MathOptInterface: a data structure for mathematical optimization problems

Benoît Legat
ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium, benoit.legat@uclouvain.be

Oscar Dowson
Department of Industrial Engineering and Management Sciences at Northwestern University, Evanston, IL, oscar.dowson@northwestern.edu

Joaquim Dias Garcia
PSR Inc & PUC-Rio, Rio de Janeiro, Brazil, joaquimgarcia@psr-inc.com

Miles Lubin
Google Research, New York, NY, mlubin@google.com

JuMP is an open-source algebraic modeling language in the Julia language. In this work, we discuss a complete re-write of JuMP based on a novel abstract data structure, which we call MathOptInterface, for representing instances of mathematical optimization problems. MathOptInterface is significantly more general than existing data structures in the literature, encompassing, for example, a spectrum of problems classes from integer programming with indicator constraints to bilinear semidefinite programming. We highlight the challenges that arise from this generality, and how we overcame them in the re-write of JuMP.

Key words: problem formats; Julia; JuMP; JSON

1. Introduction

JuMP (Lubin and Dunning 2015, Dunning et al. 2017) is an algebraic modeling language for mathematical optimization written in the Julia language (Bezanson et al. 2017). JuMP, like other algebraic modeling languages (e.g., AMPL (Fourer et al. 1990), Convex.jl (Udell et al. 2014), CVX (Grant and Boyd 2014), CVXPY (Diamond and Boyd 2016), GAMS (Brook et al. 1988), Pyomo (Hart et al. 2017), and YALMIP (Löfberg 2004)), has an appearingly simple job: it takes a mathematical optimization problem written by a user, converts it into a standard form, passes that standard form to a solver, waits for the solver to complete, then queries the solver for a solution and returns the solution to the user.

At the heart of this process is the definition of the standard form. By standard form, we mean a concrete data structure, specified either by an in-memory API or via a file format, of a mathematical optimization problem that the user and solver agree upon so that they
can communicate. For example, based on the textbook presentation of linear programming (LP), one might assume that the following is a standard form accepted by solvers:

\[
\begin{align*}
\min_{x \in \mathbb{R}^N} & \quad c^T x \\
\text{subject to:} & \quad Ax = b \\
& \quad x \geq 0,
\end{align*}
\]

(1)

where \(c\) is a dense \(N\)-dimensional vector, \(b\) a dense \(M\)-dimensional vector, and \(A\) a sparse \(M \times N\) matrix.

Anyone who has interacted directly with LP solvers would know this is far from accurate. Some solvers allow linear constraints to have lower and upper bounds, so the user must pass \(l \leq Ax \leq u\). Other solvers allow only one bound per row of \(A\), but the user must also pass a vector of constraint senses (i.e., =, \(\leq\), or \(\geq\)), representing the problem constraints \(Ax \rhd b\), where \(\rhd\) is the vector of constraint senses. Differences in these formulations also flow through to solutions, where, given an affine constraint \(l \leq a^T x \leq u\), some solvers may return two dual variables—one for each side of the constraint—whereas other solvers may return one dual variable corresponding to the active side of the constraint. In addition, some solvers may support variable bounds, whereas others based on conic methods may require variables to be non-negative.

Moreover, as mathematical optimization has matured, research has focused on formulating and solving new types of optimization problems. Each time a new type of objective function or constraint has been added to the modeler’s toolbox, the standard form necessarily has had to change. For example, the commercial solvers MOSEK (MOSEK ApS 2019) and Gurobi (Gurobi Optimization 2019) have both developed independent (and incompatible) extensions to the MPS file format (IBM World Trade Corporation 1976) to support quadratic objectives and constraints. In our opinion, this has led to a fracturing of the optimization community, as each sub-community developed a different standard form and solver for the problems of their interest. For example, nonlinear programming solvers often require the standard form:

\[
\begin{align*}
\min_{x \in \mathbb{R}^N} & \quad f(x) \\
\text{subject to:} & \quad g(x) \leq 0 \\
& \quad h(x) = 0,
\end{align*}
\]

1 For this discussion, consider any standard sparse storage format, e.g., compressed sparse column or coordinate-list format.
where \( f : \mathbb{R}^N \mapsto \mathbb{R} \), \( g : \mathbb{R}^N \mapsto \mathbb{R}^G \), and \( h : \mathbb{R}^N \mapsto \mathbb{R}^H \) (and their respective derivatives) are specified via callbacks. In another sub-community, semidefinite solvers require the standard form:

\[
\min_{X \in \mathbb{R}^N \times \mathbb{R}^N} \langle C, X \rangle
\]

subject to: \( \langle A_i, X \rangle = b_i, \ i = 1, 2, \ldots, M \)

\[X \succeq 0,\]

where \( C \) and \( A_i \) are \( N \times N \) matrices, \( b_i \) is a constant scalar, \( \langle \cdot, \cdot \rangle \) denotes the inner product, and \( X \succeq 0 \) enforces the matrix \( X \) to be positive semidefinite.

Even within communities, there can be equivalent formulations. For example, in conic optimization, solvers such as CSDP (Borchers 1999) accept what we term the \textit{standard conic} form:

\[
\min_{x \in \mathbb{R}^N} c^\top x
\]

subject to: \( Ax = b \)

\[x \in \mathcal{K},\]  

(2)

whereas others solvers, such as SCS (O’Donoghue et al. 2016), accept what we term the \textit{geometric conic} form:

\[
\min_{x \in \mathbb{R}^N} c^\top x
\]

subject to: \( Ax + b \in \mathcal{K} \)

\[x \text{ free.}\]  

(3)

Here, \( c \) is a \( N \)-dimensional vector, \( A \) is an \( M \times N \) matrix, \( b \) is an \( M \)-dimensional vector, and \( \mathcal{K} \subseteq \mathbb{R}^N \) (\( \mathcal{K} \subseteq \mathbb{R}^M \) for the geometric conic form) is a convex cone from some pre-defined list of supported cones.

The variation in standard forms accepted by solvers makes writing a generalized algebraic modeling language such as JuMP difficult. Our main contribution of this paper is to define a new abstract data structure\(^2\) for representing mathematical optimization problems that generalizes the real-world diversity of the forms expected by solvers. We call this representation \textit{MathOptInterface}. As a second contribution, we explain the challenges that arose (and how we overcame them) when implementing our ideas in the Julia package \textit{MathOptInterface}. As a third contribution, we introduce another concrete implementation of MathOptInterface, this time in the form of a file format called MathOptFormat.

\(^2\)By \textit{abstract} data structure, we mean that we omit discussions of details like which storage format to use for sparse matrices and how an API should be implemented in a specific programming language.
It is important to note that this paper deals with both the abstract idea of the MathOpt-Interface standard form, and an implementation of this idea in Julia. To clearly distinguish between the two, we will always refer to the Julia constructs in typewriter font. Readers should note however, that our standard form is not restricted to the Julia language. Instead, it is intended to be a generic framework for thinking and reasoning about mathematical optimization. It is possible to write implementations in other languages such as Python; however, we chose Julia because JuMP is in Julia. Incidentally, the features of Julia make it well suited for implementing MathOptInterface in a performant way.

The rest of this paper is laid out as follows. In Section 2, we review the approaches taken by other algebraic modeling languages. Then, in Section 3, we introduce the MathOptInterface standard form, which is the main contribution of this paper. In Section 4, we highlight the cost of the generality arising from our choice of standard form and how we overcame this problem using bridges. In Section 5, we present a new file format—called MathOptFormat—for mathematical optimization that is based on MathOptInterface. Finally, in Section 6, we describe how the introduction of MathOptInterface has influenced JuMP.

2. Literature review

In this section, we review how existing modeling packages manage the conflict between the models provided by users and the standard forms expected by solvers.

2.1. A history of modeling packages

Orchard-Hays (1984) provides a detailed early history of the inter-relationship between computing and optimization, beginning with the introduction of the simplex algorithm in 1947, through to the emergence of microcomputers in the early 1980’s. Much of this early history was dominated by a punch-card input format called MPS (IBM World Trade Corporation 1976), which users created using problem-specific computer programs called matrix generators.

However, as models kept getting larger, issues with matrix generators began to arise. Fourer (1983) argues that the main issues were: (i) a lack of verifiability, which meant that bugs would creep into the matrix generating code; and (ii) a lack of documentation, which meant that it was often hard to discern what algebraic model the matrix generator actually produced. Instead of matrix generators, Fourer (1983) advocated the adoption of algebraic modeling languages. Algebraic modeling languages can be thought of as advanced
matrix generators which build the model by parsing an algebraic model written in a human-readable domain-specific language. Two examples of early modeling languages that are still in wide-spread use are AMPL (Fourer et al. 1990) and GAMS (Brook et al. 1988).

