JuMP.jl Documentation

Release 0.17

Miles Lubin, Iain Dunning, and Joey Huchette

Jun 14, 2017

Contents

JuMP is a domain-specific modeling language for mathematical optimization embedded in Julia. It currently supports a number of open-source and commercial solvers (see below) for a variety of problem classes, including **linear programming**, **mixed-integer programming**, **second-order conic programming**, **semidefinite programming**, and **nonlinear programming**. JuMP's features include:

- User friendliness
  - Syntax that mimics natural mathematical expressions.
  - Complete documentation.
- Speed
  - Benchmarking has shown that JuMP can create problems at similar speeds to special-purpose modeling languages such as AMPL.
  - JuMP communicates with solvers in memory, avoiding the need to write intermediary files.
- Solver independence
  - JuMP uses a generic solver-independent interface provided by the MathProgBase package, making it easy to change between a number of open-source and commercial optimization software packages ("solvers").
  - Currently supported solvers include Artelys Knitro, Bonmin, Cbc, Clp, Couenne, CPLEX, ECOS, FICO Xpress, GLPK, Gurobi, Ipopt, MOSEK, NLopt, and SCS.
- Access to advanced algorithmic techniques
  - Including *efficient LP re-solves* and *callbacks for mixed-integer programming* which previously required using solver-specific and/or low-level C++ libraries.
- Ease of embedding
  - JuMP itself is written purely in Julia. Solvers are the only binary dependencies.
  - Being embedded in a general-purpose programming language makes it easy to solve optimization problems as part of a larger workflow (e.g., inside a simulation, behind a web server, or as a subproblem in a decomposition algorithm).
    - As a trade-off, JuMP's syntax is constrained by the syntax available in Julia.
  - JuMP is MPL licensed, meaning that it can be embedded in commercial software that complies with the terms of the license.

While neither Julia nor JuMP have reached version 1.0 yet, the releases are stable enough for everyday use and are being used in a number of research projects and neat applications by a growing community of users who are early adopters. JuMP remains under active development, and we welcome your feedback, suggestions, and bug reports.

CHAPTER 1

Installing JuMP

If you are familiar with Julia you can get started quickly by using the package manager to install JuMP:

```
julia> Pkg.add("JuMP")
```

And a solver, e.g.:

```
julia> Pkg.add("Clp")  # Will install Cbc as well
```

Then read the *Quick Start Guide* and/or see a *Simple Example*. The subsequent sections detail the complete functionality of JuMP.

CHAPTER 2

Contents

Installation Guide

This guide will briefly guide you through installing Julia, JuMP and[a] solver[s] of your choice.

Getting Julia

At the time of writing this documentation the latest release of Julia is version `0.5`, which is the version required by JuMP. You can easily build from source on OS X and Linux, but the binaries will work well for most people.

Download links and more detailed instructions are available on the Julia website.

Getting JuMP

Once you've installed Julia, installing JuMP is simple. Julia has a git-based package system. To use it, open Julia in interactive mode (i.e. `julia` at the command line) and use the package manager:

```
julia> Pkg.add("JuMP")
```

This command checks METADATA.jl to determine what the most recent version of JuMP is and then downloads it from its repository on GitHub.

To start using JuMP (after installing a solver), it should be imported into the local scope:

```
julia> using JuMP
```

Getting Solvers

Solver support in Julia is currently provided by writing a solver-specific package that provides a very thin wrapper around the solver's C interface and providing a standard interface that JuMP can call. If you are interested in providing an interface to your solver, please

get in touch. The table below lists the currently supported solvers and their capabilities.

Solver	Julia Package	`solver=`	License	LP	SOCP	MILP	NLP	MINLP	SDP
Artelys Knitro	KNITRO.jl	`KnitroSolver()`	Comm.				X	X	
BARON	BARON.jl	`BaronSolver()`	Comm.				X	X	
Bonmin	AmplNLWriter.jl	`BonminNLSolver()` *	EPL	X		X	X	X	
	CoinOptServices.jl	`OsilBonminSolver()`							
Cbc	Cbc.jl	`CbcSolver()`	EPL			X			
Clp	Clp.jl	`ClpSolver()`	EPL	X					
Couenne	AmplNLWriter.jl	`CouenneNLSolver()` *	EPL	X		X	X	X	
	CoinOptServices.jl	`OsilCouenneSolver()`							
CPLEX	CPLEX.jl	`CplexSolver()`	Comm.	X	X	X			
ECOS	ECOS.jl	`ECOSSolver()`	GPL	X	X				
FICO Xpress	Xpress.jl	`XpressSolver()`	Comm.	X	X	X			
GLPK	GLPKMath...	`GLPKSolver[LP|MIP]()`	GPL	X		X			
Gurobi	Gurobi.jl	`GurobiSolver()`	Comm.	X	X	X			
Ipopt	Ipopt.jl	`IpoptSolver()`	EPL	X			X		
MOSEK	Mosek.jl	`MosekSolver()`	Comm.	X	X	X	X		X
NLopt	NLopt.jl	`NLoptSolver()`	LGPL				X		
SCS	SCS.jl	`SCSSolver()`	MIT	X	X				X

Where:

- LP = Linear programming
- SOCP = Second-order conic programming (including problems with convex quadratic constraints and/or objective)
- MILP = Mixed-integer linear programming
- NLP = Nonlinear programming
- MINLP = Mixed-integer nonlinear programming
- SDP = Semidefinite programming

* requires CoinOptServices installed, see below.

To install Gurobi, for example, and use it with a JuMP model `m`, run:

```
Pkg.add("Gurobi")
using JuMP
using Gurobi

m = Model(solver=GurobiSolver())
```

Setting solver options is discussed in the *Model* section.

Solver-specific notes follow below.

Artelys Knitro

Requires a license. The KNITRO.jl interface currently supports only nonlinear problems.

BARON

Requires a license. A trial version is available for small problem instances.

COIN-OR Clp and Cbc

Binaries for Clp and Cbc are provided on OS X and Windows (32- and 64-bit) by default. On Linux, they will be compiled from source (be sure to have a C++ compiler installed). Cbc supports "SOS" constraints but does *not* support MIP callbacks.

CPLEX

Requires a working installation of CPLEX with a license (free for faculty members and graduate teaching assistants). The interface requires using CPLEX as a shared library, which is unsupported by the CPLEX developers. Special installation steps are required on OS X. CPLEX supports MIP callbacks and "SOS" constraints.

ECOS

ECOS can be used by JuMP to solve LPs and SOCPs. ECOS does not support general quadratic objectives or constraints, only second-order conic constraints specified by using `norm` or the quadratic form `x'x <= y^2`.

FICO Xpress

Requires a working installation of Xpress with an active license (it is possible to get license for academic use, see FICO Academic Partner Program). Supports SOCP and "SOS" constraints. The interface is experimental, but it does pass all JuMP and MathProgBase tests. Callbacks are not yet supported.

> **Warning:** If you are using 64-bit Xpress, you must use 64-bit Julia (and similarly with 32-bit Xpress).

GLPK

GLPK binaries are provided on OS X and Windows (32- and 64-bit) by default. On Linux, it will be compiled from source. Note that `GLPKSolverLP` should be used for continuous problems and `GLPKSolverMIP` for problems with integer variables. GLPK supports MIP callbacks but does not support "SOS" constraints.

Gurobi

Requires a working installation of Gurobi with an activated license (free for academic use). Gurobi supports MIP callbacks and "SOS" constraints.

> **Warning:** If you are using 64-bit Gurobi, you must use 64-bit Julia (and similarly with 32-bit Gurobi).

Ipopt

Ipopt binaries are provided on OS X and Windows (32- and 64-bit) by default. On Linux, it will be compiled from source. The default installation of Ipopt uses the open-source MUMPS library for sparse linear algebra. Significant speedups can be obtained by manually compiling Ipopt to use proprietary sparse linear algebra libraries instead. Julia can be pointed to use a custom version of Ipopt; we suggest posting to the julia-opt mailing list with your platform details for guidance on how to do this.

MOSEK

Requires a license (free for academic use). Mosek does not support the MIP callbacks used in JuMP. For nonlinear optimization, Mosek supports only convex problems. The Mosek interface is maintained by the Mosek team. (Thanks!)

NLopt

NLopt supports only nonlinear models. An algorithm must be specified as an option when using `NLoptSolver`. NLopt is not recommended for large-scale models, because it does not currently exploit sparsity of derivative matrices.

SCS

SCS can be used by JuMP to solve LPs and SOCPs, and SDPs. SCS is a first order solver and has low accuracy (10^{-4}) by default; see the SCS.jl documentation for more information.

COIN-OR Bonmin and Couenne

Binaries of Bonmin and Couenne are provided on OS X and Windows (32- and 64-bit) by the CoinOptServices.jl package. On Linux, they will be compiled from source. Once installed, they can be called either via `.osil` files using `OsilBonminSolver` and `OsilCouenneSolver` from CoinOptServices.jl, or via `.nl` files using `BonminNLSolver` and `CouenneNLSolver` from AmplNLWriter.jl. We recommend using the `.nl` format option, which is currently more stable and has better performance for derivative computations. Since both Bonmin and Couenne use Ipopt for continuous subproblems, the same MUMPS sparse linear algebra performance caveat applies.

Other AMPL-compatible solvers

Any other solver not listed above that can be called from AMPL can be used by JuMP through the AmplNLWriter.jl package. The first argument to `AmplNLSolver` can be used to specify a solver executable name.

For example, SCIP is a powerful noncommercial mixed-integer programming solver. To use SCIP within JuMP, you must first download and compile SCIP with support for AMPL. Then you may use `AmplNLSolver("/path/to/scipampl")` where `scipampl` is the executable produced from the compilation process.

