Solvers and Solutions

Arpit Bhatia

May 30, 2019

The purpose of this part of the tutorial is to introduce you to solvers and how to use them with JuMP. We'll also learn what to do with a problem after the solver has finished optimizing it.

1 What is a Solver?

A solver is a software package that incorporates algorithms for finding solutions to one or more classes of problem. For example, GLPK, which we used in the previous tutorials is a solver for linear programming (LP) and mixed integer programming (MIP) problems. It incorporates algorithms such as the simplex method, interior-point method etc. JuMP currently supports a number of open-source and commercial solvers which can be viewed here.

2 What is MathOptInterface?

Each mathematical optimization solver API has its own concepts and data structures for representing optimization models and obtaining results. However, it is often desirable to represent an instance of an optimization problem at a higher level so that it is easy to try using different solvers. MathOptInterface (MOI) is an abstraction layer designed to provide an interface to mathematical optimization solvers so that users do not need to understand multiple solver-specific APIs. MOI can be used directly, or through a higher-level modeling interface like JuMP.

3 Interacting with solvers

JuMP models can be created in three different modes: `AUTOMATIC`, `MANUAL` and `DIRECT`. We'll use the following LP to illustrate them.

$$\max_{x,y} \quad x + 2y \tag{1}$$

$$\text{s.t.} \quad x + y \leq 1 \tag{2}$$

$$0 \leq x, y \leq 1 \tag{3}$$

```
using JuMP
using GLPK
```

3.1 AUTOMATIC Mode

3.1.1 With Optimizer

This is the easiest method to use a solver in JuMP. In order to do so, we simply set the solver inside the Model constructor.

```
model_auto = Model(with_optimizer(GLPK.Optimizer))
@variable(model_auto, 0 <= x <= 1)
@variable(model_auto, 0 <= y <= 1)
@constraint(model_auto, x + y <= 1)
@objective(model_auto, Max, x + 2y)
optimize!(model_auto)
objective_value(model_auto)
```

```
2.0
```

3.1.2 No Optimizer (at first)

It is also possible to create a JuMP model with no optimizer attached. After the model object is initialized empty and all its variables, constraints and objective are set, then we can attach the solver at `optimize!` time.

```
model_auto_no = Model()
@variable(model_auto_no, 0 <= x <= 1)
@variable(model_auto_no, 0 <= y <= 1)
@constraint(model_auto_no, x + y <= 1)
@objective(model_auto_no, Max, x + 2y)
optimize!(model_auto_no, with_optimizer(GLPK.Optimizer))
objective_value(model_auto_no)
```

```
2.0
```

Note that we can also enforce the automatic mode by passing `caching_mode = MOIU.AUTOMATIC` in the Model function call.

3.2 MANUAL Mode

This mode is similar to the `AUTOMATIC` mode, but there are less protections from the user getting errors from the solver API. On the other side, nothing happens silently, which might give the user more control. It requires attaching the solver before the solve step using the `MOIU.attach_optimizer()` function.

```
model_man = Model(with_optimizer(GLPK.Optimizer),caching_mode = MOIU.MANUAL)
@variable(model_man, 0 <= x <= 1)
@variable(model_man, 0 <= y <= 1)
@constraint(model_man, x + y <= 1)
@objective(model_man, Max, x + 2y)
MOIU.attach_optimizer(model_man)
optimize!(model_man)
objective_value(model_man)
```

```
2.0
```

3.3 `DIRECT` Mode

Some solvers are able to handle the problem data directly. This is common for LP/MIP solver but not very common for open-source conic solvers. In this case we do not set a optimizer, we set a backend which is more generic and is able to hold data and not only solving a model.

```
model_dir = direct_model(GLPK.Optimizer())
@variable(model_dir, 0 <= x <= 1)
@variable(model_dir, 0 <= y <= 1)
@constraint(model_dir, x + y <= 1)
@objective(model_dir, Max, x + 2y)
optimize!(model_dir)
objective_value(model_dir)
```

```
2.0
```

3.4 Solver Options

Many of the solvers also allow options to be passed in. However, these options are solver-specific. To find out the various options available, please check out the individual solver packages. Some examples for the CBC solver are given below.

```
using Cbc
```

To turn off printing (i.e. silence the solver),

```
model1 = Model(with_optimizer(Cbc.Optimizer, logLevel = 0))
```