Modeling languages allowed users to construct larger and more complicated models. In addition, they were extended to support the ability to model different types of programs, e.g., nonlinear programs, and specific constraints such as complementarity constraints. Because of the nonlinear constraints, modeling languages such as AMPL and GAMS represent constraints as expression graphs. They are able to communicate with solvers through the .nl file-format (Gay 2005), or through modeling-language specific interfaces such as the AMPL Solver Library (Gay 1997).

More recently, the last 15 years has seen the creation of modeling languages embedded in high-level programming languages. Examples include CVX (Grant and Boyd 2014) and YALMIP (Löfberg 2004) in MATLAB™, CVXPY (Diamond and Boyd 2016) and Pyomo (Hart et al. 2017) in Python, and Convex.jl (Udell et al. 2014) and JuMP in Julia.

Like AMPL and GAMS, Pyomo represents models using expression graphs and interfaces with solvers either through files (e.g., MPS and .nl files), or, for a small number of solvers, via direct in-memory interfaces which convert the expression graph into each solver’s specific standard form.

YALMIP is a MATLAB™-based modeling language for mixed-integer conic and non-linear programming. It also has support for other problem classes, including geometric programs, parametric programs, and robust optimization. Internally, YALMIP represents conic programs in the geometric form (3). Because of this design decision, if the user provides a model that is close to the standard conic form (2), YALMIP must convert the problem back to the geometric form, introducing additional slack variables and constraints. This can lead to sub-optimal formulations being passed to solvers. As a work-around for this issue, YALMIP provides functionality for automatic dualizing conic models (Löfberg 2009). Solving the dual instead of the primal can lead to significantly better performance in some cases; however, the choice of when to dualize the model is left to the user.

Since YALMIP represents the problem in geometric form and has no interface allowing users to specify the cones the variables belong to, when dualizing it needs to reinterpret the affine constraints when the affine expression contains only one variable as a specification of the cone for the variable. Given that there is no unique interpretation of the dual form if
a variable is interpreted to belong to two or more cones, in such cases YALMIP considers
one of the constraints as a variable constraint and the others as affine constraints. It uses
the following heuristic if the constraints are on different cones: “Notice that it is important
to detect the primal cone variables in a certain order, starting with SDP cones, then SOCP
cones, and finally LP cones” (Löfberg 2009).

The rigidity of a standard form chosen by a modeling language such as YALMIP also
limits the structure that the user is able to transmit to the solver. The resulting transforma-
tions needed to make a problem fit in the standard form can have significant impacts on the
runtime performance of the solver. For example, in addition to (2), SDPNAL+ (Sun et al.
2019) supports adding bounds on the variables and adding affine interval constraints. Forc-
ing the problem to fit in the standard form (2) requires the addition of slack variables
and equality constraints that have a negative impact on the performance of SDPNAL+ as
described in Sun et al. (2019):

The final number of equality constraints present in the data input to SDPNAL+ can
also be substantially fewer than those present in CVX or YALMIP. It is important
to note here that the number of equality constraints present in the generated prob-
lem data can greatly affect the computational efficiency of the solvers, especially for
interior-point based solvers.

CVXPY (Diamond and Boyd 2016) is a modeling language for convex optimization in
Python. A notable feature of CVXPY is that it is based on disciplined convex programming
(Grant et al. 2006); this is a key difference from many other modeling languages, including
JuMP. The atomic rules of disciplined convex programming mean that the convexity of
a user-provided function can be inferred at construction time in an axiomatic way. This
has numerous benefits for computation, but restricts the user to formulating models with
a reduced set of operators (called atoms), for which the convexity of the atom is known.

One feature of CVXPY that the re-write of JuMP does inherit is the concept of a
reduction. A reduction is a transformation of one problem into an equivalent form. Reduc-
tions allow CVXPY to re-write models formulated by the user into equivalent models
that solvers accept (Agrawal et al. 2018). Examples of reductions implemented in CVXPY
include Complex2Real, which lifts complex-valued variables into the real domain by intro-
ducing variables for the real and imaginary terms, and FlipObjective, which converts
a minimization objective into a maximization objective, and vice versa. Reductions can
be chained together to form a sequence of reductions. CVXPY uses pre-defined chains of reductions to convert problems from the disciplined convex programming form given by the user into a standard form required by a solver. This is similar to the way that MathProgBase has transformations between different problem classes. In Section 4, we show how MathOptInterface extends the idea of applying a fixed set of reductions into dynamically deciding which reductions to perform. Note that JuMP’s reductions, called bridges, apply only to classes of constraints, variables, and objective functions, rather than applying global transformations as CVXPY does.

2.2. The previous design of JuMP

The first version of JuMP, described by Lubin and Dunning (2015) and Dunning et al. (2017), featured an ad-hoc collection of three different standard forms: (i) a linear-quadratic standard form for specifying problems with linear, quadratic, and integrality constraints; (ii) a conic standard form for specifying problems with linear, conic, and integrality constraints; and (iii) a nonlinear standard form for specifying problems with nonlinear constraints. (The ad-hoc nature of JuMP’s initial development is reflected in the paper of Lubin and Dunning (2015), which states that “we did not set out to design a full modeling language with as wide a variety of options as AMPL.”)

In code, the three standard forms were implemented in an intermediate layer called MathProgBase. As the first step, JuMP converted the problem given by the user into one of the three MathProgBase standard forms. Underneath MathProgBase, each solver required a thin layer of Julia code (called wrappers) that connected the solver’s native interface (typically written in C) to one of the three standard forms, e.g., Clp to linear-quadratic, Ipopt to nonlinear, and SCS to conic. There were also automatic converters between the standard forms, including linear-quadratic to conic, conic to linear-quadratic, and linear-quadratic to nonlinear. This enabled, for example, the user to formulate a linear program and solve it with Ipopt (a nonlinear programming solver). Figure 1 visualizes the architecture of JuMP just described.

The design of MathProgBase made it impossible to flexibly combine different standard forms. For example, JuMP could not communicate a program with both nonlinear and conic constraints to a solver. Because of this inflexibility, along with other design decisions (e.g., deleting variables was not supported, and it was not possible to extend the standard form), the developers decided in mid-2017 to remove MathProgBase and re-write JuMP.
based on a new, all-encompassing standard form and interface layer, implemented in the MathOptInterface package.

2.3. A history of file formats

In Section 5, we introduce a new file format for mathematical optimization. Creating a new file format is a large undertaking, and it was not a decision that we made lightly. Our main motivation was to create a way to serialize MathOptInterface problems to disk. However, to fully understand our justification for creating a new format, it is necessary to give a brief history of the evolution of file formats in mathematical optimization.

As we have outlined, in order to use an optimization solver, it is necessary to communicate a model instance to the solver. This can either be done through an in-memory interface, or through a file written to disk. File formats are also used to collate models into instance libraries for benchmarking purposes, e.g., CBLIB (Friberg 2016) and MIPLIB (Zuse Institute Berlin 2018).

Many different instance formats have been proposed over the years, but only a few (such as MPS (IBM World Trade Corporation 1976)) have become the industry standard. Each format is a product of its time in history and the problem class it tried to address. For example, we retain the rigid input format of the MPS file that was designed for 1960s punch-cards despite the obsolescence of this technology (Orchard-Hays 1984). Although the MPS format has since been extended to problem classes such as nonlinear and stochastic linear programming, MPS was not designed with extensibility in mind. This has led some authors (e.g., Friberg (2016)) to conclude that developing a new format is easier than extending the existing MPS format.
The LP file-format is an alternative to the MPS file-format that is human-readable and row-oriented (LP-solve 2016). However, there is no longer a single standard for the LP file-format. This has led to subtle differences between implementations in different readers that hampers the usefulness of the format as a medium for interchange. Much like the MPS file, the LP file is also limited in the types of problems it can represent and was not designed for extensibility.

In contrast to the LP file, the .nl file (Gay 2005) explicitly aims for machine-readability at the expense of human-readability. It is also considerably more flexible in the problem classes it can represent (in particular, nonlinear functions are supported). However, once again, the format is not extensible to new problem formats and has limited support for conic problems.

