Tutorial IV.I - Introduction to Mathematical Optimization with JuMP and HiGHS

Applied Optimization with Julia

Introduction

This Julia script is an interactive tutorial to introduce you to mathematical optimization using JuMP and the HiGHS solver. Follow the instructions, write your code in the designated code blocks, and confirm your understanding with @assert statements.

JuMP is a general framework to formulate mathematical models in Julia, HiGHS is a open source solver with a MIT license and draws quiet some attention, as it is pretty fast for an open source solver. Nonetheless, we could also use other solvers in JuMP by only changing of a few lines of code, making it more versatile as for example GurobiPy in Python.

If you ever feel like you need more information, the JuMP Documentation is a great ressource: JuMP Documentation

We are going to structure this tutorial based on a small example instance:

- Suppose we are planning the production of two products
  - The profit from each unit of Product A is 100
  - The profit from each unit of Product B is 150
- The production is constrained by the available hours in two departments
  - Each unit of Product A requires 2 hours of Cutting and 4 hours of Finishing
  - Each unit of Product B requires 4 hours of Cutting and 3 hours of Finishing
  - 40 hours are available in total in the Cutting department
  - 60 hours are available in total in the Finishing department
- We want to maximize the profit given these constraints

If you followed the struture of the tutorials, you first have to install the JuMP Package and the HiGHS Package before you can start using it:

```
import Pkg; Pkg.add(["JuMP","HiGHS"])
```

```
using JuMP, HiGHS
```

Section 1 - Defining an Optimization Model

An optimization model in JuMP is defined within a Model object.

```
model = Model(HiGHS.Optimizer)
println("Optimization model created successfully!")
```

```
Optimization model created successfully!
```

Section 2 - Adding Variables

In optimization models, variables represent the quantities we want to determine. In JuMP, we use the `@variable` macro to define these variables.

To define a variable for the model, we use the syntax:

```
@variable(model_name, variable_name, [additional_properties])
```

Where: - `model_name` is the name of your JuMP model - `variable_name` is what you want to call your variable - `[additional_properties]` can include bounds or variable types

For example, to create a continuous variable that's greater than or equal to 0:

```
@variable(model_name, variable_name >= 0)
```

This defines a continuous variable that's equal to or larger than 0.

Exercise 2.1 - Create Variables

Create two continous variables equal or larger 0 called `productA` and `productB` that represent the the number of units produced in our problem for our model `model`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert @isdefined productA
@assert typeof(productA) == VariableRef
@assert has_upper_bound(productA) == false
@assert has_lower_bound(productA) == true
@assert lower_bound(productA) == 0
@assert @isdefined productA
@assert typeof(productB) == VariableRef
@assert has_upper_bound(productB) == false
@assert has_lower_bound(productB) == true
@assert lower_bound(productB) == 0
println("Variables added to the model successfully!")
```

Section 3 - Adding Constraints

Constraints are conditions that limit the possible values of the variables in our optimization problem. In JuMP, we use the `@constraint` macro to define these constraints.

The general syntax for adding a constraint is:

```
@constraint(model_name, constraint_name, constraint_expression)
```

Where: - `model_name` is the name of your JuMP model - `constraint_name` is a label you give to the constraint (optional, but useful for reference) - `constraint_expression` is the mathematical expression of the constraint

For example:

```
constraint(model_name, constraint_name, 4 * variable_name <= 100)
```

This defines a constraint that ensures, that the variable `variable_name` can maximally be 25. Note, that you will have to change `model_name`, `constraint_name` and `variable_name` according to your instance.

Exercise 3.1 - Create Constraints

Create two constraints based on the on the Cutting and Finishing department hours of the problem description in this tutorial. Call the first constraint `cutting_constraint` and the second constraint `finishing_constraint`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert is_valid(model, cutting_constraint)
@assert is_valid(model, finishing_constraint)
println("Constraints added to the model successfully!")
println("Note, that only the existence of these constraints was checked!")
println("The optimization later will show, whether the formulation was correct.")
```

Section 4 - Defining the Objective Function

The objective function represents the goal of our optimization problem - what we want to maximize or minimize. In JuMP, we use the `@objective` macro to define this function.

The general syntax for defining an objective function is:

```
@objective(model_name, optimization_direction, objective_expression)
```

Where: - `model_name` is the name of your JuMP model - `optimization_direction` is either `Max` for maximization or `Min` for minimization - `objective_expression` is the mathematical expression of the objective function

For example:

```
@objective(model_name, Max, 2*variableA + 3*variableB)
```

This defines an objective function that maximizes something based on the values of `variableA` and `variableB`.

Exercise 4.1 - Create the Objective Function

Create the objective function based on the problem description of this tutorial. The objective is to maximize profit based on the values of `productA` and `productB`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert typeof(objective_function(model)) == AffExpr
println("An objective function defined successfully!")
println("The optimization later will show, whether the formulation was correct.")
```

Section 5 - Solving the Model

Once we have defined our variables, constraints, and objective function, we can solve the optimization model. Here's how to do it step by step:

1. Solve the model:

```
optimize!(model)
```

2. Check the status of the solution:

```
termination_status(model)
```

3. If the solution is optimal, we can retrieve the optimal values of our variables and the objective function. If the time limit is reached, we can check whether a primal solution is available. We do so by checking whether the model has values.

```
begin
    if termination_status(model) == OPTIMAL
        println("Solution is optimal")
    elseif termination_status(model) == TIME_LIMIT && has_values(model)
        println("Solution is suboptimal due to a time limit,
            but a primal solution is available")
    else
        error("The model was not solved correctly.")
    end
    println("Objective value = ", objective_value(model))
end
```

We can then get the values of the variables as follows:

```
println("Product A quantity: $(value(productA))")
println("Product B quantity: $(value(productB))")
```

> **Note**
>
> Note, that the values can be slightly off due to the nature of Float64 numbers!

The following code block tests whether the solution is correct or whether you have made a mistake in the formulation of the problem.

```
# Test your answer
@assert termination_status(model) == MOI.OPTIMAL "Sorry, something didn't work out as the
 ↪  model status is $termination_status(model)".
println("Solution: Product A = ", val_productA, ", Product B = ", val_productB)
@assert val_productA   12 atol=1e-4 "Although you have a solution, val_productA should be
 ↪  12 not $val_productA"
```

```
@assert val_productB  4 atol=1e-4 "Although you have a solution, val_productB should be
 ↪  4 not $val_productB"
println("You have solved the model correctly!")
```

Conclusion

Excellent! You've completed the tutorial on mathematical optimization with JuMP and HiGHS. You've learned how to define optimization models, add variables and constraints, define an objective function, and solve the model. 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.