To increase the maximum number of iterations

```
model2 = Model(with_optimizer(Cbc.Optimizer, max_iters=10000))
```

To set the solution timeout limit

```
model3 = Model(with_optimizer(Cbc.Optimizer, seconds=5))
```

4 Querying Solutions

So far we have seen all the elements and constructs related to writing a JuMP optimization model. In this section we reach the point of what to do with a solved problem. JuMP follows closely the concepts defined in MathOptInterface to answer user questions about a finished call to `optimize!(model)`. The three main steps in querying a solution are given below. We'll use the model we created in `AUTOMATIC` mode with an optimizer attached in this section.

4.1 Termination Statuses

Termination statuses are meant to explain the reason why the optimizer stopped executing in the most recent call to `optimize!`.

```
termination_status(model_auto)
```

```
OPTIMAL::TerminationStatusCode = 1
```

You can view the different termination status codes by referring to the docs or though checking the possible types using the below command.

```
display(typeof(MOI.OPTIMAL))
```

```
Enum MathOptInterface.TerminationStatusCode:
OPTIMIZE_NOT_CALLED = 0
OPTIMAL = 1
INFEASIBLE = 2
DUAL_INFEASIBLE = 3
LOCALLY_SOLVED = 4
LOCALLY_INFEASIBLE = 5
INFEASIBLE_OR_UNBOUNDED = 6
ALMOST_OPTIMAL = 7
ALMOST_INFEASIBLE = 8
ALMOST_DUAL_INFEASIBLE = 9
ALMOST_LOCALLY_SOLVED = 10
ITERATION_LIMIT = 11
TIME_LIMIT = 12
NODE_LIMIT = 13
SOLUTION_LIMIT = 14
MEMORY_LIMIT = 15
OBJECTIVE_LIMIT = 16
NORM_LIMIT = 17
OTHER_LIMIT = 18
SLOW_PROGRESS = 19
NUMERICAL_ERROR = 20
INVALID_MODEL = 21
INVALID_OPTION = 22
INTERRUPTED = 23
OTHER_ERROR = 24
```

4.2 Solution Statuses

These statuses indicate what kind of result is available to be queried from the model. It's possible that no result is available to be queried. We shall discuss more on the dual status and solutions in the Duality tutorial.

```
primal_status(model_auto)
dual_status(model_auto)
```

```
FEASIBLE_POINT::ResultStatusCode = 1
```

As we saw before, the result (solution) status codes can be viewed directly from Julia.

```
display(typeof(MOI.FEASIBLE_POINT))
```

```
Enum MathOptInterface.ResultStatusCode:
NO_SOLUTION = 0
FEASIBLE_POINT = 1
NEARLY_FEASIBLE_POINT = 2
INFEASIBLE_POINT = 3
INFEASIBILITY_CERTIFICATE = 4
NEARLY_INFEASIBILITY_CERTIFICATE = 5
UNKNOWN_RESULT_STATUS = 6
OTHER_RESULT_STATUS = 7
```

4.3 Obtaining Solutions

Provided the primal status is not `MOI.NO_SOLUTION`, we can inspect the solution values and optimal cost.

```
@show value(x)
```

```
value(x) = 0.0
```

```
@show value(y)
```

```
value(y) = 1.0
```

```
@show objective_value(model_auto)
```

```
objective_value(model_auto) = 2.0
2.0
```

Since it is possible that no solution is available to be queried from the model, calls to `value` may throw errors. Hence, it is recommended to check for the presence of solutions.

```
model_nosol = Model(with_optimizer(GLPK.Optimizer))
@variable(model_nosol, 0 <= x <= 1)
@variable(model_nosol, 0 <= y <= 1)
@constraint(model_nosol, x + y >= 3)
@objective(model_nosol, Max, x + 2y)
optimize!(model_nosol)

if termination_status(model_nosol) == MOI.OPTIMAL
    optimal_solution = value(x)
    optimal_objective = objective_value(model_nosol)
elseif termination_status(model_nosol) == MOI.TIME_LIMIT && has_values(model_nosol)
    suboptimal_solution = value(x)
    suboptimal_objective = objective_value(model_nosol)
else
    error("The model was not solved correctly.")
end
```

```
Error: The model was not solved correctly.
```