More recently, considerable work has been put into developing the OSiL format (Fourer et al. 2010). In developing OSiL, Fourer et al. identified many of the challenges and limitations of previous formats and attempted to overcome them. In particular, they choose to use XML as the basis for their format. This removed the burden of writing custom readers and writers for each programming language that wished to interface with optimization software and allowed more focus on the underlying data structures. XML is also human-readable and can be rigidly specified with a schema to prevent the proliferation of similar, but incompatible versions. The XML approach also allows for easy extensibility and can support multiple problem classes including nonlinear, stochastic, and conic.

However, despite the many apparent advantages of the OSiL format, we believe it has enough short-comings to justify the development of a new instance format. The main reason is the lack of a strong, extensible standard form. A secondary reason is the waning popularity of XML in favor of simpler formats such as JSON.

In summary, software for formulating mathematical optimization programs has matured considerably since the days of hand-constructed MPS files on punch-cards. However, the tyranny of solvers requiring a multitude of different standard forms has not been overcome. At present, almost all modeling languages use a variation of the structure in Figure 1 with a small set of standard forms (or file formats) and some direct solver interfaces. In the next two sections, we outline a novel standard form for optimization and a generalization of CVXPY’s reduction system. This will lead us, in Section 6, to a new architecture for the interface between modeling languages and solvers.
3. MathOptInterface

MathOptInterface is an abstract specification for a data structure for mathematical optimization problems. It represents problems in the following form:

\[
\begin{align*}
\min_{x \in \mathbb{R}^N} & \quad f_0(x) \\
\text{subject to:} & \quad f_i(x) \in S_i, \quad i = 1, 2, \ldots, I,
\end{align*}
\]

with functions \( f_i : \mathbb{R}^N \to \mathbb{R}^{M_i} \) and sets \( S_i \subset \mathbb{R}^{M_i} \) drawn from a pre-defined set of functions \( \mathcal{F} \) and sets \( \mathcal{S} \).

The sets \( \mathcal{F} \) and \( \mathcal{S} \) are provided in Section 3.1 (for \( \mathcal{F} \)) and Appendix A (for \( \mathcal{S} \)). In addition, we provide a concrete description in the form of a JSON schema as part of the MathOptFormat file format described in Section 5. In the JuMP ecosystem, the definitions of supported functions and sets are contained in the Julia package MathOptInterface. JuMP additionally allows third-party packages to extend the set of recognized functions and sets at runtime, but this is not required for an implementation of MathOptInterface; for example, MathOptFormat does not allow extensions.

Since constraints are formed by the combination of a function and a set, we will often refer to constraints by their function-in-set pairs. The key insight is the ability to mix-and-match a small number of pre-defined functions and sets to create a wide variety of different problem classes.

We believe model (4) is very general, and encompasses almost all of existing deterministic mathematical optimization with real-valued variables. (An extension to complex numbers could be achieved by replacing \( \mathbb{R} \) with \( \mathbb{C} \).) Readers should note that when the objective is vector-valued, the objective vectors are implicitly ranked according to partial ordering such that if \( y_1 = f_0(x_1) \) and \( y_2 = f_0(x_2) \), then \( y_1 \preceq y_2 \iff y_2 - y_1 \in \mathbb{R}^{M_0}_{++} \). In the future, we plan to extend MathOptInterface to model general vector-valued programs, which define the partial ordering in terms of a convex cone \( \mathcal{C} \) (see, e.g., Löhne (2011)). However, we omit a full description of this extension because we do not have a practical implementation.

3.1. Functions

MathOptInterface defines the following functions in the set \( \mathcal{F} \):

- The SingleVariable function \( f(x) = e_i^\top x \), where \( e_i \) is an \( N \)-dimensional vector of zeros with a 1 in the \( i^{th} \) element.
• The **VectorOfVariables** function \(f(x) = [x_{i_1}, x_{i_2}, \ldots, x_{i_M}]\), where \(i_j \in \{1, 2, \ldots, N\}\) for all \(j \in 1, \ldots, M\).

• The **ScalarAffineFunction** \(f(x) = a^\top x + b\), where \(a\) is a sparse \(N\)-dimensional vector and \(b\) is a scalar constant.

• The **VectorAffineFunction** \(f(x) = A^\top x + b\), where \(A\) is a sparse \(M \times N\) matrix and \(b\) is a dense \(M\)-dimensional vector.

• The **ScalarQuadraticFunction** \(f(x) = \frac{1}{2}x^\top Qx + a^\top x + b\), where \(Q\) is a sparse \(N \times N\) matrix, \(a\) is a sparse \(N\)-dimensional vector, and \(b\) is a scalar constant.

• The **VectorQuadraticFunction** \(f(x) = [x^\top Q_1 x, \ldots, x^\top Q_i x, \ldots, x^\top Q_M x]^\top + A^\top x + b\), where \(Q_i\) is a sparse \(N \times N\) matrix for \(i = 1, 2, \ldots, M\), \(A\) is a sparse \(M \times N\) matrix, and \(b\) is a dense \(M\)-dimensional vector.

Notably missing from this list is **ScalarNonlinearFunction** and **VectorNonlinearFunction**.

At present, **MathOptInterface** defines (for historical reasons) a separate mechanism for declaring a block of nonlinear constraints \(l \leq g(x) \leq u\) and/or a nonlinear objective \(f(x)\).

However, an upcoming goal is a complete re-write of JuMP’s nonlinear support to bring first-class support for nonlinear functions to **MathOptInterface**. The lack of a current implementation should not distract from the conceptual ability of MathOptInterface to support scalar- and vector-valued nonlinear functions in the future.

Moreover, note that many of the function definitions are redundant, e.g., a **ScalarAffineFunction** is a **VectorAffineFunction** where \(M = 1\). The reason for this redundancy will become clear in Section 4.

### 3.2. Sets

The set of sets supported by MathOptInterface, \(\mathcal{S}\), contains a large number of elements. The complete list is given in Appendix A and by the JSON schema described in Section 5 so we only present some of the more common sets that will later be referenced in this paper:

• The **LessThan** set \((-\infty, u]\) where \(u \in \mathbb{R}\)

• The **GreaterThan** set \([l, \infty)\) where \(l \in \mathbb{R}\)

• The **Interval** set \([l, u]\), where \(l \in \mathbb{R}\) and \(u \in \mathbb{R}\)

• The **Integer** set \(\mathbb{Z}\)

• The **Nonnegatives** set \(\{x \in \mathbb{R}^N : x \geq 0\}\)

• The **Zeros** set \(\{0\} \subset \mathbb{R}^M\)
• The SecondOrderCone set \( \{(t, x) \in \mathbb{R}^{1+N} : ||x||_2 \leq t\} \)

• The RotatedSecondOrderCone set \( \{(t, u, x) \in \mathbb{R}^{2+N} : ||x||_2^2 \leq 2tu, t \geq 0, u \geq 0\} \)

MathOptInterface also defines sets like the positive semidefinite cone, and even sets that are not cones or standard sets like Interval and Integer. For example, MathOptInterface defines the SOS1 and SOS2 sets, which are special ordered sets of Type I and Type II respectively (Beale and Tomlin 1970). In addition, it also defines the Complements set, which can be used to specify mixed complementarity constraints (Dirkse and Ferris 1995). See Appendix A for more details.

To demonstrate how these functions and sets can be combined to create mathematical programs, we now consider a number of examples.

3.3. Example: linear programming

Linear programs are often given in the form:

\[
\min_{x \in \mathbb{R}^N} \ c^\top x + 0 \\
\text{subject to: } A x \geq b,
\]

where \( c \) is a vector with \( N \) elements, \( A \) is an \( M \times N \) matrix, and \( b \) is a vector with \( M \) elements.

In the MathOptInterface standard form, \( f_0(x) \) is the ScalarAffineFunction \( f_0(x) = c^\top x + 0 \), \( f_1(x) \) is the VectorAffineFunction \( f_1(x) = Ax - b \), and \( S_1 \) is the \( M \)-dimensional Nonnegatives set.

