UW EE 553 Power System Economics, Spring 2017
Using Julia for Optimization

Yishen Wang

April 2017

1 Introduction

In this course we will be using *Julia* to solve optimization problems. This is a relatively new programming language for scientific computing purpose, and it is fully open source [1]. To have a better exposure for the optimization modeling, we will mainly code with *JuMP* package embedded in *Julia*, a domain-specific modeling language for mathematical optimization [2, 3].

This guide provides two simple examples on how to model Linear Programming (LP) and Mixed-integer Linear Programming (MILP) problems in *Julia* and *JuMP*. For a more detailed document, please refer to JuMP Official Documents.

Keep in mind that a strong background in programming (with *Julia* or otherwise) is not expected. In this course, we only touch a tiny subset of *Julia*'s (powerful though) features.

2 Using *Julia* with JuliaBox

You don't have to install *Julia* locally! We recommend using *Julia* on the free online platform JuliaBox. It is very easy to use, and it should be sufficient for most cases during the class.

To start a new project, click "New" to open a "Julia 0.5.1" Notebook as shown in Figure 1. Figure 2 presents the interactive markdown window. In the code cell, we can write codes and click "run" to obtain the results. The provided code cell (code block) feature helps to better organize and test long codes in a more structural way. Figure 3 provides an example for an easy LP problem.

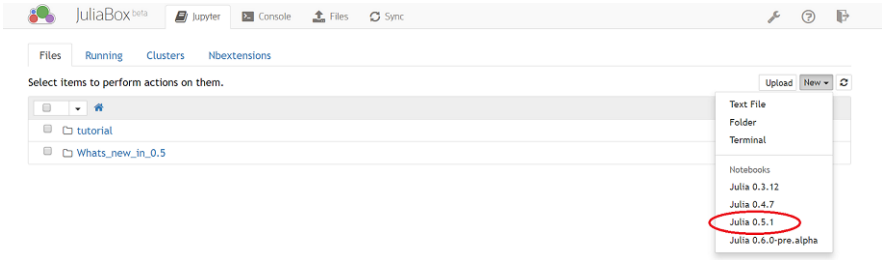

Figure 1: Initiate a new code in JuliaBox

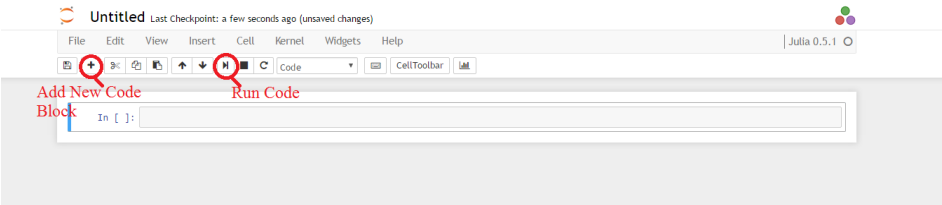

Figure 2: New interactive script in JuliaBox

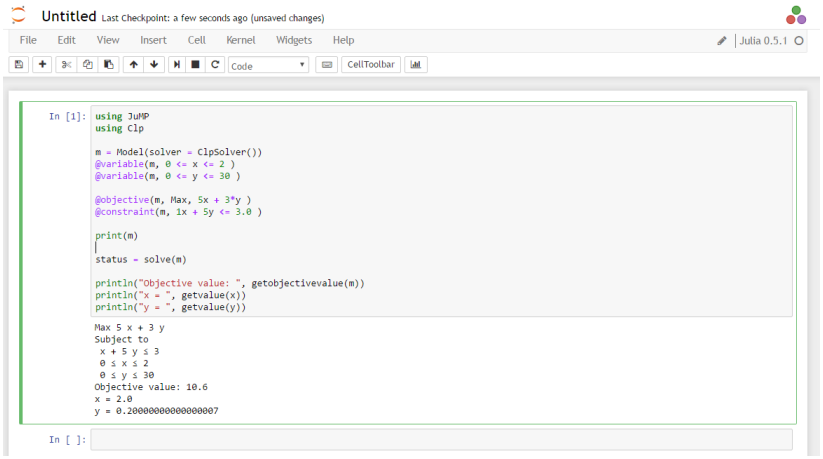

Figure 3: Example with JuliaBox

3 Using *Julia* with JuliaPro

As for a local version of *Julia*, JuliaPro is a good alternative to include *Julia* compiler, debugger, profiler, *Julia* integrated development environment and 100+ curated packages. It is free for personal use as well after registration.

