Workshop on Optimization Techniques for Data Science in Python and Julia

11. Summary: comparison of Pyomo and JuMP

Bogumił Kamiński

Solving sudoku (Hart et al., chap. 14.6.2)

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			

Solving optimization problems

1. Mathematical formulation
2. Problem type identification
3. Software implementation
4. Solution

Problem type identification

- Decision variables: boolean
- Objective: linear in variables
- Constraints: linear in variables

Type of problem:
mixed integer programming

Software implementation (sudoku.jl)

```julia
using JuMP
using GLPKMathProgInterface

function sudokusolver(board)
    m = Model(solver=GLPKSolverMIP())
    @variable(m, x[1:9, 1:9, 1:9], Bin)
    for i in 1:9, j in 1:9
        @constraint(m, sum(x[i, j, :]) == 1)
        @constraint(m, sum(x[i, :, j]) == 1)
        @constraint(m, sum(x[:, i, j]) == 1)
        for (i, j, k) in board
            @constraint(m, x[i, j, k] == 1)
        end
    end
    for i in 0:2, j in 0:2, k in 1:9
        @constraint(m, sum(x[3i .+ (1:3), 3j .+ (1:3), k]) == 1)
    end
```

```julia
 solution_count = 0
    sol = zeros(Int, 9, 9)
    while true
        res = solve(m)
        if res == :Optimal
            solution_count += 1
            print("Solution #$solution_count")
            xv = round.(Int, getvalue(x))
            for idx in findall(==(1), xv)
                sol[idx[1], idx[2]] = idx[3]
            end
            display(sol)
            @constraint(m, sum(xv .* x) <= 80)
        else
            print("All board solutions have been found")
            return
        end
    end
end
```

Solution (sudoku.jl)

```
board = [(1,1,5),(1,2,3),(1,5,7),(2,1,6),(2,4,1),(2,5,9),(2,6,5),
         (3,2,9),(3,3,8),(3,8,6),(4,1,8),(4,5,6),(4,9,3),(5,1,4),
         (5,4,8),(5,6,3),(5,9,1),(6,1,7),(6,5,2),(6,9,6),(7,2,6),
         (7,7,2),(7,8,8),(8,4,4),(8,5,1),(8,6,9)]

sudokusolver(board)
```

Solution (sudoku.jl)

```
julia> sudokusolver(board)
Solution #19×9 Array{Int64,2}:
 5  3  4  6  7  8  1  9  2
 6  7  2  1  9  5  3  4  8
 1  9  8  3  4  2  5  6  7
 8  5  9  7  6  1  4  2  3
 4  2  6  8  5  3  9  7  1
 7  1  3  9  2  4  8  5  6
 9  6  1  5  3  7  2  8  4
 2  8  7  4  1  9  6  3  5
 3  4  5  2  8  6  7  1  9
Solution #29×9 Array{Int64,2}:
 5  3  4  6  7  8  9  1  2
 6  7  2  1  9  5  3  4  8
 1  9  8  3  4  2  5  6  7
 8  5  9  7  6  1  4  2  3
 4  2  6  8  5  3  7  9  1
 7  1  3  9  2  4  8  5  6
 9  6  1  5  3  7  2  8  4
 2  8  7  4  1  9  6  3  5
 3  4  5  2  8  6  1  7  9
Solution #39×9 Array{Int64,2}:
 5  3  4  6  7  8  1  9  2
 6  7  2  1  9  5  3  4  8
 1  9  8  3  4  2  7  6  5
 8  5  9  7  6  1  4  2  3
 4  2  6  8  5  3  9  7  1
 7  1  3  9  2  4  8  5  6
 9  6  1  5  3  7  2  8  4
 2  8  5  4  1  9  6  3  7
 3  4  7  2  8  6  5  1  9
┌ Warning: Not solved to optimality, status: Infeasible
└ @ JuMP ~\.julia\packages\JuMP\PbnIJ\src\solvers.jl:212
┌ Warning: Infeasibility ray (Farkas proof) not available
└ @ JuMP ~\.julia\packages\JuMP\PbnIJ\src\solvers.jl:223
All board solutions have been found
```

Code comparison (a personal perspective)

- Both Pyomo and JuMP follow the same pattern of separation of model specification from the solver
- In general you can solve the same classes of problems in both of them