3.4. Example: multi-objective problems with conic constraints

Because of its generality, MathOptInterface is able to represent problems that do not neatly fit into typical standard forms. For example, here is a multi-objective mathematical program with a second-order cone constraint:

\[
\min_{x \in \mathbb{R}^N} \ C x + 0 \\
\text{subject to: } A x = b \\
\left\| x_2, x_3, \ldots, x_N \right\|_2 \leq x_1 \\
l_i \leq x_i \leq u_i, i = 1, 2, \ldots, N,
\]

where \( C \) is an \( P \times N \) matrix, \( A \) is an \( M \times N \) matrix, \( b \) is an \( M \)-dimensional vector, and \( l_i \) and \( u_i \) are constant scalars.
In the MathOptInterface standard form, \( f_0(x) \) is the VectorAffineFunction \( f_0(x) = Cx + 0 \), \( f_1(x) \) is the VectorAffineFunction \( f_1(x) = Ax - b \), \( S_1 \) is the \( M \)-dimensional Zeros set, \( f_2(x) \) is the VectorOfVariables function \( f_2(x) = x \), \( S_2 \) is the SecondOrderCone set, functions \( f_{2+i}(x) \) are the SingleVariable functions \( f_{2+i}(x) = x_i \), and sets \( S_{2+i} \) are the Interval sets \([l_i, u_i]\).

### 3.5. Example: special ordered sets

Many mixed-integer solvers support a constraint called a special ordered set (Beale and Tomlin 1970). There are two types of special ordered sets. Special ordered sets of type I require that at most one variable in an ordered set of variables can be non-zero. Special ordered sets of type II require that at most two variables in an ordered set of variables can be non-zero, and, if two variables are non-zero, they must be adjacent in the ordering.

An example of a problem with a special ordered set of type II constraint is as follows:

\[
\begin{align*}
\min_{x \in \mathbb{R}^N} & \quad c^T x + 0 \\
\text{subject to:} & \quad Ax \geq b \\
& \quad [x_1, x_2, x_3] \in \text{SOS}_II([1, 3, 2]).
\end{align*}
\]

Here, the weights on the variables imply an ordering \( x_1, x_3, x_2 \).

In the MathOptInterface standard form, \( f_0(x) \) is the ScalarAffineFunction \( f_0(x) = c^T x + 0 \), \( f_1(x) \) is the VectorAffineFunction \( f_1(x) = Ax - b \), \( S_1 \) is the \( M \)-dimensional Nonnegatives set, function \( f_2(x) \) is the VectorOfVariables function \( f_2(x) = [x_1, x_2, x_3] \), and set \( S_2 \) is the SOS2 set \( \text{SOS2}([1, 3, 2]) \).

### 3.6. Example: mixed-complementarity problems

A mixed-complementarity problem, which can be solved by solvers such as PATH (Dirkse and Ferris 1995), can be defined as follows:

\[
\begin{align*}
\min_{x \in \mathbb{R}^N} & \quad 0 \\
\text{subject to:} & \quad Ax + b \perp x \\
& \quad l \leq x \leq u,
\end{align*}
\]

(5)

where \( A \) is an \( N \times N \) matrix, and \( b, l, \) and \( u \) are \( N \)-dimensional vectors. The constraint \( Ax + b \perp x \) requires that the following conditions hold in an optimal solution:

- if \( x = l \), then \( Ax + b \geq 0 \);
- if \( l < x < u \), then \( Ax + b = 0 \); and
• if \( x = u \), then \( Ax + b \leq 0 \).

Thus, we can represent model (5) in the MathOptInterface standard form as:

\[
\begin{align*}
\min_{x \in \mathbb{R}^N} & \quad 0 \\
\text{subject to: } & \quad \begin{bmatrix} A \\ I \end{bmatrix} x + \begin{bmatrix} b \\ 0 \end{bmatrix} \in \text{Complements}() \\
& \quad x_i \in [l_i, u_i], i = 1, 2, \ldots, N.
\end{align*}
\]  

(6)

Here, \( f_0(x) \) is the ScalarAffineFunction \( f_0(x) = 0 \), \( f_1(x) \) is the VectorAffineFunction \( f_1(x) = [A; I]x + [b; 0] \), \( S_1 \) is the \( N \)-dimensional Complements set, functions \( f_{1+i}(x) \) are the Single-Variable functions \( f_{1+i}(x) = x_i \), and sets \( S_{1+i} \) are the Interval sets \([l_i, u_i]\).

These examples demonstrate the flexible modeling power of the MathOptInterface standard form. In the next section, we describe a downside to this generality, and we explain how we overcame it with the concept of bridges.

4. Bridges

The generality of MathOptInterface poses a major challenge to both modelers and solvers because there are often multiple ways to formulate the same constraint using different combinations of functions and sets. For example, the constraint \( l \leq a^\top x \leq u \) can be formulated in many ways, three of which are listed here:

• using the original formulation: \( l \leq a^\top x \leq u \) (ScalarAffineFunction-in-Interval);
• by splitting the constraint into \( a^\top x \leq u \) (ScalarAffineFunction-in-LessThan) and \( a^\top x \geq l \) (ScalarAffineFunction-in-GreaterThan); or
• by introducing a slack variable \( y \), with constraints \( a^\top x - y = 0 \) (ScalarAffineFunction-in-EqualTo) and \( l \leq y \leq u \) (SingleVariable-in-Interval).

This generality means that solver authors need to decide which functions and sets to support, users need to decide how to formulate constraints, and modeling language developers need to decide how to translate constraints between the user and the solver.

One approach to this problem is to require every solver to implement an interface to every combination of function-in-set that the user could provide, and inside each solver transform the user-provided constraint into a form that the solver natively understands. However, as the number of functions and sets increases, this approach quickly becomes burdensome.
An alternative approach, and the one implemented in MathOptInterface, is to centralize the problem transformations into a collection of what we call bridges. A bridge is a thin layer that transforms a function-in-set pair into an equivalent list of function-in-set pairs. An example is the transformation of a ScalarAffineFunction-in-Interval constraint into a ScalarAffineFunction-in-LessThan and a ScalarAffineFunction-in-GreaterThan constraint. The bridge is also responsible for reversing the transform to provide information such as dual variables back to the user.

MathOptInterface defines a large number of bridges. For example, there are slack bridges, which convert inequality constraints like $a^\top x \geq b$ into equality constraints like $a^\top x - y = b$ by adding a slack variable $y \geq 0$. There are also bridges to convert between different cones. For example there is a bridge to convert a rotated second-order cone into a second-order cone using the following relationship:

$$2tu \geq ||x||^2 \iff (t/\sqrt{2} + u/\sqrt{2})^2 \geq ||x||^2 + (t/\sqrt{2} - u/\sqrt{2})^2.$$  

Bridges can also be nested to allow multiple transformations. For example, a solver that supports only ScalarQuadraticFunction-in-EqualTo constraints can support RotatedSecondOrderCone constraints via a transformation into a SecondOrderCone constraint, then into a ScalarQuadraticFunction-in-LessThan constraint, and then into a ScalarQuadraticFunction-in-EqualTo via a slack bridge.

The proliferation in the number of these bridges leads to a new challenge: there are now multiple ways of transforming one constraint into an equivalent set of constraints via chains of bridges. To demonstrate this, consider bridging a ScalarAffineFunction-in-Interval constraint into a form supported by a solver which supports only SingleVariable-in-GreaterThan and ScalarAffineFunction-in-EqualTo constraints. Two possible reformulations are given in Figure 2.

The first reformulation converts $l \leq a^\top x \leq u$ into $l \leq a^\top x$ and $a^\top x \leq u$ via the split-interval bridge, and then converts each inequality into a ScalarAffineFunction-in-EqualTo constraint and a SingleVariable-in-GreaterThan constraint using the slack bridge, which introduces an additional slack variable $y$. The second reformulation includes an additional step of converting the temporary constraint $a^\top x \leq u$ into $-u \leq -a^\top x$ via the flip-sign bridge. Notably, both reformulations add two ScalarAffineFunction-in-EqualTo constraints, two slack variables, and two SingleVariable-in-GreaterThan constraints, but the first reformulation is preferred because it has the least number of transformations.
4.1. Hyper-graphs and shortest paths

It is easy to see that as the number of constraint types and bridges increases, the number of different equivalent reformulations also increases, and choosing an appropriate reformulation becomes difficult. We overcome the proliferation challenge by posing the question of how to transform a constraint into a set of supported equivalents as a shortest path problem through a directed hyper-graph.