Quick Start Guide

This quick start guide will introduce the main concepts of JuMP. If you are familiar with another modeling language embedded in a high-level language such as PuLP (Python) or a solver-specific interface you will find most of this familiar, with the exception of *macros*. A deep understanding of macros is not essential, but if you would like to know more please see the Julia documentation. If you are coming from an AMPL or similar background, you may find some of the concepts novel but the general appearance will still be familiar.

Creating a Model

Models are Julia objects. They are created by calling the constructor:

```
m = Model()
```

All variables and constraints are associated with a `Model` object. Usually, you'll also want to provide a solver object here by using the `solver=` keyword argument; see the simple example below. For a list of all functions related to `Model`, see *Models*.

Defining Variables

Variables are also Julia objects, and are defined using the `@variable` macro. The first argument will always be the `Model` to associate this variable with. In the examples below we assume `m` is already defined. The second argument is an expression that declares the variable name and optionally allows specification of lower and upper bounds. For example:

```
@variable(m, x )               # No bounds
@variable(m, x >= lb )         # Lower bound only (note: 'lb <= x' is not valid)
@variable(m, x <= ub )         # Upper bound only
@variable(m, lb <= x <= ub )   # Lower and upper bounds
```

All these variations introduce a new variable `x` in the local scope. The names of your variables must be valid Julia variable names. For information about common operations on variables, e.g. changing their bounds, see the *Variables* section.

Integer and **binary** restrictions can optionally be specified with a third argument, `Int` or `Bin`.

To create arrays of variables we append brackets to the variable name. For example:

```
@variable(m, x[1:M,1:N] >= 0 )
```

will create an `M` by `N` array of variables. Both ranges and arbitrary iterable sets are supported as index sets. Currently we only support ranges of the form `a:b` where `a` is an explicit integer, not a variable. Using ranges will generally be faster than using arbitrary symbols. You can mix both ranges and lists of symbols, as in the following example:

```
s = ["Green", "Blue"]
@variable(m, x[-10:10,s], Int )
# e.g. x[-4, "Green"]
```

Finally, bounds can depend on variable indices:

```
@variable(m, x[i=1:10] >= i )
```

Objective and Constraints

JuMP allows users to use a natural notation to describe linear expressions. To add constraints, use the `@constraint()` and `@objective()` macros, e.g.:

```
@constraint(m, x[i] - s[i] <= 0)  # Other options: == and >=
@constraint(m, sum(x[i] for i=1:numLocation) == 1)
@objective(m, Max, 5x + 22y + (x+y)/2) # or Min
```

Note: The `sense` passed to `@objective` must be a symbol type: `:Min` or `:Max`, although the macro accepts `:Min` and `:Max`, as well as `Min` and `Max` (without the colon) directly.

The `sum()` syntax directly follows Julia's own generator expression syntax. You may use conditions within sums, e.g.:

```
sum(expression for i = I1, j = I2 if cond)
```

which is equivalent to:

```
a = zero(AffExpr)
for i = I1
    for j = I2
        ...
        if cond
            a += expression
        end
        ...
    end
end
```

Note: JuMP previously used a special curly brace syntax for `sum{}`, `prod{}`, and `norm2{}`. This has been entirely replaced by `sum()`, `prod()`, and `norm()` since Julia 0.5. The curly brace syntax is deprecated and will be removed in a future release.

Simple Example

In this section we will construct a simple model and explain every step along the way. The are more complex examples in the `JuMP/examples/` folder. Here is the code we will walk through:

```
using JuMP
using Clp

m = Model(solver = ClpSolver())
@variable(m, 0 <= x <= 2 )
@variable(m, 0 <= y <= 30 )

@objective(m, Max, 5x + 3*y )
@constraint(m, 1x + 5y <= 3.0 )

print(m)

status = solve(m)

println("Objective value: ", getobjectivevalue(m))
println("x = ", getvalue(x))
println("y = ", getvalue(y))
```

Once JuMP is *installed*, to use JuMP in your programs, you just need to say:

```
using JuMP
```

We also need to include a Julia package which provides an appropriate solver. In this case, we'll use Clp:

```
using Clp
```

Models are created with the `Model()` function. The `solver=` keyword argument is used to specify the solver to be used:

```
m = Model(solver = ClpSolver())
```

Note: Your model doesn't have to be called m - it's just a name.

There are a few options for defining a variable, depending on whether you want to have lower bounds, upper bounds, both bounds, or even no bounds. The following commands will create two variables, `x` and `y`, with both lower and upper bounds. Note the first argument is our model variable `m`. These variables are associated with this model and cannot be used in another model.:

```
@variable(m, 0 <= x <= 2 )
@variable(m, 0 <= y <= 30 )
```

Next we'll set our objective. Note again the `m`, so we know which model's objective we are setting! The objective sense, `Max` or `Min`, should be provided as the second argument. Note also that we don't have a multiplication `*` symbol between 5 and our variable `x` - Julia is smart enough to not need it! Feel free to stick with `*` if it makes you feel more comfortable, as we have done with `3*y`:

```
@objective(m, Max, 5x + 3*y )
```

Adding constraints is a lot like setting the objective. Here we create a less-than-or-equal-to constraint using <=, but we can also create equality constraints using == and greater-than-or-equal-to constraints with >=:

```
@constraint(m, 1x + 5y <= 3.0 )
```

If you want to see what your model looks like in a human-readable format, the `print` function is defined for models.

```
print(m)
```

Models are solved with the `solve()` function. This function will not raise an error if your model is infeasible - instead it will return a flag. In this case, the model is feasible so the value of `status` will be `:Optimal`, where `:` again denotes a symbol. The possible values of `status` are described *here*.

```
status = solve(m)
```

Finally, we can access the results of our optimization. Getting the objective value is simple:

```
println("Objective value: ", getobjectivevalue(m))
```

To get the value from a variable, we call the `getvalue()` function. If `x` is not a single variable, but instead a range of variables, `getvalue()` will return a list. In this case, however, it will just return a single value.

```
println("x = ", getvalue(x))
println("y = ", getvalue(y))
```

Models

Constructor

`Model` is a type defined by JuMP. All variables and constraints are associated with a `Model` object. It has a constructor that has no required arguments:

```
m = Model()
```

The constructor also accepts an optional keyword argument, `solver`. You may specify a solver either here or later on by calling `setsolver`. JuMP will throw an error if you try to solve a problem without specifying a solver.

`solver` must be an `AbstractMathProgSolver` object, which is constructed as follows:

```
solver = solvername(Option1=Value1, Option2=Value2, ...)
```

where `solvername` is one of the supported solvers. See the *solver table* for the list of available solvers and corresponding parameter names. All options are solver-dependent; see corresponding solver packages for more information.

Note: Be sure that the solver provided supports the problem class of the model. For example `ClpSolver` and `GLPKSolverLP` support only linear programming problems. `CbcSolver` and `GLPKSolverMIP` support only mixed-integer programming problems.

As an example, we can create a `Model` object that will use GLPK's exact solver for LPs as follows:

```
m = Model(solver = GLPKSolverLP(method=:Exact))
```

Methods

General

- `MathProgBase.numvar(m::Model)` - returns the number of variables associated with the `Model m`.
- `MathProgBase.numlinconstr(m::Model)` - returns the number of linear constraints associated with the `Model m`.
- `MathProgBase.numquadconstr(m::Model)` - returns the number of quadratic constraints associated with the `Model m`.
- `JuMP.numsocconstr(m::Model)` - returns the number of second order cone constraints associated with the `Model m`.
- `JuMP.numsosconstr(m::Model)` - returns the number of sos constraints associated with the `Model m`.
- `JuMP.numsdconstr(m::Model)` - returns the number of semi-definite constraints associated with the `Model m`.
- `JuMP.numnlconstr(m::Model)` - returns the number of nonlinear constraints associated with the `Model m`.
- `MathProgBase.numconstr(m::Model)` - returns the total number of constraints associated with the `Model m`.
- `getsolvetime(m::Model)` - returns the solve time reported by the solver if it is implemented.
- `getnodecount(m::Model)` - returns the number of explored branch-and-bound nodes, if it is implemented.

- `getobjbound(m::Model)` - returns the best known bound on the optimal objective value. This is used, for example, when a branch-and-bound method is stopped before finishing.
- `getobjgap(m::Model)` - returns the final relative optimality gap as optimization terminated. That is, it returns $\frac{|b-f|}{|f|}$, where b is the best bound and f is the best feasible objective value.
- `getrawsolver(m::Model)` - returns an object that may be used to access a solver-specific API.
- `getsimplexiter(m::Model)` - returns the cumulative number of simplex iterations during the optimization process. In particular, for a MIP it returns the total simplex iterations for all nodes.
- `getbarrieriter(m::Model)` - returns the cumulative number of barrier iterations during the optimization process.
- `internalmodel(m::Model)` - returns the internal low-level `AbstractMathProgModel` object which can be used to access any functionality that is not exposed by JuMP. See the MathProgBase documentation.
- `solve(m::Model; suppress_warnings=false, relaxation=false)` - solves the model using the selected solver (or a default for the problem class), and takes two optional arguments that are disabled by default. Setting `suppress_warnings` to `true` will suppress all JuMP-specific output (e.g. warnings about infeasibility and lack of dual information) but will not suppress solver output (which should be done by passing options to the solver). Setting `relaxation=true` solves the standard continuous relaxation for the model: that is, integrality is dropped, special ordered set constraints are not enforced, and semi-continuous and semi-integer variables with bounds `[l,u]` are replaced with bounds `[min(l,0),max(u,0)]`.
- `JuMP.build(m::Model)` - builds the model in memory at the MathProgBase level without optimizing.
- `setsolver(m::Model,s::AbstractMathProgSolver)` - changes the solver which will be used for the next call to `solve()`, discarding the current internal model if present.
- `getindex(m::Model,name::Symbol)` - returns the variable, or group of variables, or constraint, or group of constraints, of the given name which were added to the model. This errors if multiple variables or constraints share the same name.
- `setindex!(m::Model, value, name::Symbol)` - stores the object `value` in the model `m` so that it can be accessed via `getindex`.

