JuMP: A MODELING LANGUAGE FOR MATHEMATICAL OPTIMIZATION

IAIN DUNNING, JOEY HUCHETTE, MILES LUBIN *

Abstract. JuMP is an open-source modeling language that allows users to express a wide range of optimization problems (linear, mixed-integer, quadratic, conic-quadratic, and nonlinear) in a high-level, algebraic syntax. JuMP takes advantage of advanced features of the Julia programming language to achieve performance on par with commercial modeling tools. In this work we will provide benchmarks, present the novel aspects of the implementation, and discuss how JuMP can be extended to new problem classes and composed with state-of-the-art tools for visualization and interactivity.

Key words. algebraic modeling languages, automatic differentiation, dynamic code generation

AMS subject classifications. 90C04, 90C05, 90C06, 90C30, 65D25

1. Introduction. Orchard-Hays, who developed some of the first software for linear programming (LP) in collaboration with Dantzig, observed that the field of mathematical optimization developed hand-in-hand with the field of computing [56]. Beginning with the introduction of IBM’s first commercial scientific computer in 1952, advancements in technology were immediately put to use for solving military and industrial planning problems. LP software was viewed as generally reliable by the 1970s, when mainframe computers had become mainstream. However, developers of these systems recognized that the difficulty of translating the complex mathematical formulation of a problem into the requisite input formats based on punch cards was a major barrier to adoption [22].

In the late 1970s, the first algebraic modeling languages (AMLs) were developed with the aim of allowing users to express LP problems in a natural, algebraic form similar to the original mathematical expressions, much in the same way that MATLAB was created contemporaneously to provide a high-level interface to linear algebra. Similar to how MATLAB translated user input into calls to LINPACK [19], AMLs do not solve optimization problems; they provide the problems to optimization routines called solvers. GAMS [13] and AMPL [24], whose development started in 1978 and 1985 respectively, are widely recognized as having made a significant impact on the adoption of mathematical optimization in a number of fields. Thanks in part to the ease-of-use of these AMLs, Dantzig’s simplex algorithm for LP was named one of the top 10 algorithms of the 20th century [17].

To date, AMPL, GAMS, and similar commercial packages represent the state of the art in AMLs and are widely used in both academia and industry. These AMLs, despite their idiosyncratic syntax, are quite efficient at what they were designed for, which is solving a single instance of an optimization problem. However, because they are not fully-featured programming languages, the seemingly simple extension to solving a sequence of related optimization problems in a loop while exploiting algorithmic hot-starts is a known challenge [44, 15]. The commercial nature of the software impedes the development of plug-ins and extensions which would arise naturally out of academic research, and the standalone nature makes it nontrivial to embed optimization into larger projects or connect to novel data sources and modern visualization systems in MATLAB, Python, or R, for example.

*MIT Operations Research Center {idunning,huchette,mlubin}@mit.edu.
Since the 2000s, a number of open-source AMLs have been developed by academics. YALMIP [47] and CVX [33], both based on MATLAB, were created to provide functionality such as handling of semidefinite and disciplined convex [34] optimization, which was not present in commercial AMLs. CVX in particular has been cited as making convex optimization as accessible from MATLAB as is linear algebra and was credited for its extensive use in both research and teaching [21]. Pyomo [39] is a Python-based AML which was originally designed as a clone of AMPL but was later extended to new problem classes such as stochastic programming [66]. These AMLs would be sufficient were it not for the issue of speed. Because they are embedded in high-level languages like MATLAB and Python, it is quite difficult to match the efficiency of commercial AMLs. The seemingly perverse situation where the AML processing time is a bottleneck more so than solving the optimization problem itself is not uncommon [48], leaving users with the well-known trade-off between quick prototyping of optimization models in a high-level language versus tedious yet efficient implementation in a language like C, C++, or Fortran.

In this paper, we present JuMP, an open-source AML under development since 2012 which has already seen use in research and teaching [20]. This paper is not a tutorial. Instead, we intend to highlight the novel technical aspects of the implementation in sufficient generality to apply broadly beyond the context of AMLs. JuMP is embedded in Julia [8], a relatively new language for technical computing that claims to shatter the performance gap between low-level and high-level languages. JuMP takes advantage of a number of advanced technical features in Julia in order to be the first high-level open-source AML to compete performance-wise with GAMS and AMPL for modeling linear, quadratic, conic-quadratic, and nonlinear optimization problems on a series of benchmarks which we will present here.

This work is an extension of [48], which introduced JuMP as an AML for linear and mixed-integer optimization with a proof-of-concept implementation of symbolic differentiation for computation of derivatives. Since then, JuMP has gained support for quadratic, conic-quadratic, and general derivative-based nonlinear optimization as well as a number of extensions for specialized problem classes. The calculation of derivatives in JuMP is now based on automatic (or algorithmic) differentiation, a body of techniques for computing derivatives of code without the cost of symbolic differentiation and without the approximation error of finite differencing.

The remainder of the paper is structured as follows. In Section 2 we introduce in more detail the tasks required of an AML. In Sections 3 and 4 we discuss JuMP’s use of syntactic macros and code generation, two advanced technical features of Julia which are key to JuMP’s performance. In Section 5 we discuss JuMP’s implementation of derivative computations. In Section 6 we discuss a number of powerful extensions which have been built on top of JuMP, and in Section 7 we conclude with a demonstration of how JuMP can be composed with the growing ecosystem of Julia packages to produce a compelling interactive and visual user interface with applications in both academia and industry.

2. The role of a modeling language. Prior to the introduction of AMLs (and continuing to a lesser degree today), users would write low-level code which directly generated the input data structures for an optimization problem. Recall that
Fig. 1. Sparsity pattern of the constraint coefficient matrix for an integer programming formulation of the popular sudoku number puzzle [5], where the points represent non-zero elements. There are 729 variables and 324 constraints, with the constraints grouped into four sets: each number must appear in every row once, each number must appear in every column once, each number must appear in every 3-by-3 sub-grid once, and each of the 81 cells must have a number.

standard-form linear programming problems can be stated as

$$\min_{x \in \mathbb{R}^n} c^T x$$

$$\text{s.t. } Ax = b,$$

$$x \geq 0,$$

that is, minimization of a linear objective subject to linear equality and inequality constraints (all elements of $x$ must be nonnegative). In the case of LP, the input data structures are the vectors $c$ and $b$ and the matrix $A$ in sparse format, and the routines to generate these data structures are called matrix generators [22]. Typical mathematical optimization models have complex indexing schemes; for example, an airline revenue management model may have decision variables $x_{s,d,c}$ which represent the number of tickets to sell from the source $s$ to destination $d$ in fare class $c$, where not all possible combinations of source, destination and fare class are valid. A matrix generator would need to efficiently map these variables into a single list of linear indices and then construct the corresponding sparse matrix $A$ as input to the solver, which is tedious, error-prone, and fragile with respect to changes in the mathematical model. An example sparsity pattern in Figure 1 demonstrates that these can be quite complex even for small problems. This discussion extends naturally to quadratic expressions $c^T x + \frac{1}{2} x^T Q x$; the matrix $Q$ is simply another component of the input data structure.

