

ÖGOR Summer-Workshop for PhD-candidates and Post-Docs

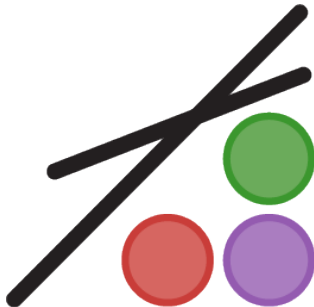
An introduction to Julia and JuMP for Operations Research

Prof. Dr. Xavier Gandibleux

Nantes Université – France
Département Informatique – Faculté des Sciences et Technique

Topic 5

Optimisation JuMP (part 1)



version 1.0.0 and later

Overview of JuMP

A modeling language for Mathematical Optimization (linear, mixed-integer, conic, semidefinite, nonlinear):

- ▶ **User friendliness**: syntax that mimics natural mathematical expressions.
- ▶ **Speed**: similar speeds to special-purpose modeling languages such as AMPL.
- ▶ **Solver independence**: JuMP uses *MathOptInterface (MOI)*, an abstraction layer designed to provide a unified interface to mathematical optimization solvers.

Currently supported solvers: Artelys Knitro, Baron, Bonmin, Cbc, Clp, Couenne, CPLEX, FICO Xpress, GLPK, Gurobi, SCIP, etc.

- ▶ **Ease of embedding**: JuMP itself is written purely in Julia. Solvers are the only binary dependencies.

Overview of JuMP

A modeling language for Mathematical Optimization (linear, mixed-integer, conic, semidefinite, nonlinear):

- ▶ **User friendliness**: syntax that mimics natural mathematical expressions.
- ▶ **Speed**: similar speeds to special-purpose modeling languages such as AMPL.
- ▶ **Solver independence**: JuMP uses *MathOptInterface (MOI)*, an abstraction layer designed to provide a unified interface to mathematical optimization solvers.

Currently supported solvers: Artelys Knitro, Baron, Bonmin, Cbc, Clp, Couenne, CPLEX, FICO Xpress, GLPK, Gurobi, SCIP, etc.

- ▶ **Ease of embedding**: JuMP itself is written purely in Julia. Solvers are the only binary dependencies.

Overview of JuMP

A modeling language for Mathematical Optimization (linear, mixed-integer, conic, semidefinite, nonlinear):

- ▶ **User friendliness**: syntax that mimics natural mathematical expressions.
- ▶ **Speed**: similar speeds to special-purpose modeling languages such as AMPL.
- ▶ **Solver independence**: JuMP uses *MathOptInterface (MOI)*, an abstraction layer designed to provide a unified interface to mathematical optimization solvers.

Currently supported solvers: Artelys Knitro, Baron, Bonmin, Cbc, Clp, Couenne, CPLEX, FICO Xpress, GLPK, Gurobi, SCIP, etc.

- ▶ **Ease of embedding**: JuMP itself is written purely in Julia. Solvers are the only binary dependencies.

Overview of JuMP

A modeling language for Mathematical Optimization (linear, mixed-integer, conic, semidefinite, nonlinear):

- ▶ **User friendliness**: syntax that mimics natural mathematical expressions.
- ▶ **Speed**: similar speeds to special-purpose modeling languages such as AMPL.
- ▶ **Solver independence**: JuMP uses *MathOptInterface (MOI)*, an abstraction layer designed to provide a unified interface to mathematical optimization solvers.

Currently supported solvers: Artelys Knitro, Baron, Bonmin, Cbc, Clp, Couenne, CPLEX, FICO Xpress, GLPK, Gurobi, SCIP, etc.

- ▶ **Ease of embedding**: JuMP itself is written purely in Julia.
Solvers are the only binary dependencies.

Overview of JuMP

A modeling language for Mathematical Optimization (linear, mixed-integer, conic, semidefinite, nonlinear):

- ▶ **User friendliness**: syntax that mimics natural mathematical expressions.
- ▶ **Speed**: similar speeds to special-purpose modeling languages such as AMPL.
- ▶ **Solver independence**: JuMP uses *MathOptInterface (MOI)*, an abstraction layer designed to provide a unified interface to mathematical optimization solvers.