Objective

- `getobjective(m::Model)` - returns the objective function as a `QuadExpr`.
- `getobjectivesense(m::Model)` - returns objective sense, either `:Min` or `:Max`.
- `setobjectivesense(m::Model, newSense::Symbol)` - sets the objective sense (`newSense` is either `:Min` or `:Max`).
- `getobjectivevalue(m::Model)` - returns objective value after a call to `solve`.
- `getobjectivebound(m::Model)` - returns the best known bound on the optimal objective value after a call to `solve`.

Output

- `writeLP(m::Model, filename::AbstractString; genericnames=true)` - write the model to `filename` in the LP file format. Set `genericnames=false` for user-defined variable names.
- `writeMPS(m::Model, filename::AbstractString)` - write the model to `filename` in the MPS file format.

Solve status

The call `status = solve(m)` returns a symbol recording the status of the optimization process, as reported by the solver. Typical values are listed in the table below, although the code can take solver-dependent values. For instance, certain solvers prove infeasibility or unboundedness during presolve, but do not report which of the two cases holds. See your solver interface documentation (as linked to in the *solver table*) for more information.

Status	Meaning
`:Optimal`	Problem solved to optimality
`:Unbounded`	Problem is unbounded
`:Infeasible`	Problem is infeasible
`:UserLimit`	Iteration limit or timeout
`:Error`	Solver exited with an error
`:NotSolved`	Model built in memory but not optimized

Quadratic Objectives

Quadratic objectives are supported by JuMP using a solver which implements the corresponding extensions of the MathProgBase interface. Add them in the same way you would a linear objective:

```
using Ipopt
m = Model(solver=IpoptSolver())
@variable(m, 0 <= x <= 2 )
@variable(m, 0 <= y <= 30 )

@objective(m, Min, x*x+ 2x*y + y*y )
@constraint(m, x + y >= 1 )

print(m)

status = solve(m)
```

Second-order cone constraints

Second-order cone constraints of the form $||Ax - b||_2 + a^T x + c \leq 0$ can be added directly using the `norm` function:

```
@constraint(m, norm(A*x) <= 2w - 1)
```

You may use generator expressions within `norm()` to build up normed expressions with complex indexing operations in much the same way as with `sum(...)`:

```
@constraint(m, norm(2x[i] - i for i=1:n if c[i] == 1) <= 1)
```

Accessing the low-level model

It is possible to construct the internal low-level model before optimizing. To do this, call the `JuMP.build` function. It is then possible to obtain this model by using the `internalmodel` function. This may be useful when it is necessary to access some functionality that is not exposed by JuMP. When you are ready to optimize, simply call `solve` in the normal fashion.

Variables

Variables, also known as columns or decision variables, are the results of the optimization.

Constructors

The primary way to create variables is with the `@variable` macro. The first argument will always be a `Model`. In the examples below we assume `m` is already defined. The second argument is an expression that declares the variable name and optionally allows specification of lower and upper bounds. Adding variables "column-wise", e.g., as in column generation, is supported as well; see the syntax discussed in the *Problem Modification* section.

```
@variable(m, x )                # No bounds
@variable(m, x >= lb )          # Lower bound only (note: 'lb <= x' is not valid)
@variable(m, x <= ub )          # Upper bound only
@variable(m, lb <= x <= ub )    # Lower and upper bounds
@variable(m, x == fixedval )    # Fixed to a value (lb == ub)
```

All these variations create a new local variable, in this case `x`. The names of your variables must be valid Julia variable names. Integer and binary restrictions can optionally be specified with a third argument, `Int` or `Bin`. For advanced users, `SemiCont` and `SemiInt` may be used to create semicontinuous or semi-integer variables, respectively.

To create arrays of variables we append brackets to the variable name.

```
@variable(m, x[1:M,1:N] >= 0 )
```

will create an `M` by `N` array of variables. Both ranges and arbitrary iterable sets are supported as index sets. Currently we only support ranges of the form `a:b` where `a` is an explicit integer, not a variable. Using ranges will generally be faster than using arbitrary symbols. You can mix both ranges and lists of symbols, as in the following example:

```
s = ["Green","Blue"]
@variable(m, x[-10:10,s] , Int)
x[-4,"Green"]
```

Bounds can depend on variable indices:

```
@variable(m, x[i=1:10] >= i )
```

And indices can have dependencies on preceding indices (e.g. "triangular indexing"):

```
@variable(m, x[i=1:10,j=i:10] >= 0)
```

Note the dependency must be on preceding indices, going from left to right. That is, `@variable(m, x[i=j:10, i=1:10] >= 0)` is not valid JuMP code.

Conditions can be placed on the index values for which variables are created; the condition follows the statement of the index sets and is separated with a semicolon:

```
@variable(m, x[i=1:10,j=1:10; isodd(i+j)] >= 0)
```

Note that only one condition can be added, although expressions can be built up by using the usual `&&` and `||` logical operators.

An initial value of each variable may be provided with the `start` keyword to `@variable`:

```
@variable(m, x[i=1:10], start=(i/2))
```

Is equivalent to:

```
@variable(m, x[i=1:10])
for i in 1:10
    setvalue(x[i], i/2)
end
```

For more complicated variable bounds, it may be clearer to specify them using the `lowerbound` and `upperbound` keyword arguments to `@variable`:

```
@variable(m, x[i=1:3], lowerbound=my_complex_function(i))
@variable(m, x[i=1:3], lowerbound=my_complex_function(i), upperbound=another_
↪function(i))
```

Variable categories may be set in a more programmatic way by providing the appropriate symbol to the `category` keyword argument:

```
t = [:Bin,:Int]
@variable(m, x[i=1:2], category=t[i])
@variable(m, y, category=:SemiCont)
```

The constructor `Variable(m::Model,idx::Int)` may be used to create a variable object corresponding to an *existing* variable in the model (the constructor does not add a new variable to the model). The variable indices correspond to those of the internal MathProgBase model. The inverse of this operation is `linearindex(x::Variable)`, which returns the flattened out (linear) index of a variable as JuMP provides it to a solver. We guarantee that `Variable(m,linearindex(x))` returns `x` itself. These methods are only useful if you intend to interact with solver properties which are not directly exposed through JuMP.

Note: `@variable` is equivalent to a simple assignment `x = ...` in Julia and therefore redefines variables. The following code will generate a warning and may lead to unexpected results:

```
@variable(m, x[1:10,1:10])
@variable(m, x[1:5])
```

After the second line, the Julia variable `x` refers to a set of variables indexed by the range `1:5`. The reference to the first set of variables has been lost, although they will remain in the model. See also the section on anonymous variables.

Anonymous variables

We also provide a syntax for constructing "anonymous" variables. In `@variable`, you may omit the name of the variable and instead assign the return value as you would like:

```
x = @variable(m) # Equivalent to @variable(m, x)
x = @variable(m, [i=1:3], lowerbound = i, upperbound = 2i) # Equivalent to␣
↪@variable(m, i <= x[i=1:3] <= 2i)
```

The `lowerbound` and `upperbound` keywords must be used instead of comparison operators for specifying variable bounds within the anonymous syntax. For creating noncontinuous anonymous variables, the `category` keyword must be used to avoid ambiguity, e.g.:

```
x = @variable(m, Bin) # error
x = @variable(m, category = :Bin) # ok
```

Besides these syntax restrictions in the `@variable` macro, the **only** differences between anonymous and named variables are:

1. For the purposes of printing a model, JuMP will not have a name for anonymous variables and will instead use `__anon__`. You may set the name of a variable for printing by using `setname` or the `basename` keyword argument described below.
2. Anonymous variables cannot be retrieved by using `getindex` or `m[name]`.

If you would like to change the name used when printing a variable or group of variables, you may use the `basename` keyword argument:

```
i = 3
@variable(m, x[1:3], basename="myvariable-$i")
# OR:
x = @variable(m, [1:3], basename="myvariable-$i")
```

Printing `x[2]` will display `myvariable-3[2]`.

Semidefinite and symmetric variables

JuMP supports modeling with semidefinite variables. A square symmetric matrix X is positive semidefinite if all eigenvalues are nonnegative; this is typically denoted by $X \succeq 0$. You can declare a matrix of variables to be positive semidefinite as follows:

```
@variable(m, X[1:3,1:3], SDP)
```

Note in particular the indexing: 1) exactly two index sets must be specified, 2) they must both be unit ranges starting at 1, 3) no bounds can be provided alongside the `SDP` tag. If you wish to impose more complex semidefinite constraints on the variables, e.g. $X - I \succeq 0$, you may instead use the `Symmetric` tag, along with a semidefinite constraint:

```
@variable(m, X[1:n,1:n], Symmetric)
@SDconstraint(m, X >= eye(n))
```

Bounds can be provided as normal when using the `Symmetric` tag, with the stipulation that the bounds are symmetric themselves.

`@variables` blocks

JuMP provides a convenient syntax for defining multiple variables in a single block:

```
@variables m begin
    x
    y >= 0
    Z[1:10], Bin
    X[1:3,1:3], SDP
    q[i=1:2], (lowerbound = i, start = 2i, upperbound = 3i)
    t[j=1:3], (Int, start = j)
end

# Equivalent to:
@variable(m, x)
@variable(m, y >= 0)
@variable(m, Z[1:10], Bin)
@variable(m, X[1:3,1:3], SDP)
```

```
@variable(m, q[i=1:2], lowerbound = i, start = 2i, upperbound = 3i)
@variable(m, t[j=1:3], Int, start = j)
```

The syntax follows that of `@variable` with each declaration separated by a new line. Note that unlike in `@variable`, keyword arguments must be specified within parentheses.