- Pyomo
  - You have to name variables, objectives and constraints in model object
  - Somewhat cumbersome interface function-definition
- JuMP
  - You can optionally assign names to objects
  - More convenient definition flow due to metaprogramming facilities in Julia
  - Shorter code due to Julia syntax being more expressive than Python for mathematical formulas

Performance (model generation) https://mlubin.github.io/pdf/jump-sirev.pdf

Problem	Pyomo	JuMP
quadratic objective + linear constraints 50	55 s.	8 s.
quadratic objective + linear constraints 1000	232 s.	11 s.
quadratic objective + linear constraints 1500	530 s.	15 s.
quadratic objective + linear constraints 2000	> 600 s.	22 s.
Linear objective + conic quadratic constraints 25	14 s.	7 s.
Linear objective + conic quadratic constraints 50	114 s.	9 s.
Linear objective + conic quadratic constraints 75	391 s.	13 s.
Linear objective + conic quadratic constraints 100	>600 s.	24 s.
derivative based nonlinear optimization 5	3 s.	18 s.
derivative based nonlinear optimization 50	26 s.	21 s.
derivative based nonlinear optimization 500	261 s.	66 s.

Performance (model generation) a basic example

JuMP

```julia
using JuMP
using GLPKMathProgInterface

function genproblem()
    m = Model(solver=GLPKSolverMIP())
    @variable(m, x[1:4_000_000])
    @variable(m, y[1:2000], Bin)
    @objective(m, Max, sum(x)+sum(y))
    for i in 1:4_000_000
        @constraint(m, x[i] <= i)
    end
    for i in 1:2000, j in 1:2000
        @constraint(m, y[i]+y[j] <= 1)
    end
    m
end

@time genproblem();
```

Pyomo

```python
from pyomo.environ import *
from timeit import default_timer as timer

def genproblem():
    m = ConcreteModel()
    m.XRANGE = RangeSet(1,4000000)
    m.YRANGE = RangeSet(1,2000)
    m.x = Var(m.XRANGE)
    m.y = Var(m.YRANGE, within=Binary)
    m.obj = Objective(expr=sum(m.x[i] for i in m.XRANGE) +
                           sum(m.y[i] for i in m.YRANGE))
    def xconstraint(m, i):
        return m.x[i] <= i
    m.con1 = Constraint(m.XRANGE, rule=xconstraint)
    def yconstraint(m, i, j):
        return m.y[i]+m.y[j] <= 1
    m.con2 = Constraint(m.YRANGE, m.YRANGE, rule=yconstraint)
    return m

start = timer()
genproblem()
end = timer()
print(end - start)
```

Performance (model generation) a basic example

JuMP

```julia
using JuMP
using GLPKMathProgInterface

function genproblem()
    m = Model(solver=GLPKSolverMIP())
    @variable(m, x[1:4_000_000])
    @variable(m, y[1:2000], Bin)
    @objective(m, Max, sum(x)+sum(y))
    for i in 1:4_000_000
        @constraint(m, x[i] <= i)
    end
    for i in 1:2000, j in 1:2000
        @constraint(m, y[i]+y[j] <= 1)
    end
    m
end

@time genproblem();
```

7.5 s.

Pyomo

```python
from pyomo.environ import *
from timeit import default_timer as timer

def genproblem():
    m = ConcreteModel()
    m.XRANGE = RangeSet(1,4000000)
    m.YRANGE = RangeSet(1,2000)
    m.x = Var(m.XRANGE)
    m.y = Var(m.YRANGE, within=Binary)
    m.obj = Objective(expr=sum(m.x[i] for i in m.XRANGE) +
                           sum(m.y[i] for i in m.YRANGE))
    def xconstraint(m, i):
        return m.x[i] <= i
    m.con1 = Constraint(m.XRANGE, rule=xconstraint)
    def yconstraint(m, i, j):
        return m.y[i]+m.y[j] <= 1
    m.con2 = Constraint(m.YRANGE, m.YRANGE, rule=yconstraint)
    return m

start = timer()
genproblem()
end = timer()
print(end - start)
```

95.5 s.

Concluding remarks

- Pyomo and JuMP are very similar in design
- They differ in details
- Use whichever suits your general development workflow better

Thank you!