We define our directed hyper-graph \( G(N, E) \) by a set of nodes \( N \), containing one node \( n \) for each possible function-in-set pair, and a set of directed hyper-edges \( E \). Each directed hyper-edge \( e \in E \), corresponding to a bridge, is comprised of a source node \( s(e) \in N \) and a set of target nodes \( T(e) \subseteq N \). For each hyper-edge \( e \), we define a weight \( w(e) \). For simplicity, \texttt{MathOptInterface} chooses \( w(e) = 1 \) for all \( e \in E \), but this need not be the case. In addition, each solver defines a set of supported nodes \( S \). Finally, for each node \( n \in N \), we define a cost function, \( C(n) \), which represents the cost of bridging node \( n \) into an equivalent set of supported constraints:

\[
C(n) = \begin{cases} 
0 & n \in S \\
\min_{e \in E : s(e) = n} \{w(e) + \sum_{n' \in T(e)} C(n')\} & \text{otherwise.}
\end{cases}
\]

In the spirit of dynamic programming, if we can find the minimum cost \( C(n) \) for any node \( n \), we also obtain a corresponding hyper-edge \( e \). This repeats recursively until we reach a terminal node at which \( C(n) = 0 \), representing a constraint that the solver natively supports. The collection of edges associated with a solution is referred to as a hyper-path.

Problems of this form are well studied by Gallo et al. (1993), who propose an efficient algorithm for computing \( C(n) \) and obtaining the minimum cost edge \( e \) associated with

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{fig2.pdf}
\caption{Two equivalent solutions to the problem of bridging a ScalarAffineFunction-in-Interval constraint (oval nodes). Outlined rectangular nodes represent constraint actually added to the model. Nodes with no outline are intermediate nodes.}
\end{figure}
each node. Due to the large number of nodes in the hyper-graph, we do not precompute the shortest path for all nodes \textit{a priori}. Instead, we compute $C(n)$ in a \textit{just-in-time} fashion whenever the first constraint of type $n$ is added to the model. Because the computation is performed once per type of constraint, the decision is independent of any constraint data like coefficient values.

The choice of cost function has a significant impact both on the optimal solution and on the computational tractability of the problem. Indeed, if the cost function is chosen to be the number of different bridges used, the shortest path problem is NP-complete (Italiano and Nanni 1989). In the present case, if a bridge is used twice, it makes sense to include its weight twice as well. This cost function is part of the more general family of \textit{additive cost functions} for which the shortest hyper-path problem can be solved efficiently with a generalization of the Bellman-Ford or Dijkstra algorithms; see Gallo et al. (1993, Section 6) for more details.

4.2. Variable and objective bridges

In the interest of simplicity, we have described only \textit{constraint bridges}, in which the nodes in the hyper-graph correspond to function-in-set pairs. In practice, there are three types of nodes in $N$: \textit{constraint} nodes for each pair of function type $f$ and set type $S$ representing $f$-in-$S$ constraints; \textit{objective} nodes for each type $f$ representing an objective function of type $f$; and \textit{variable} nodes for each set type $S$ representing variables constrained to $S$. Hyper-edges beginning at a node $n$ can have target nodes of different types.

Objective nodes (and corresponding bridges) allow, for example, conic solvers that support only affine objectives to solve problems modeled with a quadratic objectives by replacing the objective with a slack variable $y$, and then adding a quadratic inequality constraint. If necessary, the quadratic inequality constraint may be further bridged to a second-order cone constraint.

Variable nodes correspond to a concept we call \textit{variables constrained on creation}, and they are needed due to differences in the way solvers handle variable initialization. A naïve way of creating variables is to first add $N$ variables to the model, and then add Single-Variable and VectorOfVariables constraints to constrain the domain. This approach works for many solvers, but fails in two common cases: (i) some solvers (e.g., CSDP (Borchers 1999)) do not support free variables; and (ii) some solvers (e.g., MOSEK (MOSEK ApS 2019)) have a special type of variable for PSD variables, and this must be specified at
creation time. For example, adding the constraint $X \succeq 0$ to MOSEK after $X$ has been created will result in a bridge that creates a new PSD matrix variable $Y \succeq 0$, and then a set of \texttt{ScalarAffineFunction-in-EqualTo} constraints such that $X = Y$.

Similar to constraints, solvers specify a set of supported variable sets (i.e., so $C(n) = 0$). For most solvers, the supported variable set is the singleton \texttt{Reals}. If the user attempts to add a variable constrained on creation to a set $S$ that is not supported, a bridge first adds a free variable ($x$-in-\texttt{Reals}) and then adds a \texttt{SingleVariable}-in-$S$ constraint. Thus, variable nodes allow solvers such as CSDP to declare that they support only $x$-in-\texttt{Nonnegatives} and not $x$-in-\texttt{Reals}, and they provide an efficient way for users to add PSD variables to MOSEK, bypassing the slack variables and equality constraints that would need to be added if the PSD constraint was added after the variables were created.

It is important to note that constraint and objective bridges are self-contained; they do not return objects that are used in other parts of the model. However, variable bridges do return objects that are used in other parts of the model. For example, adding $x \in \texttt{Reals}$ may add two variables $[x^+, x^-] \in \texttt{Nonnegatives}$ and return the expression $x^+ - x^-$ for $x$. The expression $x^+ - x^-$ must then be substituted for $x$ on every occurrence. A detailed description of how this substitution is achieved in code is non-trivial and is outside the scope of this paper.

### 4.3. Example

To demonstrate the combination of the three types of nodes in the hyper-graph, consider bridging a \texttt{ScalarQuadraticFunction} objective function to a solver that supports only:

- \texttt{VectorAffineFunction}-in-\texttt{RotatedSecondOrderCone} constraints;
- \texttt{ScalarAffineFunction} objective functions; and
- Variables in \texttt{Nonnegatives}.

As a simple example, we use:

$$\min x^2 + x + 1$$

s.t. $x \in \mathbb{R}_+^1$.

The first step is to introduce a slack variable $y$ and replace the objective with the \texttt{SingleVariable} function $y$:

$$\min y$$

s.t. $x^2 + x + 1 \leq y$

$$x \in \mathbb{R}_+^1$$

$y$ free.
However, since the solver supports only `ScalarAffineFunction` objective functions, the objective function is further bridged to:

\[
\begin{align*}
\min & \quad 1y + 0 \\
\text{s.t.} & \quad x^2 + x + 1 \leq y \\
& \quad x \in \mathbb{R}_+^1 \\
& \quad y \text{ free.}
\end{align*}
\]

The second step is to bridge the `ScalarQuadraticFunction-in-LessThan` constraint into a `VectorAffineFunction-in-RotatedSecondOrderCone` constraint using the relationship:

\[
0.5x^\top Q x + a^\top x + b \leq 0 \iff \|Ux\|^2 \leq 2(-a^\top x - b)
\implies \quad [1, -a^\top x - b, Ux] \in \text{RotatedSecondOrderCone},
\]

where \(Q = U^\top U\). Therefore, we get:

\[
\begin{align*}
\min & \quad 1y + 0 \\
\text{s.t.} & \quad [1, -x + y - 1, \sqrt{2}x] \in \text{RotatedSecondOrderCone} \\
& \quad x \in \mathbb{R}_+^1 \\
& \quad y \text{ free.}
\end{align*}
\]

Finally, since the solver does not support free variables, a variable bridge is used to convert \(y\) into two non-negative variables, resulting in:

\[
\begin{align*}
\min & \quad 1y^+ - 1y^- + 0 \\
\text{s.t.} & \quad [1, -x + y^+ - y^- - 1, \sqrt{2}x] \in \text{RotatedSecondOrderCone} \\
& \quad [x, y^+, y^-] \in \mathbb{R}_+^3.
\end{align*}
\]

Note how the expression \(y^+ - y^-\) is substituted for \(y\) throughout the model.

The optimal hyper-path corresponding to this example is given in Figure 3. To summarize, the `ScalarQuadraticFunction` objective node is bridged to a `SingleVariable` objective node, a \(x \in \text{Reals}\) variable node, and a `ScalarQuadraticFunction-in-LessThan` constraint node. Then, the `SingleVariable` objective is further bridged to a `ScalarAffineFunction` objective node, the \(x \in \text{Reals}\) variable node is bridged to a \(x \in \text{Nonnegatives}\) node, and the `ScalarQuadraticFunction-in-LessThan` constraint is bridged to a `VectorAffineFunction-in-RotatedSecondOrderCone` constraint node.
min \textit{ScalarQuadraticFunction}

min \textit{ScalarAffineFunction}

min \textit{SingleVariable}

\( x \in \text{Reals} \)

\( x \in \text{Nonnegatives} \)

\( \epsilon \in \text{LessThan} \)

\( \epsilon \in \text{RotatedSecondOrderCone} \)

Figure 3 Optimal hyper-path of example in Section 4.3. Dashed box is the objective node we want to add to the model, solid boxes are supported nodes, nodes with no outline are unsupported intermediate nodes, and arcs are bridges.