Methods

Bounds

- `setlowerbound(x::Variable, lower)`, `getlowerbound(x::Variable)` - Set/get the lower bound of a variable.
- `setupperbound(x::Variable, upper)`, `getupperbound(x::Variable)` - Set/get the upper bound of a variable.

Variable Category

- `setcategory(x::Variable, v_type::Symbol)` - Set the variable category for `x` after construction. Possible categories are listed above.
- `getcategory(x::Variable)` - Get the variable category for `x`.

Helper functions

- `sum(x)` - Operates on arrays of variables, efficiently produces an affine expression. Available in macros.
- `dot(x, coeffs)` - Performs a generalized "dot product" for arrays of variables and coefficients up to three dimensions, or equivalently the sum of the elements of the Hadamard product. Available in macros, and also as `dot(coeffs, x)`.

Values

- `getvalue(x)` - Get the value of this variable in the solution. If `x` is a single variable, this will simply return a number. If `x` is indexable then it will return an indexable dictionary of values. When the model is unbounded, `getvalue` will instead return the corresponding components of an unbounded ray, if available from the solver.
- `setvalue(x,v)` - Provide an initial value `v` for this variable that can be used by supporting MILP solvers. If `v` is `NaN`, the solver may attempt to fill in this value to construct a feasible solution. `setvalue` cannot be used with fixed variables; instead their value may be set with `JuMP.fix(x,v)`.
- `getdual(x)` - Get the reduced cost of this variable in the solution. Similar behavior to `getvalue` for indexable variables.

Note: The `getvalue` function always returns a floating-point value, even when a variable is constrained to take integer values, as most solvers only guarantee integrality up to a particular numerical tolerance. The built-in `round` function should be used to obtain integer values, e.g., by calling `round(Integer, getvalue(x))`.

Names

Variables (in the sense of columns) can have internal names (different from the Julia variable name) that can be used for writing models to file. This feature is disabled for performance reasons, but will be added if there is demand or a special use case.

- `setname(x::Variable, newName)`, `getname(x::Variable)` - Set/get the variable's internal name.

Fixed variables

Fixed variables, created with the `x == fixedval` syntax, have slightly special semantics. First, it is important to note that fixed variables are considered optimization variables, not constants, for the purpose of determining the problem class. For example, in:

```
@variable(m, x == 5)
@variable(m, y)
@constraint(m, x*y <= 10)
```

the constraint added is a nonconvex quadratic constraint. For efficiency reasons, JuMP will *not* substitute the constant `5` for `x` and then provide the resulting *linear* constraint to the solver. Two possible uses for fixed variables are:

1. For computing sensitivities. When available from the solver, the sensitivity of the objective with respect to the fixed value may be queried with `getdual(x)`.
2. For solving a sequence of problems with varying parameters. One may call `JuMP.fix(x, val)` to change the value of a fixed variable or to fix a previously unfixed variable. For LPs in particular, most solvers are able to efficiently hot-start when solving the resulting modified problem.

Expressions and Constraints

Constructor

`AffExpr` is an affine expression type defined by JuMP. It has three fields: a vector of coefficients, a vector of variables, and a constant. Apart from a default constructor that takes no arguments, it also has a full constructor that can be useful if you want to manually build an affine expression:

```
aff = AffExpr([x, z], [3.0, 4.0], 2.0)  # 3x + 4z + 2
```

Note that the coefficients must be floating point numbers. The matching constraint for `AffExpr` is `LinearConstraint` which is defined by an `AffExpr` and a lower and upper bound. If a solver interface does not support range constraints, this will automatically translated into two constraints at solve time. Constructing constraints manually is not an expected behavior and won't add the constraint to a model automatically. See below for the correct methods.

There is also `QuadExpr` for quadratic expressions type that also provides a default constructor that takes no arguments and a full constructor. There are four fields: two vectors of variables, a vector of coefficients, and the affine part of the expression. This is best explained by example:

```
aff = AffExpr([x, z], [3.0, 4.0], 2.0)  # 3x + 4z + 2
quad = QuadExpr([x,y],[x,z],[3.0,4.0],aff)  # 3x^2 + 4yz + 3x + 4z + 2
```

The corresponding constraint is `QuadConstraint`, which is expected to be a convex quadratic constraint.

Methods

- `@constraint(m::Model, con)` - add linear or quadratic constraints.
- `@constraint(m::Model, ref, con)` - add groups of linear or quadratic constraints. See Constraint Reference section for details.
- `JuMP.addconstraint(m::Model, con)` - general way to add linear and quadratic constraints.

- `@constraints` - add groups of constraints at once, in the same fashion as @constraint. The model must be the first argument, and multiple constraints can be added on multiple lines wrapped in a `begin ... end` block. For example:

```
@constraints(m, begin
  x >= 1
  y - w <= 2
  sum_to_one[i=1:3], z[i] + y == 1
end)
```

- `@expression(m::Model, ref, expr)` - efficiently builds a linear, quadratic, or second-order cone expression but does not add to model immediately. Instead, returns the expression which can then be inserted in other constraints. For example:

```
@expression(m, shared, sum(i*x[i] for i=1:5))
@constraint(m, shared + y >= 5)
@constraint(m, shared + z <= 10)
```

The `ref` accepts index sets in the same way as `@variable`, and those indices can be used in the construction of the expressions:

```
@expression(m, expr[i=1:3], i*sum(x[j] for j=1:3))
```

Anonymous syntax is also supported:

```
expr = @expression(m, [i=1:3], i*sum(x[j] for j=1:3))
```

- `@SDconstraint(m::Model, expr)` - adds a semidefinite constraint to the model `m`. The expression `expr` must be a square, two-dimensional array.
- `addSOS1(m::Model, coll::Vector{AffExpr})` - adds special ordered set constraint of type 1 (SOS1). Specify the set as a vector of weighted variables, e.g. `coll = [3x, y, 2z]`. Note that solvers expect the weights to be unique. See here for more details. If there is no inherent weighting in your model, an SOS constraint is probably unnecessary.
- `addSOS2(m::Model, coll::Vector{AffExpr})` - adds special ordered set constraint of type 2 (SOS2). Specify the set as a vector of weighted variables, e.g. `coll = [3x, y, 2z]`. Note that solvers expect the weights to be unique. See here for more details.
- `@LinearConstraint(expr)` - Constructs a `LinearConstraint` instance efficiently by parsing the `expr`. The same as `@constraint`, except it does not attach the constraint to any model.
- `@LinearConstraints(expr)` - Constructs a vector of `LinearConstraint` objects. Similar to `@LinearConstraint`, except it accepts multiple constraints as input as long as they are separated by newlines.
- `@QuadConstraint(expr)` - Constructs a `QuadConstraint` instance efficiently by parsing the `expr`. The same as `@constraint`, except it does not attach the constraint to any model.
- `@QuadConstraints(expr)` - Constructs a vector of `QuadConstraint` objects. Similar to `@QuadConstraint`, except it accepts multiple constraints as input as long as they are separated by newlines.
- `push!(aff::AffExpr, new_coeff::Float64, new_var::Variable)` - efficient way to grow an affine expression by one term. For example, to add `5x` to an existing expression `aff`, use `push!(aff, 5.0, x)`. This is significantly more efficient than `aff += 5.0*x`.
- `append!(aff::AffExpr, other::AffExpr)` - efficiently append the terms of an affine expression to an existing affine expression. For example, given `aff = 5.0*x` and `other = 7.0*y + 3.0*z`, we can grow `aff` using `append!(aff, other)` which results in `aff` equaling `5x + 7y + 3z`. This is significantly more efficient than using `aff += other`.

- `sum(affs::Array{AffExpr})` - efficiently sum an array of affine expressions.
- `getvalue(expr)` - evaluate an `AffExpr` or `QuadExpr`, given the current solution values.
- `linearterms{C,V}(aff::GenericAffExpr{C,V})` - provides an iterator over the `(a_i::C, x_i::V)` terms in affine expression $\sum_i a_i x_i + b$.

Constraint References

In order to manipulate constraints after creation, it is necessary to maintain a reference. The simplest way to do this is to use the special three-argument named constraint syntax for `@constraint`, which additionally allows you to create groups of constraints indexed by sets analogously to `@variable`. For example:

```
@variable(m, x[1:3])
@variable(m, y[2:2:6])
@constraint(m, xyconstr[i=1:3,j=6:-2:2], x[i] - y[j] == 1)
```

adds 9 constraints to the model `m` of the expected form. The variable `xyconstr` is a collection of `ConstraintRef{Model,LinearConstraint}` instances indexed by the ranges `1:3` and `6:-2:2` (the ordered tuple `(6,4,2)`), so, for example `xyconstr[2,4]` is a reference to the constraint `x[2] - y[4] == 1`. Indices can have dependencies on preceding indices, e.g. triangular indexing is allowed:

```
@constraint(m, triconstr[i=1:3,j=2i:2:6], x[i] - y[j] == 1)
```

A condition can be added following the indices; a semicolon is used to separate index sets from the condition:

```
@constraint(m, constr[i=1:5,j=1:5; i+j >= 3], x[i] - y[j] == 1)
```

Note that only one condition can be added, although expressions can be built up by using the usual `&&` and `||` logical operators.

Anonymous syntax is supported:

```
constr = @constraint(m, [i=1:5,j=1:5; i+j >= 3], x[i] - y[j] == 1)
```

To obtain the dual of a constraint, call `getdual` on the constraint reference:

```
println(getdual(xyconstr[1,6]))
```

When an LP model is infeasible, `getdual` will return the corresponding component of the infeasibility ray (Farkas proof), if available from the solver.

