JuMP 1.0: Recent improvements to a modeling language for mathematical optimization

Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, Juan Pablo Vielma

Abstract JuMP is an algebraic modeling language embedded in the Julia programming language. JuMP allows users to model optimization problems of a variety of kinds, including linear programming, integer programming, conic optimization, semidefinite programming, and nonlinear programming, and handles the low-level details of communicating with solvers. After nearly 10 years in development, JuMP 1.0 was released in March, 2022. In this short communication, we highlight the improvements to JuMP from recent releases up to and including 1.0.

1 Introduction

JuMP is an algebraic modeling language for mathematical optimization embedded in the Julia programming language [1]. JuMP supports a range of common problem classes, including linear programming, integer programming, conic optimization, semidefinite programming, and nonlinear programming,

Miles Lubin
E-mail: miles.lubin@gmail.com

Oscar Dowson
E-mail: o.dowson@gmail.com

Joaquim Dias Garcia
PSR

Joey Huchette
Google Research

Benoît Legat
KU Leuven

Juan Pablo Vielma
Google Research

and transparently handles the low-level details of communicating with the underlying solvers. Dunning et al. [6] presented JuMP as of version 0.12. Subsequently, over 90% of JuMP's code was rewritten during the transition to a new solver abstraction layer called *MathOptInterface.*

Legat et al. [10] describe the design and motivation of MathOptInterface with a focus on the lower-level details that are not visible to most users of JuMP. In this short communication, we describe key, user-visible improvements to JuMP between version 0.12 (released in February, 2016) and 1.0 (released in March, 2022). These improvements were made possible largely by the transition to MathOptInterface and other recent developments in the Julia ecosystem.

The paper is organized as a series of short sections, in which we: present a brief example demonstrating the syntax of JuMP (Section 2); discuss *direct mode*, which reduces memory overhead and allows direct access to the low-level API of a solver (Section 3); discuss how we overcome Julia's compilation latency (Section 4); update a series of benchmarks first reported in [6] (Section 5); discuss JuMP's new attribute system (Section 6); discuss solver callbacks (Section 7); and present a number of ways in which JuMP has been extended to new problem domains and applications (Section 8). We conclude in Section 9 with a brief summary of other minor—but still notable—improvements in recent releases.

2 JuMP syntax, through example

To illustrate the basic usage of JuMP, the following code snippet solves a linear regression problem with additional constraints using JuMP and the nonlinear solver Ipopt [12].

```julia
using JuMP, Ipopt
function constrained_linear_regression(A, y)
    m, n = size(A)
    model = Model(Ipopt.Optimizer)
    @variable(model, 0 <= x[1:n] <= 1)
    @variable(model, residuals[1:m])
    @constraint(model, residuals .== A * x .- y)
    @constraint(model, sum(x) == 1)
    @objective(model, Min, sum(residuals[i]^2 for i in 1:m))
    optimize!(model)
    print(solution_summary(model))
    return value.(x)
end
A, y = rand(30, 20), rand(30)
x = constrained_linear_regression(A, y)
```

The macro syntax (lines beginning with @) for creating variables, adding constraints, and the setting the objective has not changed substantially since it was described by Dunning et al. [6]. However, the API for setting the solver, solving, and querying the results differs following the transition to MathOptInterface (between JuMP version 0.18 and 0.19).

3 Direct mode

In its default usage, JuMP stores its own copy of the problem data, which enables JuMP to perform problem modification and allows the solver to be changed at any time. This design is the most typical choice for modeling languages, we presume because it simplifies the task of abstraction across solvers.

