dilemma of many other modeling APIs and languages – MOSDEX handles both automatically. As an alternative, we could have embedded to coefficients objects within the constraints and variables. This approach may have merit if MOSDEX were to enable specifying particular types of constraints, such as flow-balances in network models.

1. **The Net1 Example**

The Net1 example (<https://ampl.com/BOOK/EXAMPLES/EXAMPLES2/net1.mod>) is a small instance of a network flow model, which occur extremely frequently in practice. The previous warehousing example also has a network component, but there the network is bipartite with no transshipment nodes, also a frequently occurring special case. It was chosen to illustrate how MOSDEX can handle general network models.

The definitions of the various modeling objects are very similar to the previous example. Thus, we focus on the extensions needed in this example to represent the basic flow balance constraints at all nodes. There are two issues.

First, the formulation does not distinguish among the node types; all that is known is the supply or demand at each node. Originally, the data model had two fields in the cities table for the supply and demand at each city (at most one of which could be non-zero). That formulation would have required an arithmetic data transformation in the RHS of the balance constraint:

{  
 **"Name"**: **"balance"**,  
 **"Type"**: **"Linear"**,  
 **"Index"**: **"cities"**,  
 **"Sense"**: **"=="**,  
 **"RHS"**: **"cities.supply - cities.demand"**}

However, in principle, we believe that data transformations, no matter how trivial, should be performed before the data is presented to the modeling layer of the application; that function should not be part of MOSDEX. (Furthermore, supporting arithmetic transformations would require a parser for such expressions, which would complicate MOSDEX parsing). Therefore, the data model and data itself were revised to specify only a single field called “netSupply”. Thus, partitioning the cities into source, transshipment and destination nodes would require determining the sign of the netSupply parameter, which is not an operation within the scope of MOSDEX. As a result, the balance constraint has to be written in a way that applies to all cities regardless of node type. (This issue does not usually arise in bipartite networks, such as in the warehousing example, where separate flow balance constraints are written for the sources and destinations, e.g. ctSupply and ctDemand in the warehousing model).

As for the second issue, the links data table has two fields representing nodes, “origin” and “destination” that each need to be matched to a “city”. That fact requires more than the simple matching condition of the generic constraint, which relies on having identical names for the matching fields. To handle this issue, MOSDEX permits attaching a “alias” to a field name, so that for matching purposes, the alias can match a field in another index. This feature is illustrated in the coefficients objects for the balance constraint, balance\_shipFrom and balance\_shipTo; for instance

{  
 **"Name"**: **"balance\_shipFrom"**,  
 **"Constraints"**: **"balance"**,  
 **"Variables"**: {**"Name"**:**"ship"**, **"Index"**: **"links.origin(city)"**},  
 **"Entries"**: 1.0  
}

Since network models with flow balance constraints are very common, it might be worthwhile to predefine “Flow\_Balance” as a constraint type in MOSDEX, as is done for example in AMPL, and to automatically generate the required constraints and coefficients objects.

1. **The Sailco Example**

The Sailco example (<https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.1/ilog.odms.ide.help/OPL_Studio/opllanguser/topics/opl_languser_app_areas_pwl_inventory.html>) illustrates another feature commonly found in practice, inventory tracking. The chief extension of MOSDEX needed for handling this feature is the introduction of a time dimension and the need to represent time lags among the variables. Since most of the MOSDEX syntax has already been explained in detail, we focus on these extensions. There are several issues.

First, as an index, time has an explicit order. To recognize that order, the key field designator in the data model changes from “\*” to ”+”. In this example, the periods are explicitly input as a table called “periods” which is computed prior to presentation to the modeling layer. However, creating a sequence of time periods is such a common occurrence that MOSDEX might be extended to compute them, as is done in the OPL code from which this example is taken.

The second issue is how to represent the lagged variable inventory in the balance constraint, ctBoat\_inventory. In this case, we created a coefficients block for the lagged variable, ctBoat\_lagged\_inventory, and designated the “Index”: “periods.period-1”.

{  
 **"Name"**: **"ctBoat\_inventory"**,  
 **"Constraints"**: **"ctBoat"**,  
 **"Variables"**: **"inventory"**,  
 **"Entries"**: 1.0  
},  
{  
 **"Name"**: **"ctBoat\_lagged\_inventory"**,  
 **"Constraints"**: {**"Name"**: **"ctBoat"**, **"Index"**: **"periods.period > 1"**},  
 **"Variables"**: {**"Name"**:**"inventory"**, **"Index"**: **"periods.period-1"**},  
 **"Entries"**: -1.0  
}

The MOSDEX parser would need to be extended to allow for this syntax. Note the implicit assumption that periods is a sequence of integers. While that is commonly the case, there is no reason why the sequence cannot have another domain. A more general form of the lagged index could be written as something like “periods.Previous(period)”.

The final issue is what to do about the initial inventory. Whereas the end-of-period inventories are decision variables, the first period’s starting inventory is a constant (parameters. initialInventory). Thus the inventory balance constraint has a different form in period 1 than in subsequent periods. We created two Constraints objects for the balance constraint, ctBoat, one of which applies to the first period and the other applies to the subsequent periods.

{  
 **"Name"**: **"ctBoat"**,  
 **"Type"**: **"Linear"**,  
 **"Index"**: **"periods.period >1"**,  
 **"Sense"**: **"=="**,  
 **"RHS"**: **"demands.demand"**},  
{  
 **"Name"**: **"ctBoat"**,  
 **"Type"**: **"Linear"**,  
 **"Index"**: **"periods.period ==1"**,  
 **"Sense"**: **"=="**,  
 **"RHS"**: **"demands.demand + parameters.initialInventory"**}

Note how the right-hand side in the first period adds the initial inventory to the demand, a legitimate computation within the modeling layer. Also note that the lagged coefficient block ctBoat\_lagged\_inventory applies only to the ctBoat constraint for the periods after the first.

The OPL model from which this example was derived took a different approach to handling the starting inventory – it extended the index for the inventory variable to a period 0 and used a constraint to peg its value to the initial value. While that approach is also valid, it requires a different indexing scheme from the other variables, which would complicate the MOSDEX representation. Nevertheless, MOSDEX could accommodate this approach also.

1. **The Warehousing Stochastic Model Example**

This example is a variation of the Warehousing example above and is discussed in <https://github.com/JeremyBloom/Optimization---Sample-Notebooks/blob/master/Locating%2BWarehouses%2Bto%2BMinimize%2BCosts%2BCase%2B2%2Bpub%2B3.ipynb>. It is the deterministic equivalent of a two-stage stochastic program with recourse. Interestingly, it appears to require no syntactic extensions of MOSDEX.