Dual information is also accessible for second-order cone problems as described below. Duals are unavailable for MIPs.

One may retrieve the corresponding internal `LinearConstraint` object from a `ConstraintRef{Model, LinearConstraint}` object `constr` by calling `LinearConstraint(constr)`. This functionality is not yet implemented for other classes of constraints.

For users who prefer to generate constraints in an explicit loop, we also provide the `@constraintref` convenience macro, e.g.:

```
@constraintref constraintName[1:3]
```

You can then iterate over constraints and store references in this structure, e.g.:

```
@variable(m, x[1:5] >= 0)
@constraintref myCons[1:5]
for i = 1:5
  myCons[i] = @constraint(m, x[i] >= i)
end
```

Conic constraint duals

JuMP supports accessing the dual solutions to second-order cone problems. Dual multipliers on variable bounds, linear constraints, and second-order cone constraints are accessible through `getdual()` given the corresponding variable or constraint reference object. For second-order cone constraints, `getdual(c::ConstraintRef{Model, SOCConstraint})` returns a vector of dual variables in the dimension of the corresponding cone. Duals are defined such that they are consistent in sign with linear programming duals in the case that the second-order cone constraints are inactive.

For example:

```
m = Model()
@variable(m, x[1:2] >= 1)
@variable(m, t)
@objective(m, Min, t)
@constraint(m, soc, norm( x[i] for i=1:2 ) <= t)
status = solve(m)

@show getvalue(x) # [1.000000000323643,1.0000000032357 63]
@show getvalue(t) # 1.4142135583106126
@show getdual(x)  # [0.7071067807797846,0.7071067802906756]
@show getdual(soc)# [-1.0000000004665652,0.707106779497123,0.707106779008014]
```

Note that the *negative* of the dual vector `getdual(soc)` belongs to the second-order cone. See the MathProgBase documentation for more on the definition of the dual problem. The dual solutions returned by JuMP agree with the definitions from MathProgBase up to a possible change in sign.

Vectorized operations

JuMP supports vectorized expressions and constraints for linear and quadratic models. Although this syntax may be familiar for users coming from MATLAB-based modeling languages, we caution that this syntax may be slower than the scalar versions using loops—especially for large operations. Nevertheless, the syntax often proves useful, for example in constraints involving small, dense matrix-vector products.

Linear algebraic operators are available to give meaning to expressions like `A*x` where `A` is a matrix of numbers and `x` is a vector of `Variable` objects. You may also use objects of type `Array{Variable}` in these kinds of expressions; for example, any object you construct with `@variable` where each of the index sets are of the form `1:n`. For example:

```
@variable(m, x[1:3,1:4])
expr = rand(3,3)*x
```

is allowed, while:

```
@variable(m, x[2:4])
expr = rand(3,3)*x
```

is not. Addition and subtraction are also defined in similar ways, following the usual Julia rules for linear algebra over arrays.

Vectorized constraints can be added to the model, using the elementwise comparison operators `.==`, `.>=`, and `.<=`. For instance, you can write constraints of the form:

```
@variable(m, x[1:10])
A = rand(5,10)
b = rand(5)
@constraint(m, A*x + b .<= 1)
```

Note that scalar literals (such as 1 or 0) are allowed in expressions.

Concatenation is also possible for these arrays of variables or expressions. For instance, the following will create a matrix of `QuadExpr` that you can use elsewhere in your model:

```
@variable(m, x[1:3])
A = [1 x'
     x x*x']
```

Finally, note that this feature is not currently supported directly in nonlinear expressions; for example, a matrix–vector product will not work inside a call to the `@NLconstraint` macro.

Problem Modification

It can be useful to modify models after they have been created and solved, for example when we are solving many similar models in succession or generating the model dynamically (e.g. column generation). Additionally it is sometimes desirable for the solver to re-start from the last solution to reduce running times for successive solves ("hot-start"). Where available, JuMP exposes this functionality.

Differences in Solvers

Some solvers do not expose the ability to modify a model after creation - the model must be constructed from scratch each time. JuMP will use the ability to modify problems exposed by the solver if possible, and will still work even if the solver does not support this functionality by passing the complete problem to the solver every time.

Modifying variables

As before, variables can be added using the `@variable` macro. To remove a variable, one can set the bounds on that variable to zero, e.g.:

```
setlowerbound(x, 0.0)
setupperbound(x, 0.0)
```

While bound updates are applied immediately in JuMP, variable bound changes are not transmitted to the solver until `solve` is called again.

To add variables that appear in existing constraints, e.g. in column generation, there is an alternative form of the `@variable` macro:

```
@variable(m, x, objective = objcoef, inconstraints = constrrefs, coefficients =␣
↪values)
@variable(m, x >= lb, objective = objcoef, inconstraints = constrrefs, coefficients =␣
↪values)
@variable(m, x <= ub, objective = objcoef, inconstraints = constrrefs, coefficients =␣
↪values)
```

```
@variable(m, lb <= x <= ub, objective = objcoef, inconstraints = constrrefs,␣
↪coefficients = values)
@variable(m, lb <= x <= ub, Int, objective = objcoef, inconstraints = constrrefs,␣
↪coefficients = values)  # Types are supported
```

where `objcoef` is the coefficient of the variable in the new problem, `constrrefs` is a vector of `ConstraintRef`, and `values` is a vector of numbers. To give an example, consider the following code snippet:

```
m = Model()
@variable(m, 0 <= x <= 1)
@variable(m, 0 <= y <= 1)
@objective(m, Max, 5x + 1y)
@constraint(m, con, x + y <= 1)
solve(m)  # x = 1, y = 0
@variable(m, 0 <= z <= 1, objective = 10.0, inconstraints = [con], coefficients = [1.
↪0])
# The constraint is now x + y + z <= 1
# The objective is now 5x + 1y + 10z
solve(m)  # z = 1
```

In some situations you may be adding all variables in this way. To do so, first define a set of empty constraints, e.g.

```
m = Model()
@constraint(m, con, 0 <= 1)
@objective(m, Max, 0)
@variable(m, 0 <= x <= 1, objective = 5, inconstraints = [con], coefficients = [1.0])
@variable(m, 0 <= y <= 1, objective = 1, inconstraints = [con], coefficients = [1.0])
@variable(m, 0 <= z <= 1, objective = 10, inconstraints = [con], coefficients = [1.0])
solve(m)
```

Modifying constraints

JuMP does not currently support changing constraint coefficients. For less-than and greater-than constraints, the right-hand-side can be changed, e.g.:

```
@constraint(m, mycon, x + y <= 4)
solve(m)
JuMP.setRHS(mycon, 3)  # Now x + y <= 3
solve(m)  # Hot-start for LPs
```

Modifying the objective

To change the objective, simply call `@objective` again - the previous objective function and sense will be replaced.

Modifying nonlinear models

See *nonlinear parameters*.

Solver Callbacks

Many mixed-integer (linear, conic, and nonlinear) programming solvers offer the ability to modify the solve process. Examples include changing branching decisions in branch-and-bound, adding custom cutting planes, providing custom heuristics to find feasible solutions, or implementing on-demand separators to add new constraints only when they are violated by the current solution (also known as lazy constraints).

While historically this functionality has been limited to solver-specific interfaces, JuMP provides *solver-independent* support for a number of commonly used solver callbacks. Currently, we support lazy constraints, user-provided cuts, and user-provided heuristics for the Gurobi, CPLEX, GLPK, and SCIP solvers. We do not yet support any other class of callbacks, but they may be accessible by using the solver's low-level interface.

Lazy Constraints

Lazy constraints are useful when the full set of constraints is too large to explicitly include in the initial formulation. When a MIP solver reaches a new solution, for example with a heuristic or by solving a problem at a node in the branch-and-bound tree, it will give the user the chance to provide constraint(s) that would make the current solution infeasible. For some more information about lazy constraints, see this blog post by Paul Rubin.

There are three important steps to providing a lazy constraint callback. First we must write a function that will analyze the current solution that takes a single argument, e.g. `function myLazyConGenerator(cb)`, where cb is a reference to the callback management code inside JuMP. Next you will do whatever analysis of the solution you need to inside your function to generate the new constraint before adding it to the model with `@lazyconstraint(cb, myconstraint)`. There is an optional keyword option `localcut` to `@lazyconstraint`, which indicates if the lazy constraint that will be added will only apply at the current node and the tree rooted at that node. For example, `@lazyconstraint(cb, myconstraint, localcut=true)`. By default, `localcut` is set to `false`. Finally we notify JuMP that this function should be used for lazy constraint generation using the `addlazycallback(m, myLazyConGenerator)` function before we call `solve(m)`.