The role of an AML is similar for the nonlinear optimization problems which often arise in scientific and engineering applications. The standard form for derivative-based nonlinear optimization problems is

$$\min_{x \in \mathbb{R}^n} f(x)$$

$$\text{s.t. } g_i(x) \leq 0 \quad i = 1, \ldots, m_g,$$

$$h_i(x) = 0 \quad i = 1, \ldots, m_h,$$

where $f, g_i, h_i : \mathbb{R}^n \to \mathbb{R}$ are linear or nonlinear functions. Depending on certain properties of $f, g$, and $h$ such as convexity, these problems may be easy or hard to solve to a global solution; regardless, the solution methods often rely on the availability of first-order derivatives, that is, the gradient vectors $\nabla f(x), \nabla g_i(x)$, and $\nabla h_i(x)$, and may be further accelerated by the availability of second-order derivatives, that is, the Hessian matrices $\nabla^2 f(x), \nabla^2 g_i(x)$, and $\nabla^2 h_i(x)$. Commonly, however, one might forgo second-order derivatives or even first-order derivatives to avoid the tedious
and error-prone exercise of implementing code to evaluate them by hand, even when providing them could reduce the solution time.

In both the linear and nonlinear cases, the role of a modeling language is to accept closed-form algebraic expressions as user input and transparently generate the required input to the solver, handling any low-level details of communicating with the solver, either via a callable library or by exchanging specially formatted files. For linear and quadratic problems, the input data are the vectors and sparse matrices previously discussed. For nonlinear problems, the most complex part of the input to solvers is the routines for evaluating derivatives of the user-provided expressions.

The tasks that an AML must perform, therefore, can be roughly divided into two simple categories: first, to load the user’s input into memory, and second, to generate the input required by the solver, according to the class of the problem. For both of these tasks, we have made some unorthodox design decisions in JuMP in order to achieve good performance under the constraints of being embedded within a high-level language. We will review these in the following sections.

We note that JuMP provides access to a number of advanced techniques which have not been typically available in AMLs. For example, branch-and-cut is a powerful technique in integer programming for accelerating the solution process by dynamically improving the convex (linear) relaxations used within the branch-and-bound algorithm; this technique was essential to the significant progress in the past decade in exact solution methods for the traveling salesman problem [3]. Branch-and-cut requires bidirectional communication with a solver during the solution process, which has typically required low-level coding in C++ for an efficient implementation. JuMP provides a simple, high-level interface to branch-and-cut and other similar techniques which has been used in both research and teaching [20].

3. Syntactic macros: parsing without a parser. AMLs like AMPL and GAMS are stand-alone in the sense that they have defined their own syntax entirely separate from any existing programming language. They have their own formats for providing input data (although they can also connect to databases and spreadsheets) and implement custom parsers for their proprietary syntax; for example, AMPL uses the LEX and YACC parser generator utilities [23].

Embedding an AML within an existing programming language brings with it the benefit of being able to bootstrap off the existing, well defined grammar and syntax of the language, eliminating a complex part of implementing an AML. However, it also brings with it the challenge of obtaining the desired expressiveness and ease of use within the limits of the syntax of the parent language.

The most common approach (taken by Pyomo, YALMIP, and others) to capturing user input is operator overloading. One introduces a new class of objects, say, to represent a decision variable or vector of decision variables, and extends the language’s definition of basic operators like +, *, −, etc, which, instead of performing arithmetic operations, build up data structures which represent the expression. For example, to represent a quadratic expression \( \sum_{(i,j) \in J} b_{ij} x_i x_j + \sum_{i \in I} a_i x_i + c \), one stores the constant \( c \), the coefficient vectors \( b \), \( a \), and the index sets \( I \) and \( J \). Letting \( n \) be the number of decision variables in a problem, an unfortunate property of addition of two quadratic expressions is that the size of the resulting expression is not bounded by a constant independent of \( n \), simply because the coefficient and index vectors can have as many as \( O(n^2) \) terms. This means that basic operations like addition and subtraction are no longer fast, constant-time operations, a property which is almost always taken for granted in the case of floating-point numbers. As a concrete example, consider the
following quadratic expression in the variable $x$ indexed over $\{1, \ldots, d\} \times \{1, \ldots, d\}$:

\[
(3.1) \quad 1 + \sum_{i=1}^{d} \sum_{j=1}^{d} |c_{j} - i|(1 - x_{i,j})x_{1,j}
\]

In Python, one might naturally express (3.1) as

\[
1 + \text{sum}(\text{abs}(c[j]-i)*(1-x[i,j])*x[0,j] \text{ for } i \text{ in range}(d) \text{ for } j \text{ in range}(d))
\]

which takes advantage of the built-in \text{sum} command which internally accumulates the terms one-by-one by calling the addition operator $d^2$ times. The partial sums are quadratic expressions which have $O(d^2)$ terms, so this naive approach can have a cost of $O(d^4) = O(n^2)$ operations and excessive memory allocations. An obvious workaround for this issue is to accumulate the terms in a single output expression instead of a generating a new expression for each partial sum. While there are a number of ways to mitigate this slow behavior within the framework of operator overloading, our benchmarks will demonstrate that they have not been sufficient to achieve the best performance.

When designing JuMP, we were not satisfied by the performance limitations of operator overloading and instead turned to an advanced feature of Julia called \textit{syntactic macros} [9]. Readers may be familiar with macros in C and C++ which perform textual substitutions; macros in Julia are much more powerful in that they function at the level of syntax. For example, the expression (3.1) could be written in a pseudo-JuMP syntax as

\[
@\text{expr}(1 + \text{sum}(\text{abs}(c[j]-i)*(1-x[i,j])*x[1,j], i \text{ in } 1:N, j \text{ in } 1:N))
\]

The @ sign denotes a call to a macro named \text{expr}. The input to the macro will be a data structure representing the \textit{expression} contained within, not simply a string of text. That is, Julia’s internal parser will be invoked to parse the expression, but instead of directly evaluating it or compiling it to code, it will be sent to a routine written in Julia which we (as authors of JuMP) have defined. Note that the syntax \text{sum}{} is generally not valid Julia code, although it is recognized by the Julia parser, which allows us to endow it with a new meaning in the context of JuMP.

