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

Applied Optimization with Julia

Introduction

Welcome to this beginner-friendly tutorial on mathematical optimization using JuMP and the HiGHS solver in Julia! Don't worry if these terms sound unfamiliar – we'll explain everything step by step.

In this tutorial, you'll learn how to:

1. Set up a simple optimization problem
2. Define variables and constraints
3. Create an objective function
4. Solve the problem and interpret the results

We'll use a real-world example to make these concepts more relatable. Imagine you're managing a small factory that produces two types of products. Your goal is to maximize profit while working within certain limitations. This is exactly the kind of problem that mathematical optimization can solve!

What is JuMP?

JuMP (Julia for Mathematical Programming) is a powerful tool that helps us describe optimization problems in a way that computers can understand and solve. Think of it as a translator between your business problem and the mathematical solver.

What is HiGHS?

HiGHS is an open-source solver that can find solutions to the optimization problems we describe using JuMP. It's like a very smart calculator that can handle complex problems quickly and efficiently.

Our Example Problem

Let's break down our factory management problem:

- You produce two products: Product A and Product B
- Each product gives you a different profit:
  - Product A: 100 profit per unit
  - Product B: 150 profit per unit
- You have two departments: Cutting and Finishing
- Each product requires different amounts of time in each department:
  - Product A: 2 hours in Cutting, 4 hours in Finishing
  - Product B: 4 hours in Cutting, 3 hours in Finishing

- You have limited time available in each department:
  - Cutting: 40 hours total
  - Finishing: 60 hours total

Your goal is to decide how many of each product to make to maximize your total profit, while not exceeding the available time in each department.

Setting Up

First, we need to install and load the necessary packages. If you haven't already installed JuMP and HiGHS, run the following code:

```
import Pkg
Pkg.activate("applied-optimization")
Pkg.add(["JuMP","HiGHS"])
```

Now, let's load these packages:

```
using JuMP, HiGHS
```

```
┌ Error: Error during loading of extension SpecialFunctionsExt of
ColorVectorSpace, use `Base.retry_load_extensions()` to retry.
│   exception =
│    1-element ExceptionStack:
│    ArgumentError: Package SpecialFunctionsExt
[997ecda8-951a-5f50-90ea-61382e97704b] is required but does not seem to be
installed:
│     - Run `Pkg.instantiate()` to install all recorded dependencies.
│
│    Stacktrace:
│      [1] __require_prelocked(pkg::Base.PkgId, env::Nothing)
│        @ Base ./loading.jl:2587
│      [2] _require_prelocked(uuidkey::Base.PkgId, env::Nothing)
│        @ Base ./loading.jl:2465
│      [3] _require_prelocked(uuidkey::Base.PkgId)
│        @ Base ./loading.jl:2459
│      [4] run_extension_callbacks(extid::Base.ExtensionId)
│        @ Base ./loading.jl:1579
│      [5] run_extension_callbacks(pkgid::Base.PkgId)
│        @ Base ./loading.jl:1616
│      [6] run_package_callbacks(modkey::Base.PkgId)
│        @ Base ./loading.jl:1432
│      [7] _require_search_from_serialized(pkg::Base.PkgId,
sourcepath::String, build_id::UInt128, stalecheck::Bool;
reasons::Dict{String, Int64}, DEPOT_PATH::Vector{String})
│        @ Base ./loading.jl:2106
│      [8] _require_search_from_serialized
│        @ ./loading.jl:1981 [inlined]
│      [9] __require_prelocked(pkg::Base.PkgId, env::String)
│        @ Base ./loading.jl:2599
│     [10] _require_prelocked(uuidkey::Base.PkgId, env::String)
│        @ Base ./loading.jl:2465
```

```
│     [11] macro expansion
│        @ ./loading.jl:2393 [inlined]
│     [12] macro expansion
│        @ ./lock.jl:376 [inlined]
│     [13] __require(into::Module, mod::Symbol)
│        @ Base ./loading.jl:2358
│     [14] require(into::Module, mod::Symbol)
│        @ Base ./loading.jl:2334
│     [15] eval(m::Module, e::Any)
│        @ Core ./boot.jl:489
│     [16] include_string(mapexpr::typeof(REPL.softscope), mod::Module,
code::String, filename::String)
│        @ Base ./loading.jl:2843
│     [17] softscope_include_string(m::Module, code::String,
filename::String)
│        @ SoftGlobalScope ~/.julia/packages/SoftGlobalScope/u4UzH/src/
SoftGlobalScope.jl:65
│     [18] execute_request(socket::ZMQ.Socket, msg::IJulia.Msg)
│        @ IJulia ~/.julia/packages/IJulia/eenvU/src/execute_request.jl:81
│     [19] eventloop(socket::ZMQ.Socket)
│        @ IJulia ~/.julia/packages/IJulia/eenvU/src/eventloop.jl:14
│     [20] (::IJulia.var"#waitloop##2#waitloop##3")()
│        @ IJulia ~/.julia/packages/IJulia/eenvU/src/eventloop.jl:58
└ @ Base loading.jl:1589
```

Great! We're now ready to start building our optimization model.

Section 1 - Defining an Optimization Model

The first step in solving an optimization problem is to create a model. This model will contain all the information about our problem: the variables, constraints, and objective.

In JuMP, we create a model like this:

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

```
Optimization model created successfully!
```

Let's break this down: - `Model()` creates a new optimization model - `HiGHS.Optimizer` tells JuMP to use the HiGHS solver for this model

Think of this as creating a blank canvas where we'll paint our optimization problem.

Section 2 - Adding Variables

Now that we have our model, we need to define our variables. In our factory problem, the variables represent the quantities of Product A and Product B that we want to produce.

In JuMP, we use the `@variable` macro to define variables. Here's the general 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

Now it's your turn! Create two continuous variables equal to or larger than 0 called `productA` and `productB` that represent 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

Now that we have defined our variables, constraints, and objective function, we're ready to 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))")
```

Let's break this down:

- `optimize!(model)` tells JuMP to solve our model
- `termination_status(model)` checks if we found an optimal solution
- `objective_value(model)` gives us the maximum profit we can achieve
- `value(productA)` and `value(productB)` tell us how many of each product we should produce

> **i** Note
>
> The values might be slightly off due to the nature of floating-point numbers in computers.

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 value(productA) ≈ 12 atol=1e-4 "Although you have a solution,
val_productA should be 12 not $val_productA"
@assert value(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.

Bibliography