The following is a simple example to make this more clear. In this two-dimensional problem we have a set of box constraints explicitly provided and a set of two lazy constraints we can add on the fly. The solution without the lazy constraints will be either (0,2) or (2,2), and the final solution will be (1,2):

```
using JuMP
using Gurobi

# We will use Gurobi
m = Model(solver=GurobiSolver())

# Define our variables to be inside a box, and integer
@variable(m, 0 <= x <= 2, Int)
@variable(m, 0 <= y <= 2, Int)

@objective(m, Max, y)

# We now define our callback function that takes one argument,
# the callback handle. Note that we can access m, x, and y because
# this function is defined inside the same scope
function corners(cb)
    x_val = getvalue(x)
    y_val = getvalue(y)
    println("In callback function, x=$x_val, y=$y_val")

    # We have two constraints, one cutting off the top
    # left corner and one cutting off the top right corner, e.g.
    # (0,2) +---+---+ (2,2)
```

```
    #        |xx/ \xx|
    #        |x/   \x|
    #        |/     \|
    #        +       +
    #        |       |
    #        |       |
    #        |       |
    # (0,0) +---+---+ (2,0)

    # Allow for some impreciseness in the solution
    TOL = 1e-6

    # Check top left, allowing some tolerance
    if y_val - x_val > 1 + TOL
        # Cut off this solution
        println("Solution was in top left, cut it off")
        # Use the original variables, but not m - cb instead
        @lazyconstraint(cb, y - x <= 1)
    # Check top right
    elseif y_val + x_val > 3 + TOL
        # Cut off this solution
        println("Solution was in top right, cut it off")
        # Use the original variables, but not m - cb instead
        @lazyconstraint(cb, y + x <= 3)
    end
end  # End of callback function

# Tell JuMP/Gurobi to use our callback function
addlazycallback(m, corners)

# Solve the problem
solve(m)

# Print our final solution
println("Final solution: [ $(getvalue(x)), $(getvalue(y)) ]")
```

The code should print something like (amongst the output from Gurobi):

```
In callback function, x=2.0, y=2.0
Solution was in top right, cut it off
In callback function, x=0.0, y=2.0
Solution was in top left, cut it off
In callback function, x=1.0, y=2.0
Final solution: [ 1.0, 2.0 ]
```

This code can also be found in `/JuMP/examples/simplelazy.jl`.

There is an optional `fractional` keyword option to `addlazycallback` which indicates that the callback may be called at solutions that do not satisfy integrality constraints. For example, `addlazycallback(m, myLazyConGenerator, fractional=true)`. Depending on the solver, this may invoke the callback after solving each LP relaxation in the Branch and Bound tree. By default, `fractional` is set to `false`.

User Cuts

User cuts, or simply cuts, provide a way for the user to tighten the LP relaxation using problem-specific knowledge that the solver cannot or is unable to infer from the model. Just like with lazy constraints, when a MIP solver reaches a new node in the branch-and-bound tree, it will give the user the chance to provide cuts to make the current relaxed

(fractional) solution infeasible in the hopes of obtaining an integer solution. For more details about the difference between user cuts and lazy constraints see the aforementioned blog post.

Your user cuts should not change the set of integer feasible solutions. Equivalently, your cuts can only remove fractional solutions - that is, "tighten" the LP relaxation of the MILP. If you add a cut that removes an integer solution, the solver may return an incorrect solution.

Adding a user cut callback is similar to adding a lazy constraint callback. First we must write a function that will analyze the current solution that takes a single argument, e.g. `function myUserCutGenerator(cb)`, where cb is a reference to the callback management code inside JuMP. Next you will do whatever analysis of the solution you need to inside your function to generate the new constraint before adding it to the model with the JuMP macro `@usercut(cb, myconstraint)` (same limitations as addConstraint). There is an optional keyword option `localcut` to `@usercut`, which indicates if the user cut that will be added will only apply at the current node and the tree rooted at that node. For example, `@usercut(cb, myconstraint, localcut=true)`. By default, `localcut` is set to `false`. Finally we notify JuMP that this function should be used for user cut generation using the `addcutcallback(m, myUserCutGenerator)` function before we call `solve(m)`.

Consider the following example which is related to the lazy constraint example. The problem is two-dimensional, and the objective sense prefers solution in the top-right of a 2-by-2 square. There is a single constraint that cuts off the top-right corner to make the LP relaxation solution fractional. We will exploit our knowledge of the problem structure to add a user cut that will make the LP relaxation integer, and thus solve the problem at the root node:

```
using JuMP
using Gurobi

# We will use Gurobi, which requires that we manually set the attribute
# PreCrush to 1 if we have user cuts. We will also disable PreSolve, Cuts,
# and Heuristics so only our cut will be used
m = Model(solver=GurobiSolver(PreCrush=1, Cuts=0, Presolve=0, Heuristics=0.0))

# Define our variables to be inside a box, and integer
@variable(m, 0 <= x <= 2, Int)
@variable(m, 0 <= y <= 2, Int)

# Optimal solution is trying to go towards top-right corner (2.0, 2.0)
@objective(m, Max, x + 2y)

# We have one constraint that cuts off the top right corner
@constraint(m, y + x <= 3.5)

# Optimal solution of relaxed problem will be (1.5, 2.0)
# We can add a user cut that will cut of this fractional solution.

# We now define our callback function that takes one argument,
# the callback handle. Note that we can access m, x, and y because
# this function is defined inside the same scope
function mycutgenerator(cb)
    x_val = getvalue(x)
    y_val = getvalue(y)
    println("In callback function, x=$x_val, y=$y_val")

    # Allow for some impreciseness in the solution
    TOL = 1e-6

    # Check top right
    if y_val + x_val > 3 + TOL
        # Cut off this solution
        println("Fractional solution was in top right, cut it off")
```

```
        # Use the original variables
        @usercut(cb, y + x <= 3)
    end
end  # End of callback function

# Tell JuMP/Gurobi to use our callback function
addcutcallback(m, mycutgenerator)

# Solve the problem
solve(m)

# Print our final solution
println("Final solution: [ $(getvalue(x)), $(getvalue(y)) ]")
```

The code should print something like (amongst the output from Gurobi):

```
In callback function, x=1.5, y=2.0
Fractional solution was in top right, cut it off
In callback function, x=1.0, y=2.0
Final solution: [ 1.0, 2.0 ]
```

This code can also be found in `/JuMP/examples/simpleusercut.jl`.

User Heuristics

Integer programming solvers frequently include heuristics that run at the nodes of the branch-and-bound tree. They aim to find integer solutions quicker than plain branch-and-bound would to tighten the bound, allowing us to fathom nodes quicker and to tighten the integrality gap. Some heuristics take integer solutions and explore their "local neighborhood" (e.g. flipping binary variables, fix some variables and solve a smaller MILP, ...) and others take fractional solutions and attempt to round them in an intelligent way. You may want to add a heuristic of your own if you have some special insight into the problem structure that the solver is not aware of, e.g. you can consistently take fractional solutions and intelligently guess integer solutions from them.

The user heuristic callback is somewhat different from the previous two heuristics. The general concept is that we can create multiple partial solutions and submit them back to the solver - each solution must be submitted before a new solution is constructed. As before we provide a function that analyzes the current solution and takes a single argument, e.g. `function myHeuristic(cb)`, where cb is a reference to the callback management code inside JuMP. You can build your solutions using `setsolutionvalue(cb, x, value)` and submit them with `addsolution(cb)`. Note that `addsolution` will "wipe" the previous (partial) solution. Notify JuMP that this function should be used as a heuristic using the `addheuristiccallback(m, myHeuristic)` function before calling `solve(m)`.

There is some unavoidable (for performance reasons) solver-dependent behavior - you should check your solver documentation for details. For example: GLPK will not check the feasibility of your heuristic solution. If you need to submit many heuristic solutions in one callback, there may be performance impacts from the "wiping" behavior of `addsolution` - please file an issue and we can address this issue.

Consider the following example, which is the same problem as seen in the user cuts section. The heuristic simply rounds the fractional variable to generate integer solutions.:

```
using JuMP
using Gurobi

# We will use Gurobi and disable PreSolve, Cuts, and (in-built) Heuristics so
# only our heuristic will be used
m = Model(solver=GurobiSolver(Cuts=0, Presolve=0, Heuristics=0.0))
```

```
# Define our variables to be inside a box, and integer
@variable(m, 0 <= x <= 2, Int)
@variable(m, 0 <= y <= 2, Int)

# Optimal solution is trying to go towards top-right corner (2.0, 2.0)
@objective(m, Max, x + 2y)

# We have one constraint that cuts off the top right corner
@constraint(m, y + x <= 3.5)

# Optimal solution of relaxed problem will be (1.5, 2.0)

# We now define our callback function that takes one argument,
# the callback handle. Note that we can access m, x, and y because
# this function is defined inside the same scope
function myheuristic(cb)
    x_val = getvalue(x)
    y_val = getvalue(y)
    println("In callback function, x=$x_val, y=$y_val")

    setsolutionvalue(cb, x, floor(x_val))
    # Leave y undefined - solver should handle as it sees fit. In the case
    # of Gurobi it will try to figure out what it should be.
    addsolution(cb)

    # Submit a second solution
    setsolutionvalue(cb, x, ceil(x_val))
    addsolution(cb)
end  # End of callback function

# Tell JuMP/Gurobi to use our callback function
addheuristiccallback(m, myheuristic)

# Solve the problem
solve(m)

# Print our final solution
println("Final solution: [ $(getvalue(x)), $(getvalue(y)) ]")
```

The code should print something like:

```
In callback function, x=1.5, y=2.0
     0     0    5.50000    0    1          -    5.50000      -      -    0s
H    1     0                       5.0000000    5.50000  10.0%   0.0    0s
```

where the `H` denotes a solution found with a heuristic - our heuristic in this case. This code can also be found in `/JuMP/examples/simpleheur.jl`.