Macros enable JuMP to provide a natural syntax for algebraic modeling without writing a custom text-based parser and without the drawbacks of operator overloading. Within the computer science community, macros have been recognized as a useful tool for developing domain-specific languages, of which JuMP is an example [62]. Indeed, the implementation of macros in Julia draws its inspiration from Lisp [9]. However, such functionality historically has not been available within programming languages targeted at scientific computing, and, to our knowledge, JuMP is the first AML to be designed around syntactic macros.

4. Code generation for linear and conic-quadratic models. Linear and conic-quadratic optimization problems are essential and surprisingly general modeling paradigms that appear throughout operations research and other varied fields—often at extremely large scales. Quadratic optimization generalizes linear optimization by allowing convex quadratic terms $\frac{1}{2}x^TQx$ in the objective, and conic-quadratic generalizes quadratic by allowing constraints of the form $||x||_2 \leq t$, where both $x$ and $t$ are decision variables [46]. Computational tools for solving these problems derive their success from exploiting the well-defined structure of these problems. Analogously, JuMP is able to efficiently process enormous problems by taking advantage
of structural properties and generating efficient code through Julia’s code generation functionality.

Julia is, at the same time, both a dynamic and compiled language. Julia uses the LLVM compiler [45] dynamically at runtime, and can generate efficient, low-level code as needed. This technical feature is one of the reasons why Julia can achieve C-like performance in general [8], but we will restrict our discussion to how JuMP takes advantage of it.

In the previous section we described how JuMP uses macros to accept user input in the form of a data structure which represents the input expression. The other side of macros is code generation. More specifically, macros can be understood as functions whose input is code and whose output is code. Given an input expression, a macro produces a data structure which represents an output expression, and that expression is then substituted in place and compiled. For example, the macro which takes as input the expression (3.1) would output, in pseudo-code from, the following code:

Initialize an empty quadratic expression q
Add 1 to the constant term
Count the number of terms K in the sum{} expression
for i in 1:d, j in 1:d
   Append -abs(c[j]-i)*x[i,j]*x[1,j] to the quadratic terms in q
   Append abs(c[j]-i)*x[1,j] to the linear terms in q
end

Note that this code runs in \( O(d^2) \) operations, a significant improvement over the \( O(d^4) \) naive operator overloading approach. The code produced is also similar to a hand-written matrix generator. Indeed, one could summarize JuMP’s approach to generating linear and quadratic models as translating users’ algebraic input into fast, compiled code which acts as a matrix generator.

Currently, JuMP follows the convention of many commercial solvers by accepting conic-quadratic input in the quadratic form \( x^T x \leq t^2 \) with the additional restriction \( t \geq 0 \), although we intend to implement more specialized syntax in a future release.

4.1. Benchmarks. We now provide computational evidence that JuMP is able to produce quadratic and conic-quadratic optimization models, in a format suitable for consumption by a solver, as fast as state-of-the-art commercial modeling languages. To do so we measure the time elapsed between launching the executable that builds the model and the time that the solver begins the solution process, as determined by recording when the first output appears from the solver. This methodology allows the modeling language to use a direct in-memory solver interface if it desires, or in the case of some tools a compact file representation. We selected Gurobi 6.0.0 [38] as the solver, and evaluated the following modeling systems: the Gurobi C++ interface (based on operator overloading), JuMP 0.9 with Julia 0.3.7, AMPL 20141228 [24], GAMS 24.3.3 [13], Pyomo 4.0.0 with Python 2.7.9 [39], and CVX 2.1 [33] and YALMIP 20150204 [47] with MATLAB R2014b.

We implemented two different optimization problems in all seven modeling languages: a linear-quadratic control problem (lqcp) and a facility location problem (fac). The models are further described in the appendix. The results (Table 1) show that for lqcp, JuMP, AMPL, and the C++ interface are roughly equivalent at the largest scale, with GAMS approximately five times slower and CVX fourteen times slower than JuMP. Pyomo and YALMIP were significantly slower and were unable to construct the largest model within ten minutes. For fac, JuMP, AMPL, GAMS
and the C++ interface times all perform roughly the same, while Pyomo is unable to build the largest instance with ten minutes, YALMIP can build only the smallest instance within the time limit, and CVX is unable to build any instances within the time limit.

JuMP has a noticeable start-up cost of a few seconds even for the smallest instances. This start-up cost is primarily composed of compilation time; however, if a family of models is solved multiple times within a single session, this cost of compilation is only paid for the first time that an instance is solved. That is, when solving a sequence of instances in a loop, the amortized cost of compilation is negligible.

5. Computing derivatives for nonlinear models. Recall that the role of a modeling language for nonlinear optimization is to allow users to specify closed-form, algebraic expressions for the objective function $f$ and constraints $g$ and $h$ in the formulation (2.2) and communicate first-order and typically second-order derivatives with the optimization solver. Commercial modeling languages like AMPL and GAMS represent the state of the art in modeling languages for nonlinear optimization. Likely because of the increased complexity of computing derivatives, even fewer open-source implementations exist than for linear or quadratic models.

Notable alternative approaches to traditional algebraic modeling for nonlinear optimization include CasADi [2] and CVX [33]. CasADi allows interactive, scalar- and matrix-based construction of nonlinear expressions via operator overloading with automatic computation of derivatives for optimization. CasADi has specialized features for optimal control but, unlike traditional AMLs, does not support linear optimization as a special case. CVX, based on the principle of disciplined convex programming (DCP) [34], allows users to express convex optimization problems in a specialized format which can be transformed into or approximated by conic programming without the need for computing derivatives. The traditional nonlinear optimization formulation considered here applies more generally to derivative-based convex and nonconvex optimization.