This software provides multiple ways to use Julia. Juno IDE probably is a convenient one to use if you are familiar with MATLAB-style IDE.

We provide the same example here in Figure 4. Noticed that initially some packages are not pre-installed, we can add these packages with command "Pkg.add()" as shown in the first line for package *Clp*.

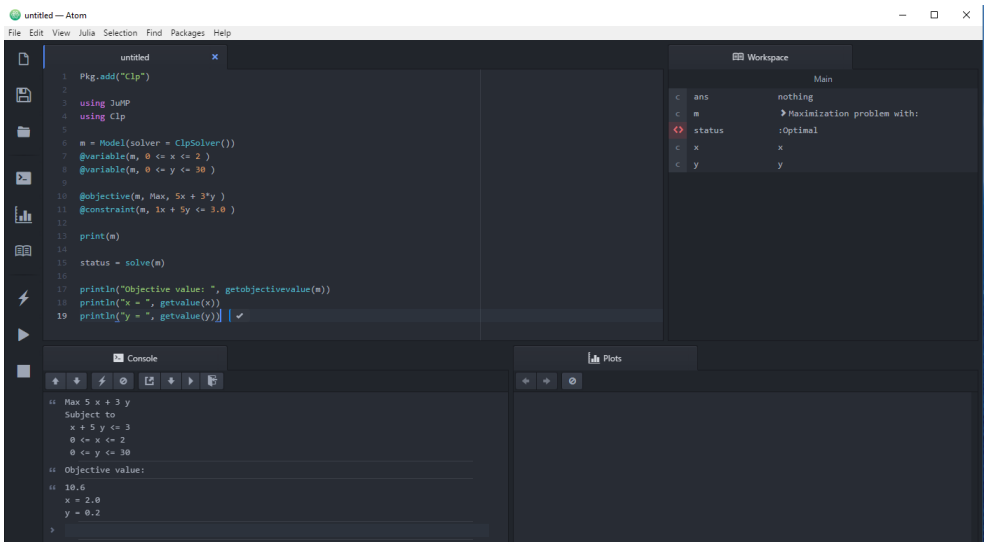

Figure 4: Example with JuliaPro

4 Getting Gurobi

Juliabox has already installed a number of open source solvers, like *Clp*, *GLPK*, *Cbc*, etc. These solvers generally work fine for small-scale problems. However, their performances are not guaranteed for realistic sized problems. In this class, we also have the chance to experience a commercial solver *Gurobi*, which is also academically free.

First, go to *Gurobi* website and register an account for Academic use. Then download the *Gurobi Optimizer* by choosing proper distribution.

Although *Gurobi* is a commercial software, it provides free academic licenses. Simply request this free license at the license center. Then, we can activate such license with *grbgetkey* command as shown in Figure 5. In addition, make sure that system environment variables for *GRB_LICENSE_FILE* and *GUROBI_HOME* are properly set.

In *Julia*, add corresponding *Gurobi* package with command:

Pkg.add("Gurobi")

Then, we can call this package by command:

using Gurobi


License Detail

License ID [redacted]

Information and installation instructions

License ID	[redacted]
Date Issued	2017-03-29
Purpose	Trial
License Type	Free Academic
Key Type	ACADEMIC
Version	7
Distributed Limit	0
Expiration Date	2018-03-29
Host Name	[redacted]
Host ID	[redacted]
User Name	[redacted]

To install this license on a computer where Gurobi Optimizer is installed, copy and paste the following command to the Start/Run menu (Windows only) or a command/terminal prompt (any system):

grbgetkey [redacted]

The grbgetkey command requires an active internet connection. If you get no response or an error message such as "Unable to contact key server", please click here for additional instructions.

Figure 5: Gurobo License

5 Optimization with *JuMP*

5.1 Getting *JuMP*

The *JuMP* has been installed at JuliaBox and JuliaPro, so you don't have to install it. If you uses other *Julia* distributions, please use Pkg.add function to install this package.

