Tutorial IV.II - Variables and Bounds in JuMP

Applied Optimization with Julia

Introduction

This Julia script is an interactive tutorial to introduce you to advanced variable handling in JuMP. You'll learn about creating variables in containers, managing different types of variables, and working with indexed variables. Follow the instructions, write your code in the designated code blocks, and confirm your understanding with `@assert` statements. Make sure to have the JuMP package installed to follow this tutorial.

```julia
using JuMP
```

Now, we need a model. As we don't solve anything in this tutorial, we don't need to add the solver and can solely define the abstract model.

```julia
new_model = Model()
```

```
A JuMP Model
  solver: none
  objective_sense: FEASIBILITY_SENSE
  num_variables: 0
  num_constraints: 0
  Names registered in the model: none
```

Section 1 - Managing Different Types of Variables

Why Different Variable Types?

- **Continuous variables**: Represent quantities that can take any real value within a range (e.g., amount of raw materials in a production process).
- **Integer variables**: Represent indivisible quantities (e.g., number of products manufactured).
- **Binary variables**: Represent yes/no decisions (e.g., whether to build a facility at a specific location).

For example:

```
@variable(model, variableName)
```

This defines a continuous variable without any bound.

```
@variable(model, 0 <= variableName <= 1)
```

This defines a continuous variable in an interval.

```
@variable(model, 0 <= variableName, Bin)
```

This defines a binary variable.

```
@variable(model, 0 <= variableName, Int)
```

This defines an integer variable. Note that you will have to change `modelName` and `variableName` according to your instance.

Exercise 1.1 - Create Variables

Create continuous, integer, and binary variables. Create three different variables of different types without any additional bound: `continuous_var` as a continuous variable for `new_model`, `integer_var` as an integer variable for `new_model`, and `binary_var` as a binary variable for `new_model`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert typeof(continuous_var) == VariableRef && typeof(integer_var) == VariableRef
@assert typeof(binary_var) == VariableRef
println("Continuous, integer, and binary variables created successfully!")
```

Section 2 - Creating Variables in Containers

When dealing with large-scale problems, creating variables individually becomes impractical. Containers like arrays, matrices, and indexed sets allow us to:

1. Efficiently create and manage many variables.
2. Represent complex relationships between variables.
3. Simplify model formulation for problems with repetitive structures.

For example:

```julia
@variable(modelName, variableName[1:20], Bin)
```

This would create a container with 20 variables. To create a set based on a range, we could do:

```julia
new_range = 1:100
@variable(modelName, variableName[i in new_range] >= 0)
```

This would create a container with 100 continuous variables larger than 0.

For a container with multiple dimensions:

```julia
@variable(modelName, variableName[1:30, 1:30])
```

This would create a container with a matrix of continuous variables without any bound. Note that you will have to change `modelName` and `variableName` according to your instance.

Exercise 2.1 - Create an Array

Create an array `X` for `new_model` containing 8 variables with non-negative lower bounds.

```julia
# YOUR CODE BELOW
```

```julia
# Test your answer
@assert length(X) == 8 && all(lower_bound(X[i]) == 0 for i in 1:8)
println("Array of variables created successfully!")
```

Exercise 2.2 - Create a Matrix

Create a matrix `Y` for `new_model` containing binary variables with a size of 3 x 3.

```julia
# YOUR CODE BELOW
# Test your answer
@assert size(Y, 1) == 3 && size(Y, 2) == 3
@assert all(has_lower_bound(Y[i, j]) == false for i in 1:3, j in 1:3)
@assert all(has_upper_bound(Y[i, j]) == false for i in 1:3, j in 1:3)
println("Matrix of variables created successfully!")
```

Exercise 2.3 - Create Indexed Variables

Create indexed variables based on a set. Create variables `W` indexed by set `N` of size 1, 2, ..., 6 with non-negative lower bounds.

```julia
# YOUR CODE BELOW
```

```julia
# Test your answer
@assert length(W) == 6 && all(lower_bound(W[i]) == 0 for i in N)
println("Indexed variables created successfully!")
```

Exercise 2.4 - Create Nested Containers

Create a 3-dimensional array `Z` for `new_model` containing integer variables with dimensions 2 x 3 x 4, where each variable has a lower bound of 0 and an upper bound of 10.

```julia
# YOUR CODE BELOW
```

```julia
# Test your answer
@assert size(Z) == (2, 3, 4)
@assert all(is_integer(Z[i,j,k]) && lower_bound(Z[i,j,k]) == 0 && upper_bound(Z[i,j,k])
 ↪  == 10
           for i in 1:2, j in 1:3, k in 1:4)
println("3D array of integer variables created successfully!")
```

Conclusion

Fantastic! You've completed the tutorial on advanced variables in JuMP. You've learned how to create variables in containers, manage different types of variables, and work with indexed variables. Continue to the next file to learn more.

Solutions

You will likely find solutions to most exercises online. However, I strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them in next week's tutorial. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.