To provide an option to reduce peak memory usage and improve performance, JuMP 1.0 includes `direct_model`, a mode in which JuMP acts as a stateless wrapper around the solver, passing all problem data directly to the solver without any intermediate caches or transformations. `direct_model` also provides a safe way to interleave JuMP code with direct access to the underlying API of the solver. For example, we can call the `Highs_scaleCol` function of HiGHS's [8] C API as follows:

```julia
using JuMP, HiGHS
model = direct_model(HiGHS.Optimizer())
@variable(model, x)
#  An object that acts as a pointer to the highs problem
highs = backend(model)
#  0-indexed column of x in highs
col = HiGHS.column(highs, optimizer_index(x))
#  Call the C API
Highs_scaleCol(highs, col, 0.1)
```

Direct mode is not enabled by default because some solvers do not support the incremental modifications required to build the model in direct mode (e.g., they may be one-shot solvers that require all problem data in a single function call). Moreover, in direct mode, the user cannot change the solver associated with a model, and reformulations—particularly those associated with conic models—are disabled. In general, we suggest that users consider direct mode only if benchmarking demonstrates that model construction is a bottleneck in their application and only if they are using a solver that supports incremental modification.

4 Compilation latency

Julia was designed to be a "fast" language, but we find that many first-time users complain of a sluggish experience, particularly when running code from the command line or during the first invocation of a function during an interactive session. The reason is compilation latency due to Julia's Just-In-Time (JIT) compilation model.

As an example of this latency, consider the following linear program with two variables and two constraints:

```julia
using JuMP, HiGHS
model = Model(HiGHS.Optimizer)
```

```
@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)
@objective(model, Min, 12x + 20y)
@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)
optimize!(model)
open("model.log", "w") do io
    print(io, solution_summary(model; verbose = true))
end
```

Saving the problem in `model.jl` and calling from the command line results in:

```
$ time julia model.jl
15.78s user 0.48s system 100% cpu 16.173 total
```

Clearly, 16 seconds is a large overhead to pay for solving this trivial model. However, the compilation latency is independent of the problem size, and so 16 seconds of additional overhead may be tolerable for larger models that take minutes or hours to solve.

The recently matured `PackageCompiler.jl` package provides a work-around for Julia's compilation latency by generating a custom *sysimage*, a binary extension to Julia that caches compiled code. A custom image for our problem can be created as follows:

```
using PackageCompiler, Libdl
PackageCompiler.create_sysimage(
    ["JuMP", "HiGHS"],
    sysimage_path = "customimage." * Libdl.dlext,
    precompile_execution_file = "model.jl",
)
```

When Julia is run with the custom image, the run time is now 0.7 seconds instead of 16:

```
$ time julia --sysimage customimage model.jl
0.68s user 0.22s system 153% cpu 0.587 total
```

Note that `create_sysimage` only needs to be run once, and because the sysimage caches an intermediate representation of the code that is independent of the run-time data, the same sysimage can be used—to a slight detriment of performance—even if we modify `model.jl` or run a different file.

5 Benchmarks

In this section we update a selection of benchmarks from the previous JuMP paper [6]. The results are given in Table 1. `Model` is the default way of constructing a JuMP model, and, as explained in Section 3, `direct_model` is a

new feature of JuMP that skips an additional cache of the problem. `Pyomo` is a popular open-source algebraic modeling language in Python [2], and `GRB/C++` is the C++ interface to Gurobi [7] based on operator overloading.

We used Julia v1.6.2, `JuMP.jl` v1.0.0, `Gurobi.jl` v0.11.2, Gurobi v9.5.1, Python 3.8 and Pyomo v6.4.0. To remove compilation latency, we use a custom sysimage, as described in Section 4. The benchmarks were run on a 2018 Macbook Pro with 16 GB of RAM. All code to run the benchmarks is available at `https://github.com/jump-dev/JuMPPaperBenchmarks`.