JuMP, like AMPL and GAMS, uses techniques from automatic (or algorithmic) differentiation (AD) to evaluate derivatives of user-defined expressions. In this section, we introduce these techniques, explain novel aspects of their implementation in Julia.

\[\text{Table 1}\]

<table>
<thead>
<tr>
<th>Instance</th>
<th>JuMP</th>
<th>GRB/C++</th>
<th>AMPL</th>
<th>GAMS</th>
<th>Pyomo</th>
<th>CVX</th>
<th>YALMIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>lqcp-500</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>53</td>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>lqcp-1000</td>
<td>10</td>
<td>6</td>
<td>5</td>
<td>13</td>
<td>214</td>
<td>43</td>
<td>115</td>
</tr>
<tr>
<td>lqcp-1500</td>
<td>13</td>
<td>14</td>
<td>12</td>
<td>42</td>
<td>500</td>
<td>120</td>
<td>400</td>
</tr>
<tr>
<td>lqcp-2000</td>
<td>18</td>
<td>26</td>
<td>22</td>
<td>102</td>
<td>&gt;600</td>
<td>267</td>
<td>&gt;600</td>
</tr>
<tr>
<td>fac-25</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>&gt;600</td>
<td>465</td>
</tr>
<tr>
<td>fac-50</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>110</td>
<td>&gt;600</td>
<td>&gt;600</td>
</tr>
<tr>
<td>fac-75</td>
<td>13</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>376</td>
<td>&gt;600</td>
<td>&gt;600</td>
</tr>
<tr>
<td>fac-100</td>
<td>23</td>
<td>12</td>
<td>17</td>
<td>25</td>
<td>&gt;600</td>
<td>&gt;600</td>
<td>&gt;600</td>
</tr>
</tbody>
</table>

*Time (sec.) to generate each model and pass it to the solver, a comparison between JuMP and existing commercial and open-source modeling languages. The lqcp instances have quadratic objectives and linear constraints. The fac instances have linear objectives and conic-quadratic constraints.*

---

1 Note that the DCP paradigm is available in Julia through the Convex.jl package [63].
and JuMP, and then present a set of performance benchmarks.

5.1. Cheap gradients through reverse-mode AD. Reverse-mode is an AD technique which delivers gradients for the cost of $O(1)$ evaluations of $f$, a perhaps surprising result known as the cheap gradient principle. We present a basic introduction to the method as needed for our discussion and refer readers to Griewank and Walther [36] for further discussion.

Assume a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is given in the form

```plaintext
function f(x_1, x_2, \ldots, x_n)
    x_{n+1} \leftarrow g_1(x_1, x_j)
    x_{n+2} \leftarrow g_2(x_i, x_j)
    \ldots
    x_{n+r} \leftarrow g_r(x_i, x_j)
    return x_{n+r}
end function
```

That is, $f$ is computed by a sequence of $r$ “basic” operations like addition, multiplication, exponentiation, sine, cosine, etc, denoted by $g_k$. Without loss of generality, suppose each operation depends on at most two previously computed or input values whose indices are given by $i_k$ and $j_k$ i.e., $1 \leq i_k \leq n + k - 1$ and $1 \leq j_k \leq n + k - 1$.

Let $S_\ell = \{k : \ell = i_k \text{ or } \ell = j_k\}$ be the indices of operations which depend directly on $x_\ell$. Denoting $z_\ell = \frac{\partial x_\ell}{\partial x_k}$, an application of the chain rule yields

$$
\frac{\partial f(x)}{\partial x_\ell} = z_\ell = \sum_{k \in S_\ell} z_k \frac{\partial g_k(x_{i_k}, x_{j_k})}{\partial x_\ell}.
$$

Observing furthermore that $k \in S_\ell$ implies $k > \ell$, it follows that equation (5.1) can be used to compute all of the partial derivatives of interest by reversing the sequence of operations in $f$, since $z_{n+r}, z_{n+r-1}, \ldots, z_{\ell+1}$ are known.

More explicitly, the following function computes the gradient of $f$.

```plaintext
function \nabla f(x_1, x_2, \ldots, x_n)
    x_{n+1} \leftarrow g_1(x_1, x_j)
    x_{n+2} \leftarrow g_2(x_i, x_j)
    \ldots
    x_{n+r} \leftarrow g_r(x_i, x_j)
    z_{n+r} \leftarrow 1
    z_{n+r-1} \leftarrow \sum_{k \in S_{n+r-1}} z_k \frac{\partial g_k(x_{i_k}, x_{j_k})}{\partial x_{n+r-1}}
    z_{n+r-2} \leftarrow \sum_{k \in S_{n+r-2}} z_k \frac{\partial g_k(x_{i_k}, x_{j_k})}{\partial x_{n+r-2}}
    \ldots
    z_1 \leftarrow \sum_{k \in S_1} z_k \frac{\partial g_k(x_{i_k}, x_{j_k})}{\partial x_1}
    return (z_1, z_2, \ldots, z_n)
end function
```

To evaluate the computational cost of $\nabla f$, note that that $\sum_{\ell=1}^{n+r-1} |S_\ell| \leq 2r$, where $|S_\ell|$ is the number of elements in the set $S_\ell$, because the input to any particular operation can only be counted once. If we consider computation of each $g_k$ and its partial derivatives as an operation with $O(1)$ cost, we see that computing $\nabla f(x)$ requires $O(r)$ operations, which is the same complexity as computing $f(x)$ itself, up to a constant factor. Griewank [35] proves that this constant is at most 5.

On modern computing architectures, however, where computations are cheap and memory accesses expensive, the raw number of operations is not always the best
predictor of execution time. For example, a benchmark by Hogan [41] found that the
time to evaluate a gradient with ADOL-C [65] was between 9.2x and 106x greater than
the time to evaluate the function itself, while hand-coded derivatives were only 2.1x
to 3.5x slower, which more closely matches the theoretical bound. While both hand-
written derivatives and the automatic derivatives computed by ADOL-C perform the
same operations, ADOL-C manages explicit data structures to perform the operations
in an “interpreted” fashion at run time, while hand-written derivatives are plain code
which modern compilers can efficiently optimize.

A key technical feature of the implementation of reverse-mode AD in JuMP is that
the code to evaluate $\nabla f$ is generated symbolically and then compiled at runtime
to native machine instructions with Julia’s built-in compiler, all entirely transparently
to the user. The idea of generating source code for $\nabla f$ and then compiling it is
certainly not new; it is available in AMPL [27] and CasADi [2] but requires users
to manually compile and link the results. While generating and compiling code at
run time transparently to users is possible by using the LLVM compiler API [45] from
C++ (indeed, Julia uses LLVM internally), it is a substantial technical challenge which
requires intimate knowledge of the compiler’s internal structures. On the other hand
the JuMP implementation in JuMP of reverse-mode AD makes full use of Julia’s code
generation features. Our first implementation was composed of less than 500 lines of
human-readable code, which later grew to 900 lines.

Reverse-mode AD may be summarized as a procedure which transforms the code
to compute a function into code which computes its gradient. Of course, specifying a
function explicitly in the sequence of operations format is quite unnatural. Extracting
this information from arbitrary code with minimal effort by the user is a difficult
problem in general; packages like ADIFOR [12] directly examine a program’s source
code, while ADOL-C and others record the sequence of operations as they are executed
by using operator overloading.