Querying Solver Progress

All JuMP callback methods must take a single argument, called `cb` by convention. `cb` is a handle to the internal callback system used by the underlying solver, and allows the user to query solver state. There are a variety of methods available which are listed in the MathProgBase documentation including:

```
cbgetobj(cb)
cbgetbestbound(cb)
cbgetexplorednodes(cb)
cbgetstate(cb)
```

Informational Callbacks

Sometimes it can be useful to track solver progress without actually changing the algorithm by adding cuts or heuristic solutions. In these cases, informational callbacks can be added, wherein statistics can be tracked via the `cbget` functions discussed in the previous section. Informational callbacks are added to a JuMP model with the `addinfocallback(m::Model, f::Function; when::Symbol)` function, where the *when* argument should be one of `:MIPNode`, `:MIPSol` or `:Intermediate` (listed under `cbgetstate()` in the MathProgBase documentation)

For a simple example, we can add a function that tracks the best bound and incumbent objective value as the solver progresses through the branch-and-bound tree:

```
type NodeData
    time::Float64  # in seconds since the epoch
    node::Int
    obj::Float64
    bestbound::Float64
end

# build model ``m`` up here

bbdata = NodeData[]

function infocallback(cb)
    node      = MathProgBase.cbgetexplorednodes(cb)
    obj       = MathProgBase.cbgetobj(cb)
    bestbound = MathProgBase.cbgetbestbound(cb)
    push!(bbdata, NodeData(time(),node,obj,bestbound))
end
addinfocallback(m, infocallback, when = :Intermediate)

solve(m)

# Save results to file for analysis later
open("bbtrack.csv","w") do fp
    println(fp, "time,node,obj,bestbound")
    for bb in bbdata
        println(fp, bb.time, ",", bb.node, ",",
                    bb.obj, ",", bb.bestbound)
    end
end
```

For a second example, we can add a function that tracks the intermediate solutions at each integer-feasible solution in the Branch-and-Bound tree:

```
solutionvalues = Vector{Float64}[]

# build model ``m`` up here

function infocallback(cb)
    push!(solutionvalues, JuMP.getvalue(x))
```

```
end
addinfocallback(m, infocallback, when = :MIPSol)

solve(m)

# all the intermediate solutions are now stored in `solutionvalues`
```

Code Design Considerations

In the above examples the callback function is defined in the same scope as the model and variable definitions, allowing us to access them. If we defined the function in some other scope, or even file, we would not be able to access them directly. The proposed solution to this design problem is to separate the logic of analyzing the current solution values from the callback itself. This has many benefits, including writing unit tests for the callback function to check its correctness. The callback function passed to JuMP is then simply a stub that extracts the current solution and any other relevant information and passes that to the constraint generation logic. To apply this to our previous lazy constraint example, consider the following code:

```
using JuMP
using Gurobi
using Base.Test

function cornerChecker(x_val, y_val)
    # This function does not depend on the model, and could
    # be written anywhere. Instead, it returns a tuple of
    # values (newcut, x_coeff, y_coeff, rhs) where newcut is a
    # boolean if a cut is needed, x_coeff is the coefficient
    # on the x variable, y_coeff is the coefficient on
    # the y variable, and rhs is the right hand side
    TOL = 1e-6
    if y_val - x_val > 1 + TOL
        return (true, -1.0, 1.0, 1.0)  # Top left
    elseif y_val + x_val > 3 + TOL
        return (true,  1.0, 1.0, 3.0)  # Top right
    else
        return (false, 0.0, 0.0, 0.0)  # No cut
    end
end

# A unit test for the cornerChecker function
function test_cornerChecker()
    # Test the four corners - only two should produce cuts

    newcut, x_coeff, y_coeff, rhs = cornerChecker(0, 0)
    @test !newcut

    newcut, x_coeff, y_coeff, rhs = cornerChecker(2, 0)
    @test !newcut

    newcut, x_coeff, y_coeff, rhs = cornerChecker(0, 2)
    @test newcut
    @test x_coeff == -1.0
    @test y_coeff ==  1.0
    @test rhs == 1.0

    newcut, x_coeff, y_coeff, rhs = cornerChecker(2, 2)
    @test newcut
```

```
    @test x_coeff ==  1.0
    @test y_coeff ==  1.0
    @test rhs == 3.0
end

function solveProblem()
    m = Model(solver=GurobiSolver())

    @variable(m, 0 <= x <= 2, Int)
    @variable(m, 0 <= y <= 2, Int)
    @objective(m, Max, y)

    # Note that the callback is now a stub that passes off
    # the work to the "algorithm"
    function corners(cb)
        x_val = getvalue(x)
        y_val = getvalue(y)
        println("In callback function, x=$x_val, y=$y_val")

        newcut, x_coeff, y_coeff, rhs = cornerChecker(x_val, y_val)

        if newcut
            @lazyconstraint(cb, x_coeff*x + y_coeff*y <= rhs)
        end
    end  # End of callback function

    addlazycallback(m, corners)
    solve(m)
    println("Final solution: [ $(getvalue(x)), $(getvalue(y)) ]")
end

# Run tests
test_cornerChecker()

# Solve it
solveProblem()
```

This code can also be found in `/JuMP/examples/simplelazy2.jl`.

Exiting a callback early

If you need to exit the optimization process earlier than a solver otherwise would, it is possible to return `JuMP.StopTheSolver` from the callback code:

```
return JuMP.StopTheSolver
```

This will trigger the solver to exit immediately and return a `:UserLimit` status.

Nonlinear Modeling

JuMP has support for general smooth nonlinear (convex and nonconvex) optimization problems. JuMP is able to provide exact, sparse second-order derivatives to solvers. This information can improve solver accuracy and performance.

Nonlinear objectives and constraints are specified by using the `@NLobjective` and `@NLconstraint` macros. The familiar `sum()` syntax is supported within these macros, as well as `prod()` which analogously represents the

product of the terms within. Note that the `@objective` and `@constraint` macros (and corresponding functions) do *not* currently support nonlinear expressions. However, a model can contain a mix of linear, quadratic, and nonlinear constraints or objective functions. Starting points may be provided by using the `start` keyword argument to `@variable`. If a starting value is not provided for a variable, it will be set to the projection of zero onto the interval defined by the variable bounds. For nonconvex problems, the returned solution is only guaranteed to be locally optimal. Convexity detection is not currently provided.

For example, we can solve the classical Rosenbrock problem (with a twist) as follows:

```
using JuMP
m = Model()
@variable(m, x, start = 0.0)
@variable(m, y, start = 0.0)

@NLobjective(m, Min, (1-x)^2 + 100(y-x^2)^2)

solve(m)
println("x = ", getvalue(x), " y = ", getvalue(y))

# adding a (linear) constraint
@constraint(m, x + y == 10)
solve(m)
println("x = ", getvalue(x), " y = ", getvalue(y))
```

Examples: optimal control, maximum likelihood estimation, and Hock-Schittkowski tests.

Syntax notes

The syntax accepted in nonlinear expressions is more restricted than the syntax for linear and quadratic expressions. We note some important points below.

- All expressions must be simple scalar operations. You cannot use `dot`, matrix-vector products, vector slices, etc. Translate vector operations into explicit `sum()` operations or use the `AffExpr` plus auxiliary variable trick described below.
- There is no operator overloading provided to build up nonlinear expressions. For example, if `x` is a JuMP variable, the code `3x` will return an `AffExpr` object that can be used inside of future expressions and linear constraints. However, the code `sin(x)` is an error. All nonlinear expressions must be inside of macros.
- *User-defined functions* may be used within nonlinear expressions only after they are registered. For example:

```
myfunction(a,b) = exp(a)*b
@variable(m, x); @variable(m, y)
@NLobjective(m, Min, myfunction(x,y)) # ERROR. Needs JuMP.register() first.
@NLobjective(m, Min, exp(x)*y) # Okay
```

- `AffExpr` and `QuadExpr` objects cannot currently be used inside nonlinear expressions. Instead, introduce auxiliary variables, e.g.:

```
myexpr = dot(c,x) + 3y # where x and y are variables
@variable(m, aux)
@constraint(m, aux == myexpr)
@NLobjective(m, Min, sin(aux))
```

- You can declare embeddable nonlinear expressions with `@NLexpression`. For example:

```
@NLexpression(m, myexpr[i=1:n], sin(x[i]))
@NLconstraint(m, myconstr[i=1:n], myexpr[i] <= 0.5)
```

- Anonymous syntax is supported in `@NLexpression` and `@NLconstraint`:

```
myexpr = @NLexpression(m, [i=1:n], sin(x[i]))
myconstr = @NLconstraint(m, [i=1:n], myexpr[i] <= 0.5)
```

Nonlinear Parameters

For nonlinear models only, JuMP offers a syntax for explicit "parameter" objects which can be used to modify a model in-place just by updating the value of the parameter. Nonlinear parameters are declared by using the `@NLparameter` macro and may be indexed by arbitrary sets analogously to JuMP variables and expressions. The initial value of the parameter must be provided on the right-hand side of the == sign as seen below:

```
@NLparameter(m, x == 10)
@NLparameter(m, y[i=1:10] == my_data[i]) # set of parameters indexed from 1 to 10
```

You may use `getvalue` and `setvalue` to query or update the value of a parameter:

```
getvalue(x) # 10, from above
setvalue(y[4], 54.3) # y[4] now holds the value 54.3
```

Nonlinear parameters can be used *within nonlinear expressions* only:

```
@variable(m, z)
@objective(m, Max, x*z)       # error: x is a nonlinear parameter
@NLobjective(m, Max, x*z)     # ok
@expression(m, my_expr, x*z^2)      # error: x is a nonlinear parameter
@NLexpression(m, my_nl_expr, x*z^2) # ok
```

Nonlinear parameters are useful when solving nonlinear models in a sequence:

```
m = Model()
@variable(m, z)
@NLparameter(m, x == 1.0)
@NLobjective(m, Min, (z-x)^2)
solve(m)
getvalue(z) # equals 1.0

# Now, update the value of x to solve a different problem
setvalue(x, 5.0)
solve(m)
getvalue(z) # equals 5.0
```

Using nonlinear parameters can be faster than creating a new model from scratch with updated data because JuMP is able to avoid repeating a number of steps in processing the model before handing it off to the solver.

User-defined functions

JuMP's library of recognized univariate functions is derived from the Calculus.jl package. If you encounter a standard special function not currently supported by JuMP, consider contributing to the list of derivative rules there. In addition to this built-in list of functions, it is possible to register custom (*user-defined*) nonlinear functions to use within nonlinear expressions. JuMP does not support black-box optimization, so all user-defined functions must provide derivatives in some form. Fortunately, JuMP supports **automatic differentiation of user-defined functions**, a feature to our knowledge not available in any comparable modeling systems.