Currently supported solvers: Artelys Knitro, Baron, Bonmin, Cbc, Clp, Couenne, CPLEX, FICO Xpress, GLPK, Gurobi, SCIP, etc.

- ▶ **Ease of embedding**: JuMP itself is written purely in Julia. Solvers are the only binary dependencies.

Getting started

Install:

```
using Pkg  
Pkg.add("JuMP")
```

```
Pkg.add("GLPK")
```

Setup:

```
using JuMP
```

```
using GLPK
```


Getting started

Install:

```
using Pkg  
Pkg.add("JuMP")
```

```
Pkg.add("GLPK")
```

Setup:

```
using JuMP
```

```
using GLPK
```

Getting started

Install:

```
using Pkg  
Pkg.add("JuMP")
```

```
Pkg.add("GLPK")
```

Setup:

```
using JuMP
```

```
using GLPK
```

Writing a model (1/5)

Example:

$$\left[\begin{array}{rclcl} \max z(x) = & x_1 & + & 3x_2 & (0) \\ s.t & x_1 & + & x_2 & \leq 14 \quad (1) \\ & -2x_1 & + & 3x_2 & \leq 12 \quad (2) \\ & 2x_1 & - & x_2 & \leq 12 \quad (3) \\ & x_1 & , & x_2 & \geq 0 \quad (4) \end{array} \right]$$

Writing a model (2/5)

Creating a Model:

```
modName = Model(solver)
```

```
julia> model = Model(GLPK.Optimizer)
```

Result:

[]

Writing a model (3/5)

Defining Variables:

```
@variable(modName, varName definition)
```

```
julia> @variable(model, x1 >= 0)
```

```
julia> @variable(model, x2 >= 0)
```

Result:

$$\left[\begin{array}{ccc} & & \\ & x_1 & , \quad x_2 \geq 0 \end{array} \right] \quad (4)$$

Writing a model (4/5)

Defining Objective:

```
@objective(modName, min/max, objectiveFunction)
```

```
julia> @objective(model, Max, x1 + 3x2)
```

Result:

$$\left[\begin{array}{rcll} \max z(x) = & x_1 & + & 3x_2 & (0) \\ & x_1 & , & x_2 & \geq 0 & (4) \end{array} \right]$$

Writing a model (5/5)

Defining Constraints:

```
@constraint(modName, cstName, cstDefinition)
```

```
julia> @constraint(model, cst1, x1 + x2 <= 14)
julia> @constraint(model, cst2, -2x1 + 3x2 <= 12)
julia> @constraint(model, cst3, 2x1 - x2 <= 12)
```

Result:

$$\left[\begin{array}{rcllcl} \max z(x) = & x_1 & + & 3x_2 & & (0) \\ s.t & x_1 & + & x_2 & \leq & 14 & (1) \\ & -2x_1 & + & 3x_2 & \leq & 12 & (2) \\ & 2x_1 & - & x_2 & \leq & 12 & (3) \\ & x_1 & , & x_2 & \geq & 0 & (4) \end{array} \right]$$

Details on

The model:

- ▶ print a summary of the problem:

```
julia> @show(model)
```

- ▶ print the formulation of the model:

```
julia> print(model)
```


Solve a model

```
optimize!(modelName)
```

```
julia> optimize!(model)
```

Querying the solution

```
julia> @show termination_status(model)
julia> @show primal_status(model)
julia> @show dual_status(model)
```

```
julia> @show objective_value(model)
julia> @show value(x1)
julia> @show value(x2)
```

```
julia> @show dual(cst1)
julia> @show dual(cst2)
julia> @show dual(cst3)
```

```
julia> solution_summary(model)
```

Querying the solution

```
julia> @show termination_status(model)
julia> @show primal_status(model)
julia> @show dual_status(model)
```

```
julia> @show objective_value(model)
julia> @show value(x1)
julia> @show value(x2)
```

```
julia> @show dual(cst1)
julia> @show dual(cst2)
julia> @show dual(cst3)
```

```
julia> solution_summary(model)
```