4.4. Future extensions

There are many aspects of the shortest path problem that we have not explored in our current implementation. For example, the current implementation in MathOptInterface assigns each bridge a weight of 1.0 in the path. Therefore, the objective of our shortest path problem is to minimize the number of bridges in a transformation. However, it is possible to use other scores for the desirability of a reformulation. For example, in the \textit{ScalarAffineFunction-in-Interval} example mentioned at the start of this section, the third option may be computationally beneficial since it adds fewer rows to the constraint matrix, even though it adds an extra variable. Therefore, we may assign the corresponding edge in the graph a lower weight (e.g., 0.6).

Similar systems for automatically transforming problems have appeared in the literature before, e.g., CVXPY has a similar concept called reductions (Agrawal et al. 2018), which we discussed in Section 2. However, since CVXPY targets a fixed set of solvers, it can pre-specify the chain of reductions needed for each solver, which has the benefit of being a simpler system. On the other hand, our shortest-path formulation enables us to separate the transformation logic from the solvers, making it easier to plug in new solvers without having to specify the reduction chain. The fact that we compute the sequence of transformations \textit{at runtime} additionally makes it possible to use new bridges and sets defined in third-party extensions.

In this section, we have discussed one challenge that arose due to our adoption of the MathOptInterface standard form, namely, the multitude of different ways of formulating equivalent models. In the next section, we discuss a second challenge: we can now formulate models that we cannot express in existing file formats. This motivates our need to create a new file format for mathematical optimization.
5. A new file format for mathematical optimization

As we saw in Section 3, a model in the MathOptInterface standard form is defined by a list of functions and a list of sets. In this section we utilise that fact to describe a new file format for mathematical optimization problems called MathOptFormat. MathOptFormat is a serialization of the MathOptInterface abstract data structure into a JSON file (ECMA International 2017), and it has the file-extension .mof.json. A complete definition of the format, including a JSONSchema (JSON Schema 2019) that can be used to validate MathOptFormat files, is available at https://github.com/jump-dev/MathOptFormat.

In addition, due to the role of file formats in problem interchange, the JSONSchema serves as the canonical description of the set of functions \( \mathcal{F} \) and sets \( \mathcal{S} \) defined in MathOptInterface. We envisage that the schema will be extended over time as more functions and sets are added to the MathOptInterface abstract data structure.

Importantly, the schema is a concrete representation of the format, and it includes a description of how the sparse vectors and matrices are stored. Moreover, although specified in JSON, this representation utilizes simple underlying data structures such as lists, dictionaries, and strings. Therefore, the format could be ported to a different format (e.g., protocol buffers (Google 2019)), without changing the basic layout and representation of the data.

We believe the creation of (yet another) new file format is justified because we can now write down problems that cannot be written in any other file format, e.g., multi-objective, semidefinite programs with nonlinear constraints.

Rather than rigorously define our format, we shall, in the interest of brevity, explain the main details of MathOptFormat through an example. Therefore, consider the following simple mixed-integer program:

\[
\begin{align*}
\max_{x,y} & \quad x + y \\
\text{subject to:} & \quad x \in \{0,1\} \\
& \quad y \leq 2.
\end{align*}
\]  

(7)

This example, encoded in MathOptFormat, is given in Figure 4.

Let us now describe each part of the file in Figure 4 in turn. First, notice that the file format is a valid JSON file. Inside the document, the model is stored as a single JSON object. JSON objects are key-value mappings enclosed by curly braces ({} and { }). There are four required keys at the top level:
1. version: A JSON object describing the minimum version of MathOptFormat needed to parse the file. This is included to safeguard against later revisions. It contains two fields: major and minor. These fields should be interpreted using semantic versioning (https://semver.org). The current version of MathOptFormat is v0.4.

2. variables: A list of JSON objects, with one object for each variable in the model. Each object has a required key name which maps to a unique string for that variable. It is illegal to have two variables with the same name. These names will be used later in the file to refer to each variable.

3. objective: A JSON object with one required key:

   (a) sense: A string which must be min, max, or feasibility.

   If the sense is min or max, a second key function, must be defined:
(b) **function**: A JSON object that describes the objective function. There are many different types of functions that MathOptFormat recognizes, each of which has a different structure. However, each function has a required key called `head` which is used to describe the type of the function. In this case, the function is `ScalarAffineFunction`.

4. **constraints**: A list of JSON objects, with one element for each constraint in the model. Each object has two required fields:

   (a) **function**: A JSON object that describes the function $f_i$ associated with constraint $i$. The function field is identical to the function field in `objective`; however, in this example, the first constraint function is a `SingleVariable` function of the variable $x$.

   (b) **set**: A JSON object that describes the set $S_i$ associated with constraint $i$. In this example, the second constraint set is the MathOptFormat set `LessThan` with the field `upper`.

Reflecting on the history of file formats outlined in Section 2.3, the following goals guided our development of MathOptFormat:

1. **Human-readable**: The format should be able to be read and edited by a human.

2. **Machine-readable**: The format should be readable by a variety of different programming languages without needing to write custom parsers in each language.

3. **Standardized**: The format should describe a general mathematical standard form in a manner that is unambiguous.

4. **Extensible**: The format should be able to be easily extended to incorporate new problem-classes as they arise and be backwards compatible with previous versions.

Goals 1 and 2 guided our decision to use JSON. In our opinion, JSON fits a happy medium between human-readable file formats and machine-readable file formats (although it has a large number of braces). Moreover, its strongest strength is its widespread support in every available major programming language.

Goals 3 and 4 are met by our decision to base the format on the MathOptInterface standard form. As we have shown, we believe MathOptInterface to be the most general and extensible standard form developed to-date. An important consequence of this decision is that MathOptFormat can represent models that cannot be represented in any other instance format. This ability will help spur the creation and development of solvers and benchmark libraries for new problem classes, such as mixed-integer nonlinear programs with conic constraints.
It is interesting to note that one item is clearly missing from our list of goals: efficiency. Efficiency can be broken down into two aspects: storage size, and the speed of reading and writing. We explicitly forego efficiency in this first version of MathOptFormat; however, storage size can be addressed by compressing the raw text files, and in the future, the speed of reading and writing could be improved by using a binary format. Notably, the underlying data structure of the format would not change.

6. The impact on JuMP and conclusions
We created MathOptInterface in order to improve the user experience of JuMP. Therefore, it is useful to reflect on how JuMP has changed with the introduction of MathOptInterface.

From a software engineering perspective, the largest change is that 90% of the code in JuMP was re-written during the transition. In terms of lines of code, 11,408 were added, 10,618 were deleted, 2,858 were modified, and only 1,520 remained unchanged. This represents a substantial investment in engineering time from a large number of individual contributors. In addition, 15,593 lines of code were added to the MathOptInterface package (although, 50% of these lines were tests for solvers), and many more thousand lines were added accounting for the more than 20 individual solvers supporting MathOptInterface.

From an architectural perspective, the main change is that instead of representing optimization models using three standard forms, JuMP now represents models using a combination of functions and sets. At the bottom level, instead of solvers implementing one of the three standard forms, they now declare a subset of function-in-set constraint pairs that they natively support, along with supported objective functions and sets for variables constrained on creation. Between these two representations sits the bridging system described in Section 4. The bridging transformations are done transparently to the user. Thus, analogous to Figure 1, the JuMP architecture now looks like the diagram in Figure 5.

Despite the major changes at the solver and interface level, little of the user-facing code in JuMP changed (aside from some sensible renaming). An example of a JuMP model using the CPLEX (IBM 2019) optimizer is given in Figure 6. This deceptively simple example demonstrates many unique features discussed in this paper. The $t \geq 0$ variable lower

---

3 The first release of JuMP with MathOptInterface was in February 2019.
bound is converted into a SingleVariable-in-GreaterThan constraint. The Int tag, informing JuMP that the variable \( t \) is an integer variable, is converted into a SingleVariable-in-Integer constraint. The SecondOrderCone constraint is bridged into a ScalarQuadraticFunction-in-LessThan constraint. The \( 1 \leq \text{sum}(x) \leq 3 \) constraint is formulated as a ScalarAffineFunction-in-Interval, and then bridged into a ScalarAffineFunction-in-LessThan constraint and a ScalarAffineFunction-in-GreaterThan constraint. Finally, after solving the problem with optimize!, we check that the termination_status is OPTIMAL before querying the objective value.

```julia
using JuMP, CPLEX
model = Model(CPLEX.Optimizer)
@variable(model, t >= 0, Int)
@variable(model, x[1:3] >= 0)
@constraint(model, [t; x] in SecondOrderCone())
@constraint(model, 1 <= sum(x) <= 3)
@objective(model, Min, t)
optimize!(model)
if termination_status(model) == MOI.OPTIMAL
    @show objective_value(model)
end
```

Figure 5 Architecture of the new version of JuMP. JuMP allows users to formulate models using combinations of functions and sets, solvers implement a subset of the complete functionality, and the bridging system transforms the user-provided model into an equivalent representation of the same model supported by the solver.