Model	# Variables	JuMP		Pyomo	GRB/C++
		direct_model	Model		
fac-25	67,651	1	1	4	0
fac-50	520,301	4	7	34	3
fac-75	1,732,951	14	20	128	8
fac-100	4,080,601	37	50	338	20
lqcp-500	251,501	2	1	24	3
lqcp-1000	1,003,001	7	11	107	10
lqcp-1500	2,254,501	17	23	234	23
lqcp-2000	4,006,001	32	50	417	40

Table 1 Time (sec.) to generate each model and pass it to the solver. Models use Gurobi and terminate after a time limit of 0.0 seconds. `direct_model` uses the direct mode feature discussed in 3, and `Model` is the default way of constructing a JuMP model.

Our results are broadly consistent with those reported in Dunning et al. [6] and Jusevičius et al. [9][1]. JuMP is faster than Pyomo, especially on larger models, and similar in performance to the C++ interface to Gurobi. JuMP in `direct_model` is faster than `GRB/C++` for the `lqcp` family of models, likely due to a more efficient implementation of the way in which JuMP builds quadratic expressions.

6 Solver attributes

Prior to switching to MathOptInterface, JuMP treated solver parameters non-uniformly across solvers, and supported only a fixed set of variable and constraint attributes. In JuMP 1.0, options can be bundled with a solver constructor and changed after construction using `set_optimizer_attribute`:

```
optimizer = optimizer_with_attributes(Ipopt.Optimizer, "max_iter" => 3)
model = Model(optimizer)
set_optimizer_attribute(model, "max_iter", 10)
```

---

[1] This is after accounting for the fact that Jusevičius et al. [9] include compilation latency for JuMP and tested the time taken to write a JuMP model to file instead of the time taken for the solver to start performing useful work.

This flexibility is particularly useful for meta-solvers (solvers that call other solvers), because it allows multiple, independent copies of a solver with different parameter settings to be created from the same input, and also because it unifies the way in which meta-solvers can pass parameters from the user through to their inner solvers.

In addition to a number of solver-independent attributes (including starting points, verbosity, and time limits), the attribute system is extensible, allowing users to access solver-specific attributes:

```
model = Model(Gurobi.Optimizer)
@variable(model, x)
MOI.set(model, Gurobi.VariableAttribute("BranchPriority"), x, 1.0)
```

7 Callbacks

Earlier versions of JuMP supported a range of solver-independent callbacks, which let users write custom functions that interact with mixed-integer linear programming solvers during the solution process. Despite the promised solver independence of the callbacks, subtle implementation differences between solvers led to different observed behavior and meant that the callbacks were in fact closely tied to the solver. This made it hard to write meta-solvers that utilized solver-independent callbacks. In JuMP 1.0, we reduced the scope of solver-independent callbacks to a narrower subset of functionality that is more uniformly implemented by the solvers in order to increase robustness and consistency across solvers.

JuMP 1.0 supports three types of solver-independent callbacks: a lazy constraint callback, which allows the user to add new constraints at nodes in the branch-and-bound tree; a user-cut callback, which allows the user to add cuts which tighten the continuous relaxation at a node without removing any integer feasible solutions; and a heuristic callback, which allows the user to inject an integer feasible point. An example of a lazy constraint callback is as follows:

```
using JuMP, GLPK
model = Model()
@variable(model, 0 <= x[1:2] <= 2.5, Int)
@objective(model, Max, x[2])
function my_callback_function(cb_data)
    if callback_node_status(cb_data, model) != MOI.CALLBACK_NODE_STATUS_INTEGER
        return
    end
    x_val = callback_value.(cb_data, x)
    if x_val[2] - x_val[1] > 1 + 1e-6
        con = @build_constraint(x[2] - x[1] <= 1)
        MOI.submit(model, MOI.LazyConstraint(cb_data), con)
    elseif x_val[2] + x_val[1] > 3 + 1e-6
        con = @build_constraint(x[2] + x[1] <= 3)
        MOI.submit(model, MOI.LazyConstraint(cb_data), con)
    end
    return
```

```
end
MOI.set(model, MOI.LazyConstraintCallback(), my_callback_function)
# Would also work with CPLEX.Optimizer, Gurobi.Optimizer, etc.
set_optimizer(model, GLPK.Optimizer)
optimize!(model)
```