Querying the solution

```
julia> @show termination_status(model)
julia> @show primal_status(model)
julia> @show dual_status(model)
```

```
julia> @show objective_value(model)
julia> @show value(x1)
julia> @show value(x2)
```

```
julia> @show dual(cst1)
julia> @show dual(cst2)
julia> @show dual(cst3)
```

```
julia> solution_summary(model)
```

Querying the solution

```
julia> @show termination_status(model)
julia> @show primal_status(model)
julia> @show dual_status(model)
```

```
julia> @show objective_value(model)
julia> @show value(x1)
julia> @show value(x2)
```

```
julia> @show dual(cst1)
julia> @show dual(cst2)
julia> @show dual(cst3)
```

```
julia> solution_summary(model)
```

Details on

termination_status:

```
termination_status(modName)
```

Common return values

- ▶ **OPTIMAL:**

The algorithm found a globally optimal solution

- ▶ **INFEASIBLE:**

The algorithm concluded that no feasible solution exists.

- ▶ **TIME_LIMIT:**

The algorithm stopped after a user-specified computation time.

- ▶ **NUMERICAL_ERROR:**

The algorithm stopped because a numerical error.

- ▶ etc.

Details on

termination_status:

```
termination_status(modName)
```

Common return values

- ▶ **OPTIMAL:**
The algorithm found a globally optimal solution
- ▶ **INFEASIBLE:**
The algorithm concluded that no feasible solution exists.
- ▶ **TIME_LIMIT:**
The algorithm stopped after a user-specified computation time.
- ▶ **NUMERICAL_ERROR:**
The algorithm stopped because a numerical error.
- ▶ etc.

Recommended workflow

For solving a model and querying the solution:

```
julia> if termination_status(model) == MOI.OPTIMAL
    zOpt = objective_value(model)
    @printf(" z=%5.2f x1=%5.2f x2=%5.2f \n",
            zOpt,
            value(x1),
            value(x2))
    @printf(" u1=%5.2f u2=%5.2f u3=%5.2f \n",
            dual(cst1),
            dual(cst2),
            dual(cst3))
elseif termination_status(model) == DUAL_INFEASIBLE
    println("problem unbounded")
elseif termination_status(model) == MOI.INFEASIBLE
    println("problem infeasible")
end
```


Details on

Variables:

- ▶ by default, the variables are **continuous** and **unbounded**:

```
julia> @variable(model, x)           # x is free
```

- ▶ possible to setup lower and/or upper bounds on a variable:

```
julia> @variable(model, x ≥ lb))    # x is bounded
julia> @variable(model, x ≤ ub)
julia> @variable(model, lb ≤ x ≤ ub)
julia> @variable(model, x == 2)     # x is fixed
```

- ▶ possible to specify the type of a variable:

```
julia> @variable(model, x ≥ 0, Int) #  $x \in \mathbb{N}$ 
julia> @variable(model, x, Bin)     #  $x \in \{0, 1\}$ 
```

Details on

Variables:

- ▶ by default, the variables are **continuous** and **unbounded**:

```
julia> @variable(model, x)           # x is free
```

- ▶ possible to setup lower and/or upper bounds on a variable:

```
julia> @variable(model, x ≥ lb))    # x is bounded
julia> @variable(model, x ≤ ub)
julia> @variable(model, lb ≤ x ≤ ub)
julia> @variable(model, x == 2)     # x is fixed
```

- ▶ possible to specify the type of a variable:

```
julia> @variable(model, x ≥ 0, Int)    #  $x \in \mathbb{N}$ 
julia> @variable(model, x, Bin)        #  $x \in \{0, 1\}$ 
```

Details on

Variables:

- ▶ by default, the variables are **continuous** and **unbounded**:

```
julia> @variable(model, x)           # x is free
```

- ▶ possible to setup lower and/or upper bounds on a variable:

```
julia> @variable(model, x ≥ lb))    # x is bounded
julia> @variable(model, x ≤ ub)
julia> @variable(model, lb ≤ x ≤ ub)
julia> @variable(model, x == 2)     # x is fixed
```

- ▶ possible to specify the type of a variable:

```
julia> @variable(model, x ≥ 0, Int) #  $x \in \mathbb{N}$ 
julia> @variable(model, x, Bin)     #  $x \in \{0, 1\}$ 
```

Review and exercises

(notebook)