For the purposes of nonlinear modeling, transforming closed-form algebraic ex-
pressions into a sequence of operations is more straightforward but still exposes some
design trade-offs. JuMP translates summations ($\sum$) and products ($\prod$) into explicit
loops in code, deviating from the explanation above by re-computing values inside
of a loop when they are needed, as opposed to storing the intermediate results of all
operations at a potentially large cost of memory. In some special cases, JuMP avoids
these additional computations by fusing the so-called forward and reverse passes [40].
By contrast, AMPL flattens out loops and stores the intermediate results of all oper-
ations [25]. We intend to explore this trade-off between speed and storage in future
research; for related work, see [4, 29].

Compilation has a nontrivial cost (on the order of seconds) such that it would be
impractical to generate and compile 1,000 different functions for a problem with 1,000
different constraints. JuMP takes advantage of the fact that most models have large
sets of constraints with identical algebraic structure but different data, compiling a
single function for each set of constraints. This technique is effective in reducing
the compilation time, although it makes it difficult for JuMP to perform algebraic
simplifications that depend on the data (e.g., multiplication by zero or eliminating
branches of if statements that cannot hold). The cost of compilation is again often
negligible in an amortized sense; for example, when solving a sequence of nonlinear
problems with changing data, JuMP can skip the compilation step for all but the first
problem.
Many solvers can benefit from being provided the Hessian matrix of second-order derivatives at any point. JuMP uses reverse-mode automatic differentiation to generate a “black box” routine that computes Hessian-vector products and uses this to calculate the non-zero elements of the Hessian matrix. For efficiency, we would like to use as few Hessian-vector products as possible; by using a specialized graph coloring heuristic [28], we can find a small number of evaluations to do so. Above, a symmetric $5 \times 5$ Hessian matrix with $h_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$ for some $f$. Omitted entries are known to be zero. In this example, only two Hessian-vector products are needed.

5.2. From gradients to Hessians. In addition to gradients, off-the-shelf nonlinear optimizers typically request second-order derivatives. A basic operation for computing second-order derivatives is the Hessian-vector product $\nabla^2 f(x)d$. This product is a directional derivative of the gradient, for which we now have an efficient routine to evaluate. While one could use finite differencing of $\nabla f(x)$ to compute this directional derivative, JuMP computes it without approximation error by applying the techniques of forward-mode AD, which can be interpreted as computing a directional derivative by introducing an infinitesimal perturbation [36].

One method to implement infinitesimal perturbations is to introduce a new class of number $a + be$ where $e^2 = 0$ (analogously to $i^2 = -1$ for the complex numbers). Many programming languages, including Julia, allow definition of new types of numbers with custom algebraic rules by using operator overloading, a technique discussed in Section 3. The implementation in Julia is conceptually quite similar to that in other languages, and we refer readers to Neidinger [55] for a comprehensive introduction to forward-mode AD and its implementation in MATLAB using operator overloading. In Julia, however, user-defined types are given first-class treatment by the compiler and produce efficient low-level machine code, which is not the case for MATLAB.

Given a routine to compute Hessian-vector products, a dense Hessian matrix $\nabla^2 f(x)$ can be recovered with $n$ calls to the routine, taking the $n$ distinct unit vectors. However, for large $n$, this method quickly becomes prohibitively expensive. By exploiting the sparsity structure of $\nabla^2 f(x)$, one instead may compute the entries of the Hessian matrix with far fewer than $n$ Hessian-vector products. For example, if the Hessian is known to be diagonal, one needs only a single Hessian-vector product with $d = (1, 1, \cdots, 1)^T$ to compute all nonzero elements of the Hessian. In general, the problem of choosing a minimal number of Hessian-vector products to compute all nonzero elements is $NP$-hard; we implement the acyclic graph coloring heuristic of Gebremedhin et al. [28]. See Figure 2 for an illustration. The Hessian matrices of typical nonlinear models exhibit significant sparsity, and in practice a very small number of Hessian-vector products are needed even for high-dimensional problems. We note that AMPL exploits Hessian structure through partial separability [26] instead of using graph coloring techniques.

5.3. Benchmarks. We now present benchmarks evaluating the performance of JuMP for modeling nonlinear optimization problems. Similar to the experimental design in Section 4.1, we measure the time elapsed after starting the executable until the
JuMP solver, Ipopt [64], reports the problem dimensions as confirmation that the instance is loaded in memory. Then, we fix the total number of iterations performed to three and record the time spent in function or derivative evaluations as reported by Ipopt. We evaluated the following modeling systems: JuMP, AMPL, Pyomo, GAMS, and YALMIP. Recall that CVX does not support derivative-based nonlinear models. Also, YALMIP does not support Hessian evaluations, so we measure only model generation time.

We test two families of problems, nonlinear beam control (clnlbeam) and nonlinear optimal power flow (acpower), which are further described in the appendix. For model generation times (Table 2), JuMP has a relatively large startup cost, which is dominated by the time to compile functions for derivative evaluation. However, as the size of the instance increases, JuMP becomes significantly faster than Pyomo, YALMIP, and even GAMS for the acpower model. As suggested by its performance and the omission of Hessian computations, YALMIP’s derivative-based nonlinear functionality is seemingly not designed for large-scale problems. We did not implement acpower in YALMIP.

The results in Table 3 compare the time spent evaluating derivatives. JuMP is at most 2.7x slower than AMPL, while, on the larger models, between 2.7x and 52x faster than GAMS. Note that Pyomo does not implement its own derivative computations; instead, it re-uses AMPL’s derivative evaluation library.

<table>
<thead>
<tr>
<th>Instance</th>
<th>JuMP</th>
<th>AMPL</th>
<th>GAMS</th>
<th>Pyomo</th>
<th>YALMIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>clnlbeam-5</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>117</td>
</tr>
<tr>
<td>clnlbeam-50</td>
<td>11</td>
<td>2</td>
<td>3</td>
<td>43</td>
<td>&gt;600</td>
</tr>
<tr>
<td>clnlbeam-500</td>
<td>28</td>
<td>21</td>
<td>34</td>
<td>424</td>
<td>&gt;600</td>
</tr>
<tr>
<td>acpower-1</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>acpower-10</td>
<td>28</td>
<td>1</td>
<td>6</td>
<td>26</td>
<td>-</td>
</tr>
<tr>
<td>acpower-100</td>
<td>54</td>
<td>16</td>
<td>471</td>
<td>263</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2

Time (sec.) to generate each model and pass it to the solver, a comparison between JuMP and existing commercial and open-source modeling languages for derivative-based nonlinear optimization. Dash indicates not implemented.