Note: Automatic differentiation is *not* finite differencing. JuMP's automatically computed derivatives are not subject to approximation error.

JuMP uses ForwardDiff.jl to perform automatic differentiation; see the ForwardDiff.jl documentation for a description of how to write a function suitable for automatic differentiation. The general guideline is to write code that is generic with respect to the number type; don't assume that the input to the function is `Float64`. To register a user-defined function with derivatives computed by automatic differentiation, use the `JuMP.register` method as in the following example:

```
mysquare(x) = x^2
myf(x,y) = (x-1)^2+(y-2)^2

m = Model()

JuMP.register(m, :myf, 2, myf, autodiff=true)
JuMP.register(m, :mysquare, 1, mysquare, autodiff=true)

@variable(m, x[1:2] >= 0.5)
@NLobjective(m, Min, myf(x[1],mysquare(x[2])))
```

The above code creates a JuMP model with the objective function `(x[1]-1)^2 + (x[2]^2-2)^2`. The first argument to `JuMP.register` the model for which the functions are registered. The second argument is a Julia symbol object which serves as the name of the user-defined function in JuMP expressions; the JuMP name need not be the same as the name of the corresponding Julia method. The third argument specifies how many arguments the function takes. The fourth argument is the name of the Julia method which computes the function, and `autodiff=true` instructs JuMP to compute exact gradients automatically.

Note: All arguments to user-defined functions are scalars, not vectors. To define a function which takes a large number of arguments, you may use the splatting syntax `f(x...) = ...`.

Forward-mode automatic differentiation as implemented by ForwardDiff.jl has a computational cost that scales linearly with the number of input dimensions. As such, it is not the most efficient way to compute gradients of user-defined functions if the number of input arguments is large. In this case, users may want to provide their own routines for evaluating gradients. The more general syntax for `JuMP.register` which accepts user-provided derivative evaluation routines is:

```
JuMP.register(m::Model, s::Symbol, dimension::Integer, f::Function, ∇f::Function,␣
↪∇²f::Function)
```

The input differs for functions which take a single input argument and functions which take more than one. For univariate functions, the derivative evaluation routines should return a number which represents the first and second-order derivatives respectively. For multivariate functions, the derivative evaluation routines will be passed a gradient vector which they must explicitly fill. Second-order derivatives of multivariate functions are not currently supported; this argument should be omitted. The following example sets up the same optimization problem as before, but now we explicitly provide evaluation routines for the user-defined functions:

```
mysquare(x) = x^2
mysquare_prime(x) = 2x
mysquare_primeprime(x) = 2

myf(x,y) = (x-1)^2+(y-2)^2
function ∇f(g,x,y)
    g[1] = 2*(x-1)
    g[2] = 2*(y-2)
```

```
end

m = Model()

JuMP.register(m, :myf, 2, myf, ∇f)
JuMP.register(m, :mysquare, 1, mysquare, mysquare_prime, mysquare_primeprime)

@variable(m, x[1:2] >= 0.5)
@NLobjective(m, Min, myf(x[1],mysquare(x[2])))
```

Factors affecting solution time

The execution time when solving a nonlinear programming problem can be divided into two parts, the time spent in the optimization algorithm (the solver) and the time spent evaluating the nonlinear functions and corresponding derivatives. Ipopt explicitly displays these two timings in its output, for example:

```
Total CPU secs in IPOPT (w/o function evaluations)   =      7.412
Total CPU secs in NLP function evaluations           =      2.083
```

For Ipopt in particular, one can improve the performance by installing advanced sparse linear algebra packages, see *Installation Guide*. For other solvers, see their respective documentation for performance tips.

The function evaluation time, on the other hand, is the responsibility of the modeling language. JuMP computes derivatives by using the ReverseDiffSparse package, which implements, in pure Julia, reverse-mode automatic differentiation with graph coloring methods for exploiting sparsity of the Hessian matrix[1]. As a conservative bound, JuMP's performance here currently may be expected to be within a factor of 5 of AMPL's.

Querying derivatives from a JuMP model

For some advanced use cases, one may want to directly query the derivatives of a JuMP model instead of handing the problem off to a solver. Internally, JuMP implements the `AbstractNLPEvaluator` interface from MathProgBase. To obtain an NLP evaluator object from a JuMP model, use `JuMP.NLPEvaluator`. The `linearindex` method maps from JuMP variables to the variable indices at the MathProgBase level.

For example:

```
m = Model()
@variable(m, x)
@variable(m, y)

@NLobjective(m, Min, sin(x) + sin(y))
values = zeros(2)
values[linearindex(x)] = 2.0
values[linearindex(y)] = 3.0

d = JuMP.NLPEvaluator(m)
MathProgBase.initialize(d, [:Grad])
objval = MathProgBase.eval_f(d, values) # == sin(2.0) + sin(3.0)

∇f = zeros(2)
MathProgBase.eval_grad_f(d, ∇f, values)
# ∇f[linearindex(x)] == cos(2.0)
# ∇f[linearindex(y)] == cos(3.0)
```

[1] Dunning, Huchette, and Lubin, "JuMP: A Modeling Language for Mathematical Optimization", arXiv.

The ordering of constraints in a JuMP model corresponds to the following ordering at the MathProgBase nonlinear abstraction layer. There are three groups of constraints: linear, quadratic, and nonlinear. Linear and quadratic constraints, to be recognized as such, must be added with the `@constraint` macros. All constraints added with the `@NLconstraint` macros are treated as nonlinear constraints. Linear constraints are ordered first, then quadratic, then nonlinear. The `linearindex` method applied to a constraint reference object returns the index of the constraint *within its corresponding constraint class*. For example:

```
m = Model()
@variable(m, x)
@constraint(m, cons1, x^2 <= 1)
@constraint(m, cons2, x + 1 == 3)
@NLconstraint(m, cons3, x + 5 == 10)

typeof(cons1) # JuMP.ConstraintRef{JuMP.Model,JuMP.GenericQuadConstraint{JuMP.
↪GenericQuadExpr{Float64,JuMP.Variable}}} indicates a quadratic constraint
typeof(cons2) # JuMP.ConstraintRef{JuMP.Model,JuMP.GenericRangeConstraint{JuMP.
↪GenericAffExpr{Float64,JuMP.Variable}}} indicates a linear constraint
typeof(cons3) # JuMP.ConstraintRef{JuMP.Model,JuMP.GenericRangeConstraint{JuMP.
↪NonlinearExprData}} indicates a nonlinear constraint
linearindex(cons1) == linearindex(cons2) == linearindex(cons3) == 1
```

When querying derivatives, `cons2` will appear first, because it is the first linear constraint, then `cons1`, because it is the first quadratic constraint, then `cons3`, because it is the first nonlinear constraint. Note that for one-sided nonlinear constraints, JuMP subtracts any values on the right-hand side when computing expressions. In other words, one-sided nonlinear constraints are always transformed to have a right-hand side of zero.

The `JuMP.constraintbounds(m::Model)` method returns the lower and upper bounds of all the constraints in the model, concatenated in the order discussed above.

This method of querying derivatives directly from a JuMP model is convenient for interacting with the model in a structured way, e.g., for accessing derivatives of specific variables. For example, in statistical maximum likelihood estimation problems, one is often interested in the Hessian matrix at the optimal solution, which can be queried using the `JuMP.NLPEvaluator`.

If you are writing a "solver", we *highly encourage* use of the MathProgBase nonlinear interface over querying derivatives using the above methods. These methods are provided for convenience but do not fully integrate with JuMP's solver infrastructure. In particular, they do not allow users to specify your solver to the `Model()` constructor nor to call it using `solve()` nor to populate the solution back into the model. Use of the MathProgBase interface also has the advantage of being independent of JuMP itself; users of MathProgBase solvers are free to implement their own evaluation routines instead of expressing their model in JuMP. You may use the `JuMP.build` method to ask JuMP to populate the "solver" without calling `optimize!`.

Raw expression input

In addition to the `@NLobjective` and `@NLconstraint` macros, it is also possible to provide Julia `Expr` objects directly by using `JuMP.setNLobjective` and `JuMP.addNLconstraint`. This input form may be useful if the expressions are generated programmatically. JuMP variables should be spliced into the expression object. For example:

```
@variable(m, 1 <= x[i=1:4] <= 5)
JuMP.setNLobjective(m, :Min, :($(x[1])*$(x[4])*($(x[1])+$(x[2])+$(x[3])) + $(x[3])))
JuMP.addNLconstraint(m, :($(x[1])*$(x[2])*$(x[3])*$(x[4]) >= 25))

# Equivalent form using traditional JuMP macros:
@NLobjective(m, Min, x[1]*x[4]*(x[1]+x[2]+x[3]) + x[3])
@NLconstraint(m, x[1]*x[2]*x[3]*x[4] >= 25)
```

See the Julia documentation for more examples and description of Julia expressions.

Citing JuMP

If you find JuMP useful in your work, we kindly request that you cite the following paper:

```
@article{DunningHuchetteLubin2017,
author = {Iain Dunning and Joey Huchette and Miles Lubin},
title = {JuMP: A Modeling Language for Mathematical Optimization},
journal = {SIAM Review},
volume = {59},
number = {2},
pages = {295-320},
year = {2017},
doi = {10.1137/15M1020575},
}
```

A preprint of this paper is freely available on arXiv.

For an earlier work where we presented a prototype implementation of JuMP, see here:

```
@article{LubinDunningIJOC,
author = {Miles Lubin and Iain Dunning},
title = {Computing in Operations Research Using Julia},
journal = {INFORMS Journal on Computing},
volume = {27},
number = {2},
pages = {238-248},
year = {2015},
doi = {10.1287/ijoc.2014.0623},
}
```

A preprint of this paper is also freely available.