Variables, Constraints and Objective

Arpit Bhatia

May 30, 2019

While the last tutorial introduced you to basics of of JuMP code, this tutorial will go in depth focusing on how to work with different parts of a JuMP program.

1 Variables

1.1 Variable Bounds

All of the variables we have created till now have had a bound. We can also create a free variable.

```
@variable(model, freex)
```

$$freex$$

While creating a variable, instead of using the <= and >= syntax, we can also use the `lower_bound` and `upper_bound` keyword arguments.

```
@variable(model, altx, lower_bound=1, upper_bound=2)
```

$$altx$$

We can query whether a variable has a bound using the `has_lower_bound` and `has_upper_bound` functions. The values of the bound can be obtained using the `lower_bound` and `upper_bound` functions.

```
has_upper_bound(altx)
```

```
true
```

```
upper_bound(altx)
```

```
2.0
```

Note querying the value of a bound that does not exist will result in an error.

```
lower_bound(freex)
```

```
Error: Variable freex does not have a lower bound.
```

JuMP also allows us to change the bounds on variable. We will learn this in the problem modification tutorial.

1.2 Containers

We have already seen how to add a single variable to a model using the `@variable` macro. Let's now look at more ways to add variables to a JuMP model. JuMP provides data structures for adding collections of variables to a model. These data structures are reffered to as Containers and are of three types - `Arrays`, `DenseAxisArrays`, and `SparseAxisArrays`.

1.2.1 Arrays

JuMP arrays are created in a similar syntax to Julia arrays with the addition of specifying that the indices start with 1. If we do not tell JuMP that the indices start at 1, it will create a DenseAxisArray instead.

```
@variable(model, a[1:2, 1:2])
```

```
2×2 Array{JuMP.VariableRef,2}:
 a[1,1]  a[1,2]
 a[2,1]  a[2,2]
```

An n-dimensional variable $x \in R^n$ having a bound $l \preceq x \preceq u$ $(l, u \in R^n)$ is added in the following manner.

```
n = 10
l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
u = [10; 11; 12; 13; 14; 15; 16; 17; 18; 19]

@variable(model, l[i] <= x[i=1:n] <= u[i])
```

```
10-element Array{JuMP.VariableRef,1}:
 x[1]
 x[2]
 x[3]
 x[4]
 x[5]
 x[6]
 x[7]
 x[8]
 x[9]
 x[10]
```

Note that while working with Containers, we can also create variable bounds depending upon the indices

```
@variable(model, y[i=1:2, j=1:2] >= 2i + j)
```

```
2×2 Array{JuMP.VariableRef,2}:
 y[1,1]  y[1,2]
 y[2,1]  y[2,2]
```

1.2.2 DenseAxisArrays

DenseAxisArrays are used when the required indices are not one-based integer ranges. The syntax is similar except with an arbitrary vector as an index as opposed to a one-based range.

An example where the indices are integers but do not start with one.

```
@variable(model, z[i=2:3, j=1:2:3] >= 0)
```

```
2-dimensional DenseAxisArray{JuMP.VariableRef,2,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, 1:2:3
And data, a 2×2 Array{JuMP.VariableRef,2}:
 z[2,1]  z[2,3]
 z[3,1]  z[3,3]
```

Another example where the indices are an arbitrary vector.

```
@variable(model, w[1:5,["red","blue"]] <= 1)
```

```
2-dimensional DenseAxisArray{JuMP.VariableRef,2,...} with index sets:
    Dimension 1, 1:5
    Dimension 2, ["red", "blue"]
And data, a 5×2 Array{JuMP.VariableRef,2}:
 w[1,red]  w[1,blue]
 w[2,red]  w[2,blue]
 w[3,red]  w[3,blue]
 w[4,red]  w[4,blue]
 w[5,red]  w[5,blue]
```

1.2.3 SparseAxisArrays

SparseAxisArrays are created when the indices do not form a rectangular set. For example, this applies when indices have a dependence upon previous indices (called triangular indexing).

```
@variable(model, u[i=1:3, j=i:5])
```

```
JuMP.Containers.SparseAxisArray{JuMP.VariableRef,2,Tuple{Any,Any}} with 12
entries:
  [2, 3]  =  u[2,3]
  [2, 2]  =  u[2,2]
  [2, 5]  =  u[2,5]
  [1, 4]  =  u[1,4]
  [3, 3]  =  u[3,3]
  [1, 3]  =  u[1,3]
  [2, 4]  =  u[2,4]
  [1, 1]  =  u[1,1]
  [1, 2]  =  u[1,2]
  [1, 5]  =  u[1,5]
  [3, 4]  =  u[3,4]
  [3, 5]  =  u[3,5]
```

We can also conditionally create variables by adding a comparison check that depends upon the named indices and is separated from the indices by a semi-colon (;).

```
@variable(model, v[i=1:9; mod(i, 3)==0])
```

```
JuMP.Containers.SparseAxisArray{JuMP.VariableRef,1,Tuple{Any}} with 3 entri
es:
  [9]  =  v[9]
  [3]  =  v[3]
  [6]  =  v[6]
```

1.3 Variable Types

The last arguement to the `@variable` macro is usually the variable type. Here we'll look at how to specifiy he variable type.

