Programming

Optimisation and Operations Research algorithms

with Julia

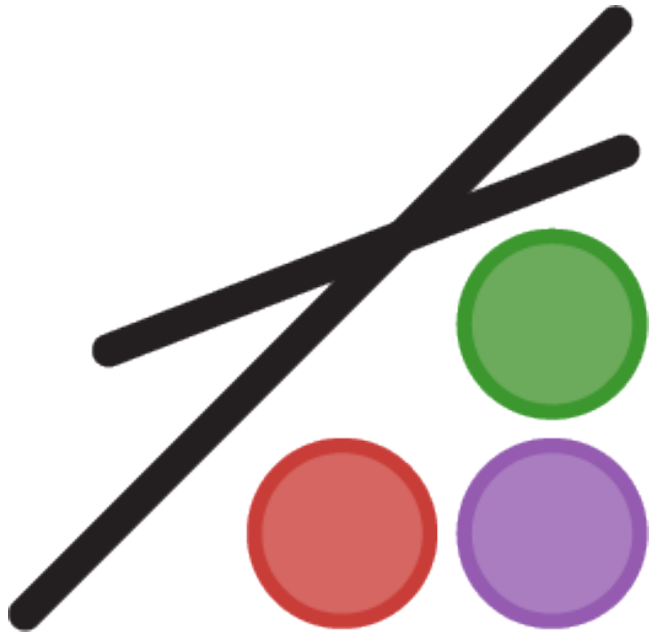
Prof. Dr. Xavier Gandibleux

Université de Nantes
Département Informatique – UFR Sciences et Techniques
France

Lesson 7 – May-June 2021

Optimisation

JuMP



version 0.21.8 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:

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

Writing a model (2/5)

Creating a Model:

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

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

Result:

$$\begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

Writing a model (3/5)

Defining Variables:

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

```
julia> @variable(model, x1 >= 0)
julia> @variable(model, x2 >= 0)
```

Result:

$$\begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ x_1 & , & x_2 & \geqslant & 0 \end{bmatrix} \quad (4)$$

Writing a model (4/5)

Defining Objective:

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

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

Result:

$$\begin{bmatrix} \max z(x) = & x_1 + 3x_2 & & (0) \\ & & & \\ & x_1 \quad , \quad x_2 \geqslant 0 & & (4) \end{bmatrix}$$

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:

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

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!(modName)
```

```
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 continous 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 ∈ ℕ
julia> @variable(model, x, Bin)             # x ∈ {0,1}
```

Details on

Variables:

- by default, the variables are continous 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 ∈ ℕ
julia> @variable(model, x, Bin)          # x ∈ {0,1}
```

Details on

Variables:

- by default, the variables are continous 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 ∈ ℕ
julia> @variable(model, x, Bin)          # x ∈ {0,1}
```

Writing an implicit model (1/5)

Example (cont'd):

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

$$\downarrow$$

$$c = (1, 3),\ d = (14, 12, 12),\ T = \begin{pmatrix} 1 & 1 \\ -2 & 3 \\ 2 & -1 \end{pmatrix}$$

$$\left[\begin{array}{lll} \max z(x) = & \sum_{j=1}^{2} c_j x_j & (0) \\ s.t & \sum_{j=1}^{2} t_{ij} x_j \leqslant d_i \quad i = 1, 3 & (1-3) \\ & x_j \geqslant 0 \qquad\qquad j = 1, 2 & (4) \end{array}\right]$$

Writing an implicit model (2/5)

Example (cont'd):

$$c = (1, 3),\ d = (14, 12, 12),\ T = \begin{pmatrix} 1 & 1 \\ -2 & 3 \\ 2 & -1 \end{pmatrix}$$

```
c = [1, 3]
d = [14, 12, 12]
T = [1 1 ; -2 3; 2 -1]
n,m = size(T)
```

Writing an implicit model (2/5)

Example (cont'd):

$$\left[\begin{array}{lllll} \max z(x) = & \sum_{j=1}^{2} c_j x_j & & & (0) \\ s.t & \sum_{j=1}^{2} t_{ij} x_j \leqslant d_i & i = 1,3 & & (1-3) \\ & x_j \geqslant 0 & j = 1,2 & & (4) \end{array}\right]$$