Pkg.add("JuMP")

Before defining optimization models, we should call *JuMP* package by

using JuMP

5.2 Getting Solver

For small problems, we suggest using *Clp* for linear programming and *Cbc* for mixed-integer linear programming as you can easily run it with Juliabox.

using Clp
using Cbc

For more challenging problems, we suggest using *Gurobi* with JuliaPro. It is suitable for both LP and MILP.

```
using Gurobi
```

5.3 Creating a Model

The optimization instances we want to solve are defined as Models in the *JuMP*. All variables and constraints are associated with a Model object. It is created by calling the constructor:

```
m = Model()
```

Usually, we also want to provide a solver object here by using the solver= keyword argument. Taking solver *Clp* as an example:

```
m = Model(solve=ClpSolver())
```

Similar commands work for solver *Cbc* and *Gurobi*:

```
m = Model(solve=CbcSolver())
m = Model(solve=GurobiSolver())
```

5.4 Defining Variables

Variables are also *Julia* objects, and are defined using the @variable macro. The first argument will always be the Model to associate this variable with. The second argument is an expression that declares the variable name and optionally allows specification of lower and upper bounds. Here are examples to define variable x with different bound conditions:

```
@variable(m, x)                    #No bounds
@variable(m, x >= lb)              #Lower bound only
@variable(m, x <= ub)              #Upper bound only
@variable(m, lb <= x <= ub)        #Lower and upper bounds
```

Integer and binary restrictions can optionally be specified with a third argument, *Int* or *Bin*.

```
@variable(m, x, Bin)   #Binary Variable
```

To create arrays of variables we append brackets to the variable name.

```
@variable(m, x[1:M, 1:N] >= 0)
```

5.5 Defining Constraints and Objective Function

JuMP allows users to use a natural notation to describe linear expressions. To add constraints, use the @constraint() and @objective() macros.

@constraint(m, x[i] - s[i] <= 0) **#Other options: == and >=**
@constraint(m, sum(x[i] for i=1:N) == 1)
@objective(m, Max, 5x + 22y + (x+y)/2) **#or Min**

5.6 Solving and Reading Solutions

After writing the optimization models in *Julia*, we can use the following command to solve this problem and obtain the optimal solution:

solve(m)

Get objective function value from model m:

getobjectivevalue(m)

Get optimal variable value of variable x:

getvalue(x)

6 Example 1: Linear Programming

For the following LP problem:

$$\begin{aligned} \min \quad & 3x + 10y \\ \text{Subject to} \quad & \\ & x + 5y \leq 8 \\ & 2x + 3y = 10 \\ & 0 \leq x \leq 4 \\ & 0 \leq y \leq 5 \end{aligned}$$

For JuliaBox users, see Figure 6. Solver *Clp* is used in this example for LP. Code can be found in the **EE553_julia_example.ipynb**.

For JuliaPro users, see Figure 7. Solver *Gurobi* is used in this example for LP. Code can be found in the **Example1_lp.jl**.

In [1]:

```
# Example 1: LP
#------------------------------------------------

# calling package JuMP and LP solver Clp
using JuMP
using Clp

# define Model with solver option
m1 = Model(solver = ClpSolver())

# define variables with given upper and lower bounds
@variable(m1, 0 <= x <= 4 )
@variable(m1, 0 <= y <= 5 )


# define objective function to minimize
@objective(m1, Min, 3x + 10*y )

# define constraints
@constraint(m1, 1x + 5y <= 8.0 )
@constraint(m1, 2x + 3y == 10.0 )

# print optimization model
print(m1)

# save the status
status = solve(m1)

# output results
println("Objective value: ", getobjectivevalue(m1))
println("x = ", getvalue(x))
println("y = ", getvalue(y))
```

```
Min 3 x + 10 y
Subject to
 x + 5 y ≤ 8
 2 x + 3 y = 10
 0 ≤ x ≤ 4
 0 ≤ y ≤ 5
Objective value: 18.666666666666664
x = 4.0
y = 0.6666666666666666
```

