Integer Programming

Arpit Bhatia

June 22, 2019

While we already know how to set a variable as integer or binary in the `@variable` macro, this tutorial covers other JuMP features for integer programming along with some modelling techniques.

```
using JuMP
using GLPK
```

1 Modelling Integer Variables

1.1 Integer Variables using Constraints

We can add binary and integer restrictions to the domain of each variable using the `@constraint` macro as well.

```
model = Model()

@variable(model, x)
@variable(model, y)
@constraint(model, x in MOI.ZeroOne())
@constraint(model, y in MOI.Integer())
```

$yinteger$

1.2 Semi-Continuous Variables

A semi-continuous variable is a continuous variable between bounds $[l, u]$ that also can assume the value zero. ie.

$$x \in \{0\} \cup \{l, u\}$$

```
lower_bound = 7.45
upper_bound = 22.22
@variable(model, a)
@constraint(model, a in MOI.Semicontinuous(lower_bound, upper_bound))
```

$a \in MathOptInterface.SemicontinuousFloat64(7.45, 22.22)$

1.3 Semi-Integer Variables

A semi-integer variable is a variable which asummes integer values between bounds $[l, u]$ and can also assume the value zero. ie.

$$x \in \{0\} \cup (\{l, u\} \cap \mathbb{Z})$$

```
lower_bound = 5
upper_bound = 34
@variable(model, b)
@constraint(model, b in MOI.Semiinteger(lower_bound, upper_bound))
```

$b \in MathOptInterface.SemiintegerInt64(5, 34)$

Note that the bounds specified in `MOI.Semiinteger` must be integral otherwise it would throw an error.

2 Special Ordered Sets

2.1 Special Ordered Sets of type 1 (SOS1 or S1)

A Special Ordered Set of type 1 is a set of variables, at most one of which can take a non-zero value, all others being at 0. They most frequently apply where a set of variables are actually 0-1 variables: in other words, we have to choose at most one from a set of possibilities.

```
@variable(model, u[1:3])
@constraint(model, u in MOI.SOS1([1.0, 2.0, 3.0]))
```

$[u_1, u_2, u_3] \in MathOptInterface.SOS1Float64([1.0, 2.0, 3.0])$

Note that we have to pass MOI.SOS1 a weight vector which is essentially an ordering on the variables. If the decision variables are related and have a physical ordering, then the weight vector, although not used directly in the constraint, can help the solver make a better decision in the solution process.

2.2 Special Ordered Sets of type 2 (SOS2 or S2)

A Special Ordered Set of type 2 is a set of non-negative variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering.

```
@variable(model, v[1:3])
@constraint(model, v in MOI.SOS2([3.0, 1.0, 2.0]))
```

$[v_1, v_2, v_3] \in MathOptInterface.SOS2Float64([3.0, 1.0, 2.0])$

The ordering provided by the weight vector is more important in this case as the variables need to be consecutive according to the ordering. For example, in the above constraint, the possible pairs are:

- Consecutive
  - (`x[1]` and `x[3]`) as they correspond to 3 and 2 resp. and thus can be non-zero

  - (`x[2]` and `x[3]`) as they correspond to 1 and 2 resp. and thus can be non-zero

- Non-consecutive
  - (`x[1]` and `x[2]`) as they correspond to 3 and 1 resp. and thus cannot be non-zero

3 Modelling Logical Conditions

Generally, in a mathematical programming problem, all constraints must hold. However, we might want to have conditions where we have some logical conditions between constraints. In such cases, we can use binary variables for modelling logical conditions between constraints.

3.1 Disjunctive Constraints (OR)

Suppose that we are given two constraints $a'x \geq b$ and $c'x \geq d$, in which all components of a and c are non-negative. We would like to model a requirement that at least one of the two constraints is satisfied. For this, we defined a binary variable y and impose the constraints:

$$\begin{aligned} a'x &\geq yb \\ c'x &\geq (1-y)d \\ y &\in \{0,1\} \end{aligned}$$

```
a = rand(1:100,5,5)
c = rand(1:100,5,5)
b = rand(1:100,5)
d = rand(1:100,5)

model = Model()
@variable(model, x[1:5])
@variable(model, y, Bin)
@constraint(model, a * x .>= y .* b)
@constraint(model, c * x .>= (1 - y) .* d)
```

```
5-element Array{ConstraintRef{Model,MathOptInterface.ConstraintIndex{MathOp
tInterface.ScalarAffineFunction{Float64},MathOptInterface.GreaterThan{Float
64}},ScalarShape},1}:
 69 x[1] + 23 x[2] + 75 x[3] + 93 x[4] + 38 x[5] + 36 y ≥ 36.0
 x[1] + 46 x[2] + 86 x[3] + 68 x[4] + 89 x[5] + 10 y ≥ 10.0
 67 x[1] + 57 x[2] + 27 x[3] + 20 x[4] + 86 x[5] + 71 y ≥ 71.0
 11 x[1] + 99 x[2] + 42 x[3] + 99 x[4] + 17 x[5] + 76 y ≥ 76.0
 85 x[1] + 7 x[2] + 94 x[3] + 98 x[4] + 91 x[5] + 17 y ≥ 17.0
```

3.2 Conditional Constraints ($\implies$)

Suppose we want to model that a certain linear inequality must be satisfied when some other event occurs. i.e. for a binary variable z, we want to model the implication

$$z = 1 \implies a^T x \leq b$$

If we know in advance an upper bound $a^T x \leq b$. Then we can write the above as a linear inequality

$$a^T x \leq b + M(1 - z)$$

```
a = rand(1:100,5,5)
b = rand(1:100,5)
m = rand(10000:11000,5)

model = Model()
@variable(model, x[1:5])
@variable(model, z, Bin)
@constraint(model, a * x .<=  b .+ (m .* (1 - z)))
# If z was a regular Julia variable, we would not have had to use the vectorized dot
    operator
```