```
julia> md = Model(GLPK.Optimizer)
julia> @variable(md, x[1:m]⩾0)
julia> @objective(md, Max,sum(c[j]*x[j] for j=1:m))
julia> @constraint(md, cst[i=1:n],sum(T[i,j]*x[j]
                                      for j=1:m)⩽d[i])
```

Structured variables in JuMP

Scalar variables:

```
julia> @variable(md, x≥0)
```

Structured variables:

- Arrays (one-based integer ranges)

```
julia> @variable(md, x[1:4]≥0)
julia> @variable(md, x[1:2, 1:2]≥0, Int)
```

- DenseAxisArrays (indices are not one-based integer ranges)

```
julia> @variable(md, x[-4:4]≥0)
julia> @variable(md, y[[:tram,:train]], Bin)
```

- SparseAxisArrays (indices do not form a rectangular set)

```
julia> @variable(model, x[i=1:2, j=i:2])
```

Structured variables in JuMP

Scalar variables:

```
julia> @variable(md, x≥0)
```

Structured variables:

- Arrays (one-based integer ranges)

```
julia> @variable(md, x[1:4]≥0)
julia> @variable(md, x[1:2, 1:2]≥0, Int)
```

- DenseAxisArrays (indices are not one-based integer ranges)

```
julia> @variable(md, x[-4:4]≥0)
julia> @variable(md, y[[:tram,:train]], Bin)
```

- SparseAxisArrays (indices do not form a rectangular set)

```
julia> @variable(model, x[i=1:2, j=i:2])
```

More on variables

Providing a primal starting solution (a MIP-start; a warmstart)

```
julia> @variable(model, x, start = 44)
```

```
julia> set_start_value(x, 44)
```

Deleting variables:

```
julia> delete(model, x)
```

More on variables

Providing a primal starting solution (a MIP-start; a warmstart)

```
julia> @variable(model, x, start = 44)
```

```
julia> set_start_value(x, 44)
```

Deleting variables:

```
julia> delete(model, x)
```

More on the objective

Modify an objective:

call `@objective` with the new objective function:

```
julia> @objective(model, Min, 2x)
```

Modify an objective coefficient:

use `set_objective_coefficient`:

```
set_objective_coefficient(model, variable, coef)
```

```
julia> set_objective_coefficient(model, x2, 5)
```

More on the objective

Modify an objective:

call `@objective` with the new objective function:

```
julia> @objective(model, Min, 2x)
```

Modify an objective coefficient:

use `set_objective_coefficient`:

```
set_objective_coefficient(model, variable, coef)
```

```
julia> set_objective_coefficient(model, x2, 5)
```

More on constraints

Modify the coefficient of a variable for a given constraint:

```
set_normalized_coefficient(cstID, var, coef)
```

```
julia> set_normalized_coefficient(model, x2, 5)
```

Set the right-hand side term of a constraint to a value:

```
set_normalized_rhs(constraintID, value)
```

```
julia> set_normalized_rhs(cst2, 10)
```

Delete a constraint from a model:

```
delete(model, constraintID)
```

```
julia> delete(model, cst2)
```

More on constraints

Modify the coefficient of a variable for a given constraint:

```
set_normalized_coefficient(cstID, var, coef)
```

```
julia> set_normalized_coefficient(model, x2, 5)
```

Set the right-hand side term of a constraint to a value:

```
set_normalized_rhs(constraintID, value)
```

```
julia> set_normalized_rhs(cst2, 10)
```

Delete a constraint from a model:

```
delete(model, constraintID)
```

```
julia> delete(model, cst2)
```

More on constraints

Modify the coefficient of a variable for a given constraint:

```
set_normalized_coefficient(cstID, var, coef)
```

```
julia> set_normalized_coefficient(model, x2, 5)
```

Set the right-hand side term of a constraint to a value:

```
set_normalized_rhs(constraintID, value)
```

```
julia> set_normalized_rhs(cst2, 10)
```

Delete a constraint from a model:

```
delete(model, constraintID)
```

```
julia> delete(model, cst2)
```

More on solutions

Primal solution values for a scalar variable:

```
value(variable)
```

```
julia> value(x1)
```

Primal solution values for a structured variable:

```
value(variable[index])
```

```
julia> value(x[3])
```

```
value.(variable)
```

```
julia> value.(x)
```

More on solutions

Primal solution values for a scalar variable:

```
value(variable)
```

```
julia> value(x1)
```

Primal solution values for a structured variable:

```
value(variable[index])
```

```
julia> value(x[3])
```

```
value.(variable)
```

```
julia> value.(x)
```

Optimisation

vOptSolver

Overview of vOptSolver

An ecosystem for modeling and solving multiobjective linear optimization problems (MOCO, MOIP, MOMIP, MOLP):

- it deals with structured and non-structured optimization problems with at least two objectives
- it integrates several specific and generic exact algorithms for computing efficient solutions
- Natural and intuitive use for mathematicians, informaticians, engineers
- Efficient, flexible, evolutive solver
- Aims to be easy to formulate a problem, to provide data, to solve a problem, to collect the outputs, to analyze the solutions
- Free, open source (MIT licence), multi-platform, reusing existing specifications
- Using usual free (GLPK, Clp/Cbc) and commercial (GUROBI, CPLEX) MILP solvers

Overview of vOptSolver

An ecosystem for modeling and solving multiobjective linear optimization problems (MOCO, MOIP, MOMIP, MOLP):

- it deals with structured and non-structured optimization problems with at least two objectives
- it integrates several specific and generic exact algorithms for computing efficient solutions
- Natural and intuitive use for mathematicians, informaticians, engineers
- Efficient, flexible, evolutive solver
- Aims to be easy to formulate a problem, to provide data, to solve a problem, to collect the outputs, to analyze the solutions
- Free, open source (MIT licence), multi-platform, reusing existing specifications
- Using usual free (GLPK, Clp/Cbc) and commercial (GUROBI, CPLEX) MILP solvers

Getting started with vOptGeneric

Install:

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

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

Setup:

```
using vOptGeneric
```

```
using GLPK
```

Getting started with vOptGeneric

Install:

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

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

Setup:

```
using vOptGeneric
```

```
using GLPK
```

Example with vOptGeneric

For the bi-objective unidimensional 01 knapsack problem,

$$\max \left\{ (p^1 x,\ p^2 x) \mid wx \leqslant c,\ x \in \{0,1\}^n \right\}$$

with[1]

$$n = 5$$

$$p^1 = (10,\ 3,\ 6,\ 8,\ 2)$$

$$p^2 = (12,\ 9,\ 11,\ 5,\ 6)$$

$$w = (4,\ 5,\ 2,\ 5,\ 6)$$

$$c = 17$$

compute Y_N, the set of non-dominated points using the ϵ-constraint method.

[1] exercise 10.2, page 290 of *Multicriteria Optimization* (2nd edt), M. Ehrgott, Springer 2005

Setup the data

```
julia> p1 = [10,3,6,8,2]          # coef vct of the obj 1
julia> p2 = [12,9,11,5,6]         # coef vct of the obj 2
julia> w = [4,5,2,5,6]              # coef vct of weights
julia> c = 17                          # nominal capacity
julia> n = length(p1)                   # number of items
```

Setup the model

```
julia> kp = vModel( GLPK.Optimizer )
julia> @variable(kp, x[1:n],Bin)
julia> @addobjective(kp,Max,sum(p1[j]*x[j] for j=1:n))
julia> @addobjective(kp,Max,sum(p2[j]*x[j] for j=1:n))
julia> @constraint(kp,sum(w[j]*x[j] for j=1:n) ⩽ c)
```

Solve and display results

Invoking the solver (dichotomic method):

```
julia> vSolve(kp,method=:epsilon,step=0.5,verbose=false)
```

Querying the results:

```
julia> Y_N = getY_N(kp)
```

Displaying the results (X_SE and Y_SN):

```
julia> for i = 1:length(Y_N)
julia>     X = value.(x, i)
julia>     print("X = ", findall(elt -> elt ≈ 1, X))
julia>     println(" | Z = ",Y_N[i])
julia> end
```

```
julia> printX_E(kp)
```

Plot results

```
julia> using PyPlot
julia> z1, z2 = map(x -> x[1],Y_N), map(x -> x[2],Y_N)
julia> PyPlot.title("Knapsack")
julia> PyPlot.xlabel("$z_1$ to maximize")
julia> PyPlot.ylabel("$z_2$ to maximize")
julia> grid()
julia> plot(z1,z2,"bx",markersize="8",label="$Y_N$")
julia> legend(loc=1,fontsize="small")
julia> show()
```

Review and exercises

(notebook)