Figure 6: Example 1 with JuliaBox

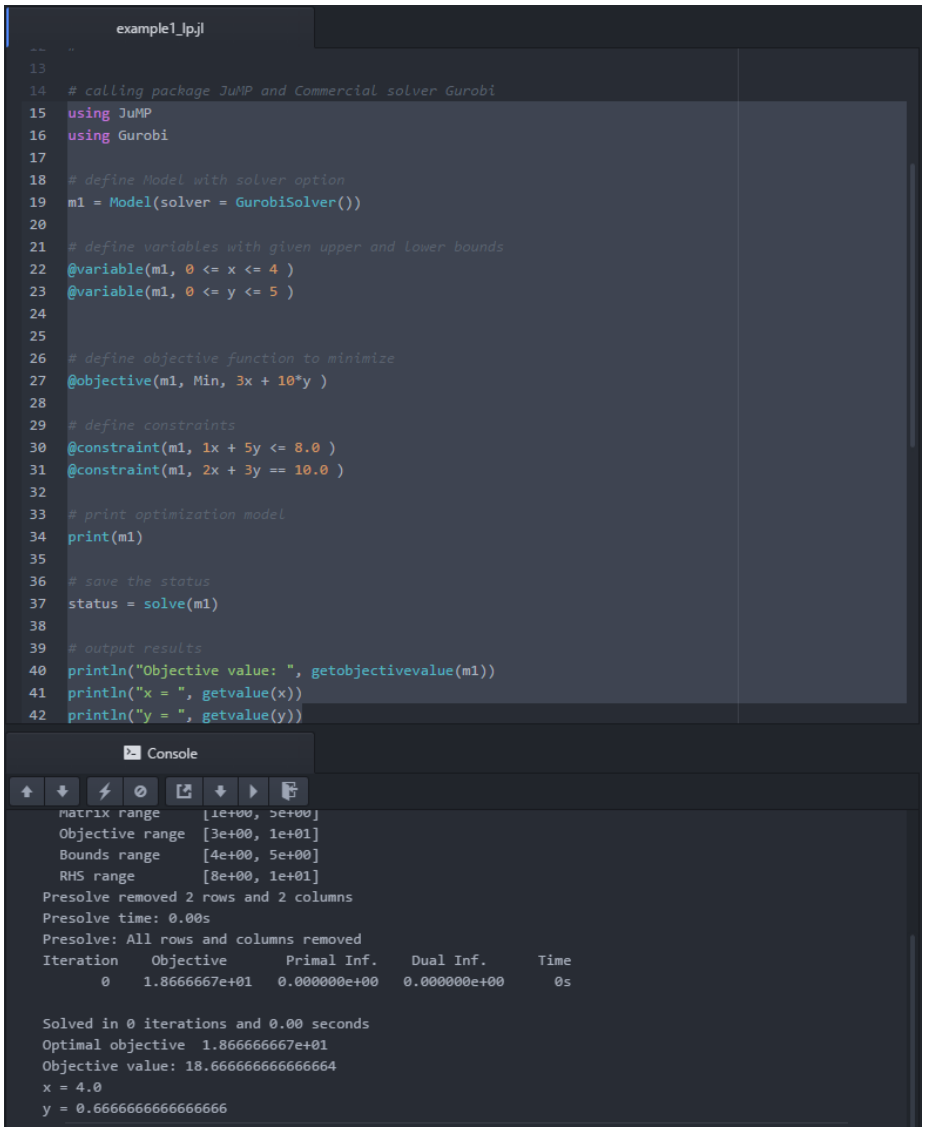

Figure 7: Example 1 with JuliaPro

7 Example 2: Mixed-integer Linear Programming

For MILP problem, we use a 0-1 knapsack problem for an example. Consider a person who needs to pack for a hike. Assume there exists a set of items to

choose, and the knapsack has a maximum allowable weight. Each item has its own weight and utility. The person needs to choose a subset of these items to maximize his total utilities while not exceeding the carrying capacity of the hiker.

Parameters include:

i	Index for item i
v_i	Utility for item i
w_i	Weight for item i
w_{max}	Maximum allowable weight

The problem is formulated as:

$$\begin{aligned} \max \quad & \sum_i v_i x_i \\ \