Figure 6 An example JuMP model demonstrating how the bridging system of MathOptInterface happens transparently behind-the-scenes.
This paper has described the MathOptInterface standard form and some of the challenges we faced when implementing it. However, the re-write of JuMP involved many more changes than the ones outlined here. In particular, we have not discussed:

- The API of MathOptInterface.jl, which includes a standardized way to get and set a variety of model and solver attributes, and the ability to incrementally modify problems in-place (e.g., deleting variables, and changing constraint coefficients);
- JuMP’s manual and automatic caching modes for solvers that do not support the aforementioned incremental modifications;
- JuMP’s new direct mode, which avoids storing an intermediate copy of the model and instead hooks directly into the underlying solver with minimal overhead;
- The introduction of a new status reporting mechanism featuring three distinct types of solution statuses: termination status (Why did the solver stop?), primal status (What is the status of the primal solution?), and dual status (What is the status of the dual solution?);
- JuMP’s re-vamped support for solver callbacks, offering both solver-independent callbacks and solver-dependent callbacks, which allow the user to interact with solver-specific functionality; and
- JuMP’s unified testing infrastructure for solvers, which subjects all solvers to thousands of tests for correctness every time a change is made to the codebase. This testing has revealed bugs and undocumented behavior in a number of solvers.

We leave a description of these changes, and many others, to future work.

For more information on JuMP, including documentation, examples, tutorials, and source code, readers are directed to https://jump.dev.

Acknowledgments

JuMP and MathOptInterface are open-source projects that are made possible by volunteer contributions that include not just writing code, but also finding and reporting bugs, writing and editing documentation, and replying to questions on the community forum. Among the contributors, we owe special thanks to Mathieu Besançon, Guilherme Bodin, Chris Coey, Carleton Coffrin, Robin Deits, Twan Koolen, Vitor Nesello, François Pacaud, Robert Schwarz, Issam Tahiri, Mathieu Tanneau, Juan Pablo Vielma, and Ulf Worsøe for their work helping to migrate the JuMP ecosystem to MathOptInterface, which included substantial rewrites of solver interfaces. We thank Ross Anderson for comments on a draft of this paper.

MathOptInterface was conceived during the first JuMP-dev workshop at MIT in June 2017; we thank the MIT Sloan Latin America Office for their sponsorship of this workshop. We thank MIT Sloan, IDEX
Bordeaux, and Changhyun Kwon for sponsorship of the 2018 workshop, and we thank NSF under grant OAC-1835443, MISTI under the MIT-Chile UAI and PUC seed funds, and the Pontificia Universidad Católica de Chile’s Institute of Mathematical and Computational Engineering and Department of Industrial Engineering for sponsorship of the 2019 workshop. The first author acknowledges the funding support of an F.R.S.-FNRS fellowship.

References

Agrawal A, Verschueren R, Diamond S, Boyd S (2018) A rewriting system for convex optimization problems. *Journal of Control and Decision* 5(1):42–60.

Beale EML, Tomlin JA (1970) Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. *OR* 69(447-454):99.

Bezanson J, Edelman A, Karpinski S, Shah VB (2017) Julia: A fresh approach to numerical computing. *SIAM Review* 59(1):65–98.

Borchers B (1999) CSDP, A C library for semidefinite programming. *Optimization methods and Software* 11(1-4):613–623.

Brook A, Kendrick D, Meeraus A (1988) GAMS, a user’s guide. *ACM Signum Newsletter* 23(3-4):10–11.

Diamond S, Boyd S (2016) CVXPY: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research* 17(1):2909–2913.

Dirkse SP, Ferris MC (1995) The PATH solver: a nonmonotone stabilization scheme for mixed complementarity problems. *Optimization Methods and Software* 5(2):123–156.

Dunning I, Huchette J, Lubin M (2017) JuMP: A Modeling Language for Mathematical Optimization. *SIAM Review* 59(2):295–320.

ECMA International (2017) The JSON Data Interchange Syntax. Technical Report 404, ECMA, Geneva.

Forrest J, Vigerske S, Ralphs T, Hafer L, jpfasano, Santos HG, Saltzman M, Gassmann H, Kristjansson B, King A (2019) coin-or/clp: Version 1.17.3. URL http://dx.doi.org/10.5281/zenodo.3246629

Fourer R (1983) Modeling languages versus matrix generators for linear programming. *ACM Transactions on Mathematical Software (TOMS)* 9(2):143–183.

Fourer R, Gay DM, Kernighan BW (1990) A modeling language for mathematical programming. *Management Science* 36(5):519–554.

Fourer R, Ma J, Martin K (2010) OSiL: An instance language for optimization. *Computational Optimization and Applications* 45(1):181–203.

Friberg HA (2016) CBLIB 2014: A benchmark library for conic mixed-integer and continuous optimization. *Mathematical Programming Computation* 8(2):191–214.

Gallo G, Longo G, Pallottino S, Nguyen S (1993) Directed hypergraphs and applications. *Discrete applied mathematics* 42(2-3):177–201.
Gay D (1997) Hooking your solver to AMPL. Technical Report 93-10, AT&T Bell Laboratories, Murray Hill, NJ.

Gay D (2005) Writing .nl files. Technical Report SAND2005-7907P, Sandia National Laboratories.

Google (2019) Protocol buffers. URL https://developers.google.com/protocol-buffers

Grant M, Boyd S (2014) CVX: Matlab software for disciplined convex programming, version 2.1.

Grant M, Boyd S, Ye Y (2006) Disciplined convex programming. Global optimization, 155–210 (Springer).

Gurobi Optimization (2019) Gurobi Optimizer Reference Manual 9.0. URL https://www.gurobi.com/documentation/9.0/refman/index.html

Hart WE, Laird CD, Watson JP, Woodruff DL, Hackebeil GA, Nicholson BL, Siïola JD (2017) Pyomo-optimization modeling in python, volume 67 (Springer).

IBM (2019) IBM ILOG CPLEX Optimization Studio V12.10.0 documentation. URL https://www.ibm.com/support/knowledgecenter/en/SSSA5P_12.10.0/COS_KC_home.html

IBM World Trade Corporation (1976) IBM Mathematical Programming System Extended/370 (MPS/370) Program Reference Manual. Technical Report SH19-1095-1, IBM, New York.

Italiano GF, Nanni U (1989) Online maintenance of minimal directed hypergraphs. Technical Report CUCS-405-89, Department of Computer Science, Columbia University Series.

JSON Schema (2019) JSON Schema 2019-09. URL https://json-schema.org

Löfberg J (2004) Yalmip: A toolbox for modeling and optimization in matlab. Proceedings of the CACSD Conference, volume 3 (Taipei, Taiwan).

Löhne A (2011) Vector optimization with infimum and supremum (Springer Science & Business Media).

LP-solve (2016) LP file format. URL http://lpsolve.sourceforge.net/5.5/lp-format.htm

Lubin M, Dunning I (2015) Computing in Operations Research Using Julia. INFORMS Journal on Computing 27(2):238–248.

Löfberg J (2009) Dualize it: software for automatic primal and dual conversions of conic programs. Optimization Methods and Software 24:313 – 325.

MOSEK ApS (2019) MOSEK Optimization Suite 9.1.9. URL https://docs.mosek.com/9.1/intro/index.html

O’Donoghue B, Chu E, Parrikh N, Boyd S (2016) Conic optimization via operator splitting and homogeneous self-dual embedding. Journal of Optimization Theory and Applications 169(3):1042–1068.

Orchard-Hays W (1984) History of Mathematical Programming Systems. Annals of the History of Computing 6(3):296–312.

Sun D, Toh KC, Yuan Y, Zhao XY (2019) SDPNAL+: A matlab software for semidefinite programming with bound constraints (version 1.0). Optimization Methods and Software 1–29.
Appendix

A. Sets defined by MathOptInterface

Here we provide a complete list of the sets defined by MathOptInterface in $\mathcal{S}$.

