Tutorial III.II - Variables and Bounds in JuMP

Optimization with Julia

Introduction

Welcome to this beginner-friendly tutorial on variables and bounds in JuMP! In this lesson, we'll explore different types of variables and how to set limits (or bounds) on them. Don't worry if you're new to optimization - we'll explain everything step by step using real-world examples.

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.

Let's start by loading the JuMP package:

```
using JuMP
```

Now, let's create a model that we'll use throughout this tutorial:

```
model = Model()
println("Great! We've created a new optimization model.")
```

```
Great! We've created a new optimization model.
```

Section 1 - Understanding Different Types of Variables

In optimization problems, we often need to represent different kinds of decisions. JuMP allows us to use three main types of variables:

1. **Continuous variables**: These can take any real value within a range. Example: The amount of water in a reservoir (can be any number, like 3.7 liters).
2. **Integer variables**: These can only be whole numbers. Example: The number of cars produced in a factory (we can't produce half a car!).
3. **Binary variables**: These can only be 0 or 1. Example: Whether to build a new store in a location (yes = 1, no = 0).

Let's see how to create each type:

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

This defines a continuous variable without any bound.

```
@variable(model, 0 <= variableName2 <= 1)
has_lower_bound(variableName2) && has_upper_bound(variableName2)
```

This defines a continuous variable in an interval.

```
@variable(model, variableName3, Bin)
is_binary(variableName3)
```

This defines a binary variable.

```
@variable(model, 0 <= variableName4, Int)
is_integer(variableName4)
```

This defines an integer variable.

> **Note**
>
> Note that you will have to change `model` and `variableName` according to your instance.

Exercise 1.1 - Create Variables

Now it's your turn! Create three variables:

1. A continuous variable called `water_amount`

2. An integer variable called `cars_produced`
3. A binary variable called `build_store`

```
# YOUR CODE BELOW
# Hint: Use the @variable macro three times, once for each variable
```

```
# Test your answer
@assert typeof(water_amount) == VariableRef && !is_integer(water_amount) &&
 ↪  !is_binary(water_amount)
@assert typeof(cars_produced) == VariableRef && is_integer(cars_produced)
@assert typeof(build_store) == VariableRef && is_binary(build_store)
println("Excellent work! You've successfully created continuous, integer, and binary
 ↪  variables.")
```

Section 2 - Creating Variables in Containers

When we have many similar variables, it's helpful to group them together. JuMP allows us to use containers like arrays and matrices for this purpose. For example:

```
@variable(model, variableName5[1:20], Bin)
```

```
20-element Vector{VariableRef}:
 variableName5[1]
 variableName5[2]
 variableName5[3]
 variableName5[4]
 variableName5[5]
 variableName5[6]
 variableName5[7]
 variableName5[8]
 variableName5[9]
 variableName5[10]
 variableName5[11]
 variableName5[12]
 variableName5[13]
 variableName5[14]
 variableName5[15]
 variableName5[16]
 variableName5[17]
 variableName5[18]
 variableName5[19]
 variableName5[20]
```

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

```
new_range = 1:100
@variable(model, variableName6[i in new_range] >= 0)
```

```
1-dimensional DenseAxisArray{JuMP.VariableRef,1,...} with index sets:
    Dimension 1, 1:100
And data, a 100-element Vector{JuMP.VariableRef}:
 variableName6[1]
 variableName6[2]
 variableName6[3]
 variableName6[4]
 variableName6[5]
 variableName6[6]
```

```
 variableName6[7]
 variableName6[8]
 variableName6[9]
 variableName6[10]

 variableName6[92]
 variableName6[93]
 variableName6[94]
 variableName6[95]
 variableName6[96]
 variableName6[97]
 variableName6[98]
 variableName6[99]
 variableName6[100]
```

This would create a container with 100 continuous variables larger than 0. For a container with multiple dimensions:

```
@variable(model, variableName7[1:30, 1:30])
```

```
30×30 Matrix{VariableRef}:
 variableName7[1,1]   variableName7[1,2]   …  variableName7[1,30]
 variableName7[2,1]   variableName7[2,2]      variableName7[2,30]
 variableName7[3,1]   variableName7[3,2]      variableName7[3,30]
 variableName7[4,1]   variableName7[4,2]      variableName7[4,30]
 variableName7[5,1]   variableName7[5,2]      variableName7[5,30]
 variableName7[6,1]   variableName7[6,2]   …  variableName7[6,30]
 variableName7[7,1]   variableName7[7,2]      variableName7[7,30]
 variableName7[8,1]   variableName7[8,2]      variableName7[8,30]
 variableName7[9,1]   variableName7[9,2]      variableName7[9,30]
 variableName7[10,1]  variableName7[10,2]     variableName7[10,30]

 variableName7[22,1]  variableName7[22,2]     variableName7[22,30]
 variableName7[23,1]  variableName7[23,2]     variableName7[23,30]
 variableName7[24,1]  variableName7[24,2]     variableName7[24,30]
 variableName7[25,1]  variableName7[25,2]     variableName7[25,30]
 variableName7[26,1]  variableName7[26,2]  …  variableName7[26,30]
 variableName7[27,1]  variableName7[27,2]     variableName7[27,30]
 variableName7[28,1]  variableName7[28,2]     variableName7[28,30]
 variableName7[29,1]  variableName7[29,2]     variableName7[29,30]
 variableName7[30,1]  variableName7[30,2]     variableName7[30,30]
```

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

Exercise 2.1 - Create an Array

Imagine you're planning production for a week. Create an array `daily_production` with 7 non-negative variables, one for each day of the week.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert length(daily_production) == 7
```

```
@assert all(lower_bound(daily_production[i]) == 0 for i in 1:7)
println("Great job! You've created an array of 7 non-negative variables for daily
 ↪  production.")
```

Exercise 2.2 - Create a Matrix of Variables

Now, imagine you're deciding whether to stock 4 different products in 3 different stores. Create a 3x4 matrix of binary variables called `stock_decision`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert size(stock_decision) == (3, 4)
@assert all(is_binary(stock_decision[i,j]) for i in 1:3, j in 1:4)
println("Excellent! You've created a 3x4 matrix of binary variables for stocking
 ↪  decisions.")
```

Section 3 - Setting Bounds on Variables

Often, we know that a variable can't go below or above certain values. We can set these limits (called bounds) when we create the variable.

For example, if a factory can produce between 100 and 500 units:

```
@variable(model, 100 <= production <= 500)
```

Or if we know a percentage must be between 0 and 100:

```
@variable(model, 0 <= percentage <= 100)
```

Exercise 3.1 - Set Bounds on a Variable

Create a variable `temperature` that represents the temperature setting on a thermostat. It should be between 0 and 37 degrees.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert lower_bound(temperature) == 0
@assert upper_bound(temperature) == 37
println("Well done! You've created a variable for temperature with appropriate bounds.")
```

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 find the solutions to all exercises online here in the `solutions` folders for each part. 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.