<table>
<thead>
<tr>
<th>Instance</th>
<th>JuMP</th>
<th>AMPL</th>
<th>GAMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>clnlbeam-5</td>
<td>0.03</td>
<td>0.03</td>
<td>0.09</td>
</tr>
<tr>
<td>clnlbeam-50</td>
<td>0.39</td>
<td>0.34</td>
<td>0.74</td>
</tr>
<tr>
<td>clnlbeam-500</td>
<td>4.72</td>
<td>3.40</td>
<td>15.69</td>
</tr>
<tr>
<td>acpower-1</td>
<td>0.08</td>
<td>0.02</td>
<td>0.19</td>
</tr>
<tr>
<td>acpower-10</td>
<td>0.81</td>
<td>0.35</td>
<td>5.07</td>
</tr>
<tr>
<td>acpower-100</td>
<td>9.28</td>
<td>3.42</td>
<td>424.89</td>
</tr>
</tbody>
</table>

Table 3

Time (sec.) to evaluate derivatives (including gradients, Jacobians, and Hessians) during 3 iterations, as reported by Ipopt. Pyomo relies on AMPL’s “solver library” for derivative evaluations, and YALMIP does not provide second-order derivatives.
6. Extensions. JuMP is designed to be extensible, allowing for developers both to plug in new solvers for existing problem classes and to extend the syntax of JuMP itself to new classes of problems. A common thread motivating extensions to an AML’s syntax is that the more natural representation of a class of models may be at a higher level than a standard-form optimization problem. These classes of models furthermore may benefit from customized solution methods which are aware of the higher-level structure. Extensions to JuMP can expose these advanced problem classes and algorithmic techniques to users who just want to solve a model and not concern themselves with the low-level details. We will present three extensions recently developed in this vein.

6.1. Extension for parallel multistage stochastic programming. The first example of a modeling extension built on top of JuMP is StochJuMP [42], a modeling layer for block-structured optimization problem of the form,

\[
\begin{align*}
\min & \quad \frac{1}{2} x_0^T Q_0 x_0 + c_0^T x_0 + \sum_{i=1}^{N} \left( \frac{1}{2} x_i^T Q_i x_i + c_i^T x_i \right) \\
\text{s.t.} & \quad A x_0 = b_0, \\
& \quad T_1 x_0 + W_1 x_1 = b_1, \\
& \quad T_2 x_0 + W_2 x_2 = b_2, \\
& \quad \vdots \quad \vdots \\
& \quad T_N x_0 + W_N x_N = b_N, \\
& \quad x_0 \geq 0, \quad x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots, \quad x_N \geq 0.
\end{align*}
\]

This structure has been well studied and arises from stochastic programming [11], contingency analysis [60], multicommodity flow [16], and many other contexts. A number of specialized methods exist for solving problems with this structure (including the classical Benders decomposition method), and they require as input data structures the matrices \(Q_i, T_i, W_i, A\), and vectors \(c_i\) and \(b_i\).

StochJuMP was motivated by the application to stochastic programming models for power systems control under uncertainty as outlined in [59]. For realistic models, the total number of variables may be in the tens to hundreds of millions, which necessitates the use of parallel computing to obtain solutions within reasonable time limits. In the context of high-performance computing, the phase of generating the model in serial can quickly become an execution bottleneck, in addition to the fact that the combined input data structures may be too large to fit in memory on a single machine. StochJuMP was designed to allow users to write JuMP models with structural annotations such that the input matrices and vectors can be generated in parallel. That is, the entire model is not built in memory in any location: each computational node only builds the portion of the model in memory that it will work with during the course of the optimization procedure. This ability to generate the model in parallel distinguishes StochJuMP from existing tools such as PySP [66].

StochJuMP successfully scaled up to 2048 cores of a high-performance cluster, and in all cases the overhead of model generation was a small fraction of the total solution time. Furthermore, StochJuMP was easy to develop, consisting of less than 500 lines of code in total, which includes interfacing with a C++-based solver and the MPI [53] library for parallel computing. For comparison, SML [37], an AMPL extension for conveying similar block structures to solvers, was implemented as a pre- and post-processor for AMPL. The implementation required reverse engineering AMPL’s syntax and developing a custom text-based parser. Such feats of software engineering are not needed to develop extensions to JuMP.
6.2. Extension for robust optimization. Robust optimization (RO) is a methodology for addressing uncertainty in optimization problems that has grown in popularity over the last decade (for a survey, see [6]). The RO approach to uncertainty models the uncertain parameters in a problem as belonging to an uncertainty set, instead of modeling them as being drawn from probability distributions. We solve an RO problem with respect to the worst-case realization of those uncertain parameters over their uncertainty set, i.e.

\[
\begin{align*}
\min_{x \in X} & \quad f(x) \\
\text{subject to} & \quad g(x, \xi) \leq 0 \quad \forall \xi \in \Xi
\end{align*}
\]

where \( x \) are the decision variables, \( \xi \) are the uncertain parameters drawn from the uncertainty set \( \Xi \), \( f : X \to \mathbb{R} \) is a function of \( x \) and \( g : X \times \Xi \to \mathbb{R}^k \) is a vector-valued function of both \( x \) and \( \xi \). Note that constraints which are not affected by uncertainty are captured by the set \( X \). As the uncertainty set \( \Xi \) is typically not a finite set of scenarios, RO problems have an infinite set of constraints. This is usually addressed by either reformulating the RO problem using duality to obtain a robust counterpart, or by using a cutting-plane method that aims to add only the subset of constraints that are required at optimality to enforce feasibility [7].

JuMPeR is an extension for JuMP that enables modeling RO problems directly by introducing the \texttt{Uncertain} modeling primitive for uncertain parameters. The syntax is essentially unchanged from JuMP, except that constraints containing only \texttt{Uncertains} and constants are treated distinctly from other constraints as they are used to define the uncertainty set. JuMPeR is then able to solve the problem by either reformulation or the cutting-plane method, allowing the user to switch between the two at will. This is an improvement over both directly modeling the robust counterpart to the RO problem and implementing a cutting-plane method, as it allows users to experiment with different uncertainty sets and solution techniques with minimal changes to their code. Building JuMPeR on top of JuMP makes it more useful than a dedicated RO modeling tool like ROME [31] as users can smoothly transition from a deterministic model to an uncertain model and can take advantage of the infrastructure developed for JuMP to utilize a wide variety of solvers.