1.3.1 Integer Variables

Integer optimization variables are constrained to the set $x \in Z$

```
@variable(model, intx, Int)
```

or

```
@variable(model, intx, integer=true)
```

$$intx$$

1.3.2 Binary Variables

Binary optimization variables are constrained to the set $x \in \{0, 1\}$.

```
@variable(model, binx, Bin)
```

or

```
@variable(model, binx, binary=true)
```

$$binx$$

1.3.3 Semidefinite variables

JuMP also supports modeling with semidefinite variables. A square symmetric matrix X is positive semidefinite if all eigenvalues are nonnegative.

```
@variable(model, psdx[1:2, 1:2], PSD)
```

```
2×2 LinearAlgebra.Symmetric{JuMP.VariableRef,Array{JuMP.VariableRef,2}}:
 psdx[1,1]  psdx[1,2]
 psdx[1,2]  psdx[2,2]
```

We can also impose a weaker constraint that the square matrix is only symmetric (instead of positive semidefinite) as follows:

```
@variable(model, symx[1:2, 1:2], Symmetric)
```

```
2×2 LinearAlgebra.Symmetric{JuMP.VariableRef,Array{JuMP.VariableRef,2}}:
 symx[1,1]  symx[1,2]
 symx[1,2]  symx[2,2]
```

2 Constraints

2.1 Constraint References

While calling the `@constraint` macro, we can also set up a constraint reference. Such a refference is useful for obtaining additional information about the constraint such as its dual.

```
@constraint(model, con, x <= 4)
```

con : $x \leq 4.0$

2.2 Containers

Just as we had containers for variables, JuMP also provides `Arrays`, `DenseAxisArrays`, and `SparseAxisArrays` for storing collections of constraints. Examples for each container type are given below.

2.2.1 Arrays

```
@constraint(model, acon[i = 1:3], i * x <= i + 1)
```

```
3-element Array{JuMP.ConstraintRef{JuMP.Model,C,Shape} where Shape<:JuMP.Ab
stractShape where C,1}:
 acon[1] : x ≤ 2.0
 acon[2] : 2 x ≤ 3.0
 acon[3] : 3 x ≤ 4.0
```

2.2.2 DenseAxisArrays

```
@constraint(model, dcon[i = 1:2, j = 2:3], i * x <= j + 1)
```

```
2-dimensional DenseAxisArray{JuMP.ConstraintRef{JuMP.Model,C,Shape} where S
hape<:JuMP.AbstractShape where C,2,...} with index sets:
    Dimension 1, 1:2
    Dimension 2, 2:3
And data, a 2×2 Array{JuMP.ConstraintRef{JuMP.Model,C,Shape} where Shape<:J
uMP.AbstractShape where C,2}:
 dcon[1,2] : x ≤ 3.0    dcon[1,3] : x ≤ 4.0
 dcon[2,2] : 2 x ≤ 3.0  dcon[2,3] : 2 x ≤ 4.0
```

2.2.3 SparseAxisArrays

```
@constraint(model, scon[i = 1:2, j = 1:2; i != j], i * x <= j + 1)
```

```
JuMP.Containers.SparseAxisArray{JuMP.ConstraintRef{JuMP.Model,C,Shape} wher
e Shape<:JuMP.AbstractShape where C,2,Tuple{Any,Any}} with 2 entries:
  [1, 2]  =  scon[1,2] : x ≤ 3.0
  [2, 1]  =  scon[2,1] : 2 x ≤ 2.0
```

2.3 Constraints in a Loop

We can add constraints using regular Julia loops

```
for i in 1:3
    @constraint(model, 6*x + 4*y >= 5*i)
end
```

or use for each loops inside the `@constraint` macro

```
@constraint(model, conRef3[i in 1:3], 6*x + 4*y >= 5*i)
```

```
3-element Array{JuMP.ConstraintRef{JuMP.Model,C,Shape} where Shape<:JuMP.Ab
stractShape where C,1}:
 conRef3[1] : 6 x + 4 y ≥ 5.0
 conRef3[2] : 6 x + 4 y ≥ 10.0
 conRef3[3] : 6 x + 4 y ≥ 15.0
```

We can also created constraints such as

$$\sum_{i=1}^{10} z_i \leq 1$$

```
@constraint(model, sum(z[i] for i in 1:10) <= 1)
```

$z_1 + z_2 + z_3 + z_4 + z_5 + z_6 + z_7 + z_8 + z_9 + z_{10} \leq 1.0$

3 Objective

While the recommended way to set the objective is with the @objective macro, the functions `set_objective_sense` and `set_objective_function` provide an equivalent lower-level interface.

```
using GLPK

mymodel = Model(with_optimizer(GLPK.Optimizer))
@variable(mymodel, x >= 0)
@variable(mymodel, y >= 0)
set_objective_sense(mymodel, MOI.MIN_SENSE)
set_objective_function(mymodel, x + y)

optimize!(mymodel)

@show objective_value(mymodel)
```

```
objective_value(mymodel) = 0.0
0.0
```

To query the objective function from a model, we use the `objective_sense`, `objective_function`, and `objective_function_type` functions.

```
objective_sense(mymodel)
```

```
MIN_SENSE::OptimizationSense = 0
```

```
objective_function(mymodel)
```

$$x + y$$

```
objective_function_type(mymodel)
```

```
JuMP.GenericAffExpr{Float64,JuMP.VariableRef}
```

4 Vectorized Constraints and Objective

We can also add constraints and objective to JuMP using vectorized linear algebra. We'll illustrate this by solving an LP in standard form i.e.

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \succeq 0 \\ & x \in \mathbb{R}^n \end{aligned}$$

```
vectormodel = Model(with_optimizer(GLPK.Optimizer))

A= [ 1 1 9 5;
     3 5 0 8;
     2 0 6 13]

b = [7; 3; 5]

c = [1; 3; 5; 2]

@variable(vectormodel, x[1:4] >= 0)
@constraint(vectormodel, A * x .== b)
@objective(vectormodel, Min, c' * x)

optimize!(vectormodel)

@show objective_value(vectormodel)

objective_value(vectormodel) = 4.9230769230769225
4.9230769230769225
```