In addition to solver-independent callbacks, access to the low-level API of a solver allows the user to write solver-*dependent* callbacks. For example, here we add a callback to GLPK which terminates the solve when an integer-feasible solution is detected:

```
using JuMP, GLPK
model = Model(GLPK.Optimizer)
@variable(model, 0 <= x[1:2] <= 2.5, Int)
@objective(model, Max, 1.0 * x[2])
function my_callback_function(cb_data)
    if GLPK.glp_ios_reason(cb_data.tree) == GLPK.GLP_IBINGO
        GLPK.glp_ios_terminate(cb_data.tree)
    end
    return
end
MOI.set(model, GLPK.CallbackFunction(), my_callback_function)
optimize!(model)
```

8 Extensions

A core design principle of JuMP was that it should be extensible by external packages. Therefore, JuMP exposes a number of code hooks that let users extend JuMP to new problem domains and syntax. One example is the UnitJuMP.jl[2] package, which adds unit information to JuMP variables and automatically scales terms in expressions to have the same units:

```
using UnitJuMP, HiGHS
model = Model(HiGHS.Optimizer)
@variable(model, x >= 0, u"m/s")
@variable(model, y >= 0, u"ft/s")
@constraint(model, x + y <= 60u"km/hr", u"km/hr")
@constraint(model, x <= 0.5y)
@objective(model, Max, x + y)
optimize!(model)
```

JuMP extensions can also combine multiple JuMP models in novel ways. For example, the BilevelJuMP.jl [4] package lets users formulate and solve bilevel optimization problems using JuMP's syntax:

```
using JuMP, BilevelJuMP, Gurobi
model = BilevelModel(Gurobi.Optimizer, mode = BilevelJuMP.SOS1Mode())
@variable(Lower(model), x)
@variable(Upper(model), y)
```

---

[2] https://github.com/trulsf/UnitJuMP.jl

```
@objective(Upper(model), Min, 3x + y)
@constraint(Upper(model), x <= 5)
@constraint(Upper(model), y <= 8)
@constraint(Upper(model), y >= 0)
@objective(Lower(model), Min, -x)
@constraints(Lower(model), begin
    x +  y <= 8
   4x +  y >= 8
   2x +  y <= 13
   2x - 7y <= 0
end)
optimize!(model)
```

Other popular JuMP extensions include: `InfiniteOpt.jl`, for problems with infinite-dimensional variables like continuous time and integrals [11]; `PowerModels.jl`, for solving steady-state power network optimization problems [3]; and `SDDP.jl`, for multistage stochastic programming [5].

9 Additional improvements

We conclude by noting additional improvements added between v0.12 and v1.0 that are minor in scope but that address longstanding feature requests.

JuMP 1.0 added the ability to delete variables and constraints, and the ability to modify the variable coefficients of linear and quadratic constraints. In addition, JuMP 1.0 provides the ability to query an irreducible inconsistent subset (IIS) of an infeasible model (if supported by the solver) and lets users access multiple primal solutions from a mixed-integer programming solver.

The number of supported solvers, as tracked by the JuMP documentation, increased from 13 to 41, and, in addition, JuMP supports any AMPL- or GAMS-compatible solver through `AmplNLWriter.jl` and `GAMS.jl`. All supported C/C++ open-source solvers are now automatically built for multiple platforms and the compiled binaries are stored online. Solver binaries are downloaded when the solver wrapper package is added by the Julia package manager. This change simplified the installation of solvers by users, greatly reducing the number of complaints of installation issues.