A.1. One-dimensional sets

MathOptInterface defines the following one-dimensional sets where $S \subseteq \mathbb{R}$:

- **LessThan**$(u)$: $(-\infty, u]$ where $u \in \mathbb{R}$.
- **GreaterThan**$(l)$: $[l, \infty)$ where $l \in \mathbb{R}$.
- **EqualTo**$(a)$: $\{a\}$ where $a \in \mathbb{R}$.
- **Interval**$(l, u)$: $[l, u]$ where $l \in \mathbb{R}$, $u \in \mathbb{R}$, and $l \leq u$.
- **Integer**: $\mathbb{Z}$.
- **ZeroOne**: $\{0, 1\}$.
- **Semionteger**$(l, u)$: $\{0\} \cup \{l, \ldots, u\}$ where $l \in \mathbb{Z}$, $u \in \mathbb{Z}$, $l \leq u$.
- **Semicontinuous**$(l, u)$: $\{0\} \cup [l, u]$ where $l \in \mathbb{R}$, $u \in \mathbb{R}$, $l \leq u$.

A.2. Cones

MathOptInterface defines the following multi-dimensional cones where $S \subseteq \mathbb{R}^N$:

- **Zeros**$: \{0\}^N$.
- **Reals**$: \mathbb{R}^N$.
- **Nonpositives**$: \{x \in \mathbb{R}^N : x \leq 0\}$.
- **Nonnegatives**$: \{x \in \mathbb{R}^N : x \geq 0\}$.
- **SecondOrderCone**$: \{(t, x) \in \mathbb{R}^{1+N} : t \geq |x|_2^2\}$.
- **RotatedSecondOrderCone**$: \{(t, x, y) \in \mathbb{R}^{2+N} : 2ty \geq |x|_2^2, t \geq 0, u \geq 0\}$.
- **ExponentialCone**$: \{(x, y) \in \mathbb{R}^2 : ye^{x/y} \leq z, y \geq 0\}$.
- **DualExponentialCone**$: \{(u, v, w) \in \mathbb{R}^3 : -ue^{v/w} \leq e^1w, u < 0\}$.
- **GeometricMeanCone**$: \{(t, x) \in \mathbb{R}^{1+N} : \prod_{i=1}^N x_i^{1/N}\}$.
- **PowerCone**$(a)$: $\{(x, y, z) \in \mathbb{R}^3 : x^a y^{1-a} \geq |z|, x \geq 0, y \geq 0\}$ where $a \in \mathbb{R}$.
- **DualPowerCone**$(a)$: $\{(u, v, w) \in \mathbb{R}^3 : (\frac{u}{v})^a (\frac{w}{1-w})^{1-a} \geq |w|, u \geq 0, w \geq 0\}$ where $a \in \mathbb{R}$.
- **NormOneCone**$: \{(t, x) \in \mathbb{R}^{1+N} : t \geq \sum_{i=1}^N |x_i|\}$.
- **NormInfinityCone**$: \{(t, x) \in \mathbb{R}^{1+N} : t \geq \max_{i=1,\ldots,N} |x_i|\}$.
- **RelativeEntropyCone**$: \{(u, v, w) \in \mathbb{R}^{1+2N} : u \geq \sum_{i=1}^N w_i \log \left(\frac{w_i}{v_i}\right), v \geq 0, w \geq 0\}$.
A.3. Matrix cones

MathOptInterface defines the following matrix-valued cones (unless specified, $X$ is assumed to be a $d \times d$ matrix):

- **RootDetConeTriangle**: \( \{(t, X) \in \mathbb{R}^{1+d(d+1)/2} : t \leq \det(X)^{1/d}\} \).
- **RootDetConeSquare**: \( \{(t, X) \in \mathbb{R}^{1+d^2} : t \leq \det(X)^{1/d}, X = X^\top\} \).
- **LogDetConeTriangle**: \( \{(t, u, X) \in \mathbb{R}^{2+d(d+1)/2} : t \leq u \det(X/u), u > 0\} \).
- **LogDetConeSquare**: \( \{(t, u, X) \in \mathbb{R}^{2+d^2} : t \leq u \det(X/u), u > 0, X = X^\top\} \).
- **PositiveSemidefiniteConeTriangle**: The cone of positive semidefinite matrices \( \{X \in \mathbb{R}^{d(d+1)/2} : X \succeq 0\} \).
- **PositiveSemidefiniteConeSquare**: The cone of positive semidefinite matrices \( \{X \in \mathbb{R}^{d^2} : X \succeq 0, X = X^\top\} \).
- **NormSpectralCone**: \( \{(t, X) \in \mathbb{R}^{1+M \times N} : t \geq \sigma_1(X)\} \), where \( \sigma_1(X) \) is the 1st singular value of the matrix $X$ with dimensions $M \times N$.
- **NormNuclearCone**: \( \{(t, X) \in \mathbb{R}^{1+M \times N} : t \geq \sum_i \sigma_i(X)\} \), where \( \sigma_i(X) \) is the $i$th singular value of the matrix $X$ with dimensions $M \times N$.

Some of these cones can take two forms: **XXXConeTriangle** and **XXXConeSquare**. In **XXXConeTriangle** sets, the matrix is assumed to be symmetric, and the elements are provided by a vector, in which the entries of the upper-right triangular part of the matrix are given column by column (or equivalently, the entries of the lower-left triangular part are given row by row). In **XXXConeSquare** sets, the entries of the matrix are given column by column (or equivalently, row by row), and the matrix is constrained to be symmetric.

As an example, given a 2-by-2 matrix of variables $X$ and a one-dimensional variable $t$, we can specify a root-det constraint as \([t, X_{11}, X_{12}, X_{22}] \in \text{RootDetConeTriangle}\) or \([t, X_{11}, X_{12}, X_{21}, X_{22}] \in \text{RootDetConeSquare}\). We provide both forms to enable flexibility for solvers who may natively support one or the other. Transformations between **XXXConeTriangle** and **XXXConeSquare** are handled by bridges, which removes the chance of conversion mistakes by users or solver developers.

A.4. Multi-dimensional sets with combinatorial structure

MathOptInterface also defines a number of multi-dimensional sets that specify some combinatorial structure.

- **Complements**: A set defining a mixed-complementarity constraint:

\[
\{(x, y) \in \mathbb{R}^{2N} : \begin{cases} y_i \in (l_i, u_i) & \implies x_i = 0 \\ y_i = l_i & \implies x_i \geq 0 & \forall i = 1, \ldots, N \\ y_i = u_i & \implies x_i \leq 0 \end{cases} \},
\]

where $l_i$ and $u_i$ are the lower- and upper-bounds on variable $y_i$ (added separately as **SingleVariable-in-XXX** constraints).

Note that \((x, y)\) is defined by a single multi-dimensional function (e.g., **VectorAffineFunction**), but the $y$ component must be interpretable as a vector of variables. For simplicity, complementarity constraints are often written $f(x) \perp y$. As an example, $-4x + 1 \perp x$ can be specified as $[-4x + 1, x] \in \text{Complements}$. Classically, the bounding set for $y$ is $[0, \infty)$, which recovers the “classical” complementarity constraint $0 \leq f(x) \perp y \geq 0$.

- **IndicatorSet**\((b, S)\): A set used to construct indicator constraints:

\[
\{(y, x) \in \mathbb{R}^{1+N} : y = b \implies x \in S\},
\]
where \( b \in \{0, 1\} \) and \( S \in S \). (Note that most solvers will require a constraint that \( y \in \text{ZeroOne} \) before the user can add this constraint.)

For example, \( y = 0 \Rightarrow 2x \leq 1 \) can be given by: \([y, 2x] \in \text{IndicatorSet}(0, \text{LessThan}(1))\).

- **SOS1(\( w \))**: A set for Special Ordered Sets of Type I constraints (Beale and Tomlin 1970):
  \[
  \{ x \in \mathbb{R}^N : |\{ x_i \neq 0 \}_{i=1}^N | \leq 1 \},
  \]
  where \( w \in \mathbb{R}^N \). Note that although the weight vector \( w \) is not used in the definition of the set, it induces an ordering on the elements of \( x \) that may be used to guide the solver’s branching decisions.

- **SOS2(\( w \))**: A set for Special Ordered Sets of Type II constraints (Beale and Tomlin 1970):
  \[
  \{ x \in \mathbb{R}^N : |\{ x_i \neq 0 \}_{i=1}^N | \leq 2, \text{non-zeros are adjacent under } w \},
  \]
  where \( w \in \mathbb{R}^N \). Here, the weight vector \( w \) induces an ordering on the elements of \( x \), and at most two elements can be non-zero. In addition, if two elements are non-zero, they must be adjacent according to the ordering.