6.3. Extension for chance constraints. Continuing with extensions for handling uncertainty, we consider chance constraints of the form

\[
\mathbb{P}(\xi^T x \leq b) \geq 1 - \epsilon,
\]

where \( x \) is a decision variable and \( \xi \) is a random variable. That is, \( x \) is feasible if and only if the random variable \( \xi^T x \) is greater than \( b \) with high probability. Depending on the distribution of \( \xi \), the constraint may be intractable and nonconvex; however, for the special case of \( \xi \) jointly Gaussian with mean \( \mu \) and covariance matrix \( \Sigma \), it is convex and representable by conic-quadratic inequalities. Bienstock et al. [10] observed that it can be advantageous to implement a custom cutting-plane algorithm similar to the case of robust optimization. The authors in [10] also examined a more conservative \textit{distributionally robust} model where we enforce that (6.3) holds for a family of Gaussian distributions where the parameters fall in some uncertainty set \( \mu \in U_\mu, \Sigma \in U_\Sigma \).

JuMPChance is an extension for JuMP which provides a natural algebraic syntax to model such chance constraints, hiding the algorithmic details of the chance
constraints from users who may be practitioners or experts in other domains. Users may declare Gaussian random variables and use them within constraints, providing $\epsilon$ though a special $\texttt{with\,probability}$ parameter. JuMPChance was used to evaluate the distributionally robust model in the context of optimal power flow under uncertainty from wind generation, finding that the increased conservatism may actually result in realized cost savings given the inaccuracy of the assumption of Gaussianity [49].

7. Interactivity and visualization. Although we have focused thus far on efficiently and intuitively communicating optimization problems to a solver, equally as important is a convenient way to interpret, understand, and communicate the solutions obtained. For many use cases, Microsoft Excel is a surprisingly versatile environment for optimization modeling [51]; one reason for its continuing success is that it is trivial to interactively manipulate the input to a problem and visualize the results, completely within Excel. Standalone commercial modeling systems, while providing a much better environment for handling larger-scale inputs and models, have in our opinion never achieved such seamless interactivity\(^2\).

Many in the scientific community are beginning to embrace the “notebook” format for both research and teaching [61]. Notebooks allow users to mix code, rich text, \LaTeX\ equations, visualizations, and interactive widgets all in one shareable document, creating compelling narratives which do not require any low-level coding to develop. Jupyter [58], in particular, contains the IJulia notebook environment for Julia and therefore JuMP as well. Taking advantage of the previously demonstrated speed of JuMP, one can easily create notebooks that embed large-scale optimization problems, which we will illustrate with two examples in this section. We believe that notebooks provide a satisfying solution in many contexts to the longstanding challenge of providing an interactive interface for optimization.

7.1. Example: Portfolio Optimization. One of the classic problems in financial optimization is the Markowitz portfolio optimization problem [50] where we seek to optimally allocate funds between $n$ assets. The problem considers the mean and variance of the return of the resulting portfolio, and seeks to find the portfolio that minimizes variance such that mean return is at least some minimal value. This is a quadratic optimization problem with linear constraints. It is natural that we would want to explore how the optimal portfolio’s variance changes as we change the minimum return: the so-called \textit{efficient frontier}.

In Figure 3 we have displayed a small notebook that solves the Markowitz portfolio optimization problem. The notebook begins with rich text describing the formulation, after which we use JuMP to succinctly express the optimization problem. The data is generated synthetically, but could be acquired from a database, spreadsheets, or even directly from the Internet. The Julia package Interact.jl [32] provides the \texttt{@manipulate} syntax, which automatically generates the minimum return slider from the definition of the \texttt{for} loop. As the user drags the slider, the model is rebuilt with the new parameter and re-solved, enabling easy, interactive experimentation. The visualization (implemented with the Gadfly [43] package) of the distribution of historical returns that would have been obtained with this optimal portfolio is also regenerated as the slider is dragged.

7.2. Example: Rocket Control. A natural goal in aerospace engineering is to maximize the altitude attained by a rocket in flight. This problem was possibly first

\(^2\)Notably, AIMMS [1], a commercial AML, enables users to create interactive graphical user interfaces on the Windows platform.
stated by Goddard [30], and has since become a standard problem in control theory, e.g. [14]. The “Goddard Rocket” optimization problem, as expressed in [18], has three state variables (altitude, velocity, and remaining mass) and one control (thrust). The rocket is affected by aerodynamic drag and gravity, and the constraints of the problem implement the equations of motion (discretized by using the trapezoidal rule).

We have implemented the optimization problem with JuMP in an IJulia notebook. Moreover we have used Interact.jl to allow the user to explore the effects of varying the maximum thrust (via $T_c$) and the coefficient that controls the relationship between altitude and drag ($h_c$). The JuMP code is omitted for the sake of brevity, but the sliders and plots of the state and control over time are displayed in Figure 4. The model is re-solved with the new parameters every time the user moves the sliders; this takes about a twentieth of a second on a laptop computer, enabling real-time interactive exploration of this complex nonlinear optimization model.

**Supplementary materials.** The benchmark instances used in Sections 4 and 5 and the notebooks presented in Section 7 are available as supplementary materials at https://github.com/mlubin/JuMPSupplement. The site http://www.juliaopt.org/ is the homepage for JuMP and other optimization-related projects in Julia.

**Acknowledgements.** We thank Paul Hovland and Jean Utke for helpful discussions on automatic differentiation. We thank Juan Pablo Vielma, Chris Maes (of Gurobi), and Victor Zverovich (of AMPL) for their comments on this manuscript which improved its presentation. This work would not be possible without the supportive community of Julia developers and users who are too many to name. We thank Carlo Baldassi, Jack Dunn, Jenny Hong, Steven G. Johnson, Tony Kelman, Dahua Lin, Karanveer Mohan, Yee Sian Ng, Elliot Saba, João Felipe Santos, Madeleine Udell, Ulf Worsøe (of Mosek), and David Zeng for significant contributions to solver interfaces in Julia. We thank the many students in the Operations Research Center at MIT who have been early adopters of JuMP. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374. M. Lubin was supported by the DOE Computational Science
Fig. 4. Visualization of the states (altitude, mass, velocity) and the control (thrust) for a rocket optimal control problem. The top set of figures is obtained for the parameters $T_c = 3.5, h_c = 500$, and the second are obtained for the parameters $T_c = 2.5, h_c = 300$, with all units normalized and dimensionless. We can see that the increased drag and reduced maximum thrust in the bottom set of figures has a substantial impact on maximum altitude and leads to a very different thrust profile.

Graduate Fellowship, which is provided under grant number DE-FG02-97ER25308.

REFERENCES

8.1. lqcp. The linear-quadratic control problem is Equation (5.2-I) from [54]. This model has a quadratic objective and linear constraints, and can be scaled by increasing the discretization (parameters $m$ and $n$) of the two-dimensional problem domain. For the purposes of benchmarking we measured the model generation time across a range of sizes, fixing $m = n$ and varying $n \in \{500, 1000, 1500, 2000\}$. In the notation below, we define $I = \{0, \ldots, m\}$ to be the index set along the first dimension and $J = \{0, \ldots, n\}$ as the index set for the second. We additionally define $I' \leftarrow I \setminus \{m\}$ and $J' \leftarrow J \setminus \{0, n\}$, with all other parameters defined as in the above reference.

\[
\begin{align*}
\min_{u,y} & \quad \frac{1}{4} \Delta_x \left( (y_{m,0} - y_0^t)^2 + 2 \sum_{j=1}^{n-1} (y_{m,j} - y_j^t)^2 + (y_{m,n} - y_n^t)^2 \right) + \\
& \quad \frac{1}{4} a \Delta_t \left( 2 \sum_{i=1}^{m-1} u_i^2 + u_m^2 \right) \\
\text{s.t.} & \quad \frac{1}{\Delta_x} (y_{i+1,j} - y_{i,j}) = \\
& \quad \frac{1}{2h_2} (y_{i,j-1} - 2y_{i,j} + y_{i,j+1} + y_{i+1,j-1} - 2y_{i+1,j} + y_{i+1,j+1}) \quad \forall i \in I', j \in J' \\
& \quad y_{0,j} = 0 \quad \forall j \in J \\
& \quad y_{i,2} - 4y_{i,1} + 3y_{i,0} = 0 \quad \forall i \in I \\
& \quad \frac{1}{2\Delta_x} (y_{i,n-2} - 4y_{i,n-1} + 3y_{i,n}) = u_i - y_{i,n} \quad \forall i \in I \\
& \quad -1 \leq u_i \leq 1 \quad \forall i \in I' \\
& \quad 0 \leq y_{i,j} \leq 1 \quad \forall i \in I, j \in J
\end{align*}
\]

8.2. fac. The fac problem is a variant on the classic facility location problem [57]: given customers (indexed by $c \in \{1, \ldots, C\}$) located at the points $x_c \in \mathbb{R}^K$, locate facilities (indexed by $f \in \{1, \ldots, F\}$) at the points $y_f \in \mathbb{R}^K$, such that the maximum distance between a customer and it’s nearest facility is minimized. This problem can be expressed most naturally in the form of a mixed-integer second-order cone problem (MISOCP), and a solved example of this problem is presented in Figure 5. We generated the problem data deterministically to enable fair comparison across the different languages: the customers are placed on a two-dimensional grid ($K = 2$) $i \in \{0, \ldots, G\}$ by $j \in \{0, \ldots, G\}$, with the points $x_c$ spaced evenly over the unit square $[0, 1]^2$. The problem size is thus parametrized by the grid size $G$ and the number of facilities $F$, with the number of variables and constraints growing proportional to $F \cdot G^2$. For the purposes of benchmarking we measured the model generation we fixed $F = G$ and varied $F \in \{25, 50, 75, 100\}$.

\[
\begin{align*}
\min_{d,y,z} & \quad d \\
\text{subject to} & \quad d \geq \|x_c - y_f\|_2 - M (1 - z_{c,f}) \quad \forall c, f \\
& \quad \sum_{f=1}^{F} z_{c,f} = 1 \quad \forall c \\
& \quad z_{c,f} \in \{0, 1\} \quad \forall c, f
\end{align*}
\]
Fig. 5. One possible optimal solution to the facility location problem with a four-by-four grid of customers (rectangles) and three facilities (circles). The dotted circles show the maximum distance between any customer and its closest facility, which is the objective.

where

\[ M = \max_{c,c'} \|x_c - x_{c'}\|_2 \]

and \( z_{c,f} \) is a binary indicator variable that is 1 if facility \( f \) is closer to customer \( c \) than any other facility, 0 otherwise.

9. Benchmark models for Section 5.3.

9.1. clnlbeam. The first model, clnlbeam, is a nonlinear beam control problem obtained from Hans Mittelmann’s AMPL-NLP benchmark set (http://plato.asu.edu/ftp/ampl-nlp.html); see also [52]. It can be scaled by increasing the discretization of the one-dimensional domain through the parameter \( n \). We test with \( n \in \{5000, 50000, 500000\} \). The model has \( 3n \) variables, \( 2n \) constraints, and diagonal Hessians. The algebraic representation follows below.

\[
\begin{align*}
\min_{t,x,u \in \mathbb{R}^{n+1}} & \quad \sum_{i=1}^{n} \left[ \frac{h}{2} (u_{i+1}^2 + u_i^2) + \frac{\alpha h}{2} (\cos(t_{i+1}) + \cos(t_i)) \right] \\
\text{subject to} & \quad x_{i+1} - x_i - \frac{1}{2n} (\sin(t_{i+1}) + \sin(t_i)) = 0 \quad i = 1, \ldots, n \\
& \quad t_{i+1} - t_i - \frac{1}{2n} u_{i+1} - \frac{1}{2n} u_i = 0 \quad i = 1, \ldots, n \\
& \quad -1 \leq t_i \leq 1, \quad -0.05 \leq x_i \leq 0.05 \quad i = 1, \ldots, n + 1 \\
& \quad x_1 = x_{n+1} = t_1 = t_{n+1} = 0.
\end{align*}
\]

9.2. acpower. The second model is a nonlinear AC power flow model published in AMPL format by KNITRO (http://www.ziena.com/elecpower.htm). The objective is to minimize active power losses

\[
(9.1) \quad \sum_k \left[ g_k + \sum_m V_k V_m (G_{km} \cos(\theta_k - \theta_m) + B_{km} \sin(\theta_k - \theta_m)) \right]^2
\]

subject to balancing both active and reactive power loads and demands at each node in the grid, where power flows are constrained by the highly nonlinear Kirchoff’s
laws. The parameter $g_k$ is the active power load (demand) at node $k$, $V_k$ is the voltage magnitude at node $k$, $\theta_k$ is the phase angle, and $Y_{km} = G_{km} + iB_{km}$ is the complex-valued admittance between nodes $k$ and $m$, which itself is a complicated nonlinear function of the decision variables. Depending on the physical characteristics of the grid, some values, like $V_k$ may be decision variables at some nodes and fixed at others. This model is quite challenging because of the combination of nonlinearity and network structure, which yields a highly structured Hessian.

We translated the AMPL model provided by KNITRO to JuMP, GAMS, and Pyomo. The base instance has a network with 662 nodes and 1017 edges; there are 1489 decision variables, 1324 constraints, and the Hessian (of the Lagrangian) has 8121 nonzero elements. We artificially enlarge the instances by duplicating the network 10-fold and 100-fold, which results in proportional increases in the problem dimensions.