The change to MathOptInterface enabled JuMP to easily support a much wider set of constraint types as well as user-defined constraint sets. Two new built-in, non-trivial constraint types are indicator and complementarity constraints, which some solvers natively support. An indicator constraint has the form $z \Rightarrow \{a^\top x \leq c\}$; if the binary variable z is true, then the inequality inside $\{\}$ holds. A complementarity constraint has the form $f(x) \perp x_i$. If x_i is at its lower bound, then $f(x) \geq 0$; if x_i is at its upper bound, then $f(x) \leq 0$; else $f(x) = 0$. We also added support for generalized inequalities, in which $X \geq Y, \mathcal{S}$ is equivalent to $X - Y \in \mathcal{S}$.

```
model = Model()
@variable(model, x[1:2] >= 0)
@variable(model, z, Bin)
```

```
# Indicator constraints
@constraint(model, z => {sum(x[i] for i in 1:2) >= 1})

# Complementarity constraints
@constraint(model, complements(2x[1] + 1, x[2]))

# Generalized inequalities
X, Y = [x[1] x[2]; x[2] x[1]], [1 0; 0 1]
@constraint(model, X >= Y, PSDCone())
```

The transition to MathOptInterface also improved JuMP status reporting. Early version of JuMP relied on a free-form `Symbol` with some generic values and other solver-dependent values. Statuses are now reported using in a more unified, robust and expressive form using MathOptInterface's enums describing the status of primal and dual results and the termination status.

Finally, we have greatly improved our documentation, which now has a wide range of how-to, reference, and tutorial materials. We recommend users get started by going to `https://jump.dev`.

Acknowledgements

References

1. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A Fresh Approach to Numerical Computing. SIAM Review **59**(1), 65–98 (2017)
2. Bynum, M.L., Hackebeil, G.A., Hart, W.E., Laird, C.D., Nicholson, B.L., Siirola, J.D., Watson, J.P., Woodruff, D.L.: Pyomo – optimization modeling in Python, vol. 67. Springer, Berlin (2021)
3. Coffrin, C., Bent, R., Sundar, K., Ng, Y., Lubin, M.: PowerModels.jl: An open-source framework for exploring power flow formulations. In: 2018 Power Systems Computation Conference (PSCC), pp. 1–8. IEEE (2018)
4. Dias Garcia, J., Bodin, G., Street, A.: BilevelJuMP.jl: Modeling and Solving Bilevel Optimization in Julia. arXiv preprint (2022). DOI 10.48550/arXiv.2205.02307
5. Dowson, O., Kapelevich, L.: SDDP.jl: a julia package for stochastic dual dynamic programming. INFORMS Journal on Computing **33**(1), 27–33 (2021)
6. Dunning, I., Huchette, J., Lubin, M.: JuMP: A Modeling Language for Mathematical Optimization. SIAM Review **59**(2), 295–320 (2017)
7. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2023). URL `https://www.gurobi.com`
8. Huangfu, Q., Hall, J.A.J.: Parallelizing the dual revised simplex method. Mathematical Programming Computation **10**(1), 119–142 (2018)
9. Jusevičius, V., Oberdieck, R., Paulavičius, R.: Experimental analysis of algebraic modelling languages for mathematical optimization. Informatica **32**(2), 283–304 (2021)
10. Legat, B., Dowson, O., Dias Garcia, J., Lubin, M.: MathOptInterface: a data structure for mathematical optimization problems. INFORMS Journal on Computing **34**(2), 672–689 (2021)

11. Pulsipher, J.L., Zhang, W., Hongisto, T.J., Zavala, V.M.: A unifying modeling abstraction for infinite-dimensional optimization. Computers & Chemical Engineering **156**, 107567 (2022)
12. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical programming **106**(1), 25–57 (2006)

Ethics declarations

Funding

Conflict of interest

The authors declare that they have no conflict of interest.

Availability of data and materials

All data analyzed during this study are publicly available. URLs are included in this published article.

Code availability

All software written by the authors for this article is open source. All third-party software used in this article is open source, with the exception of Gurobi, which is commercial software available for academic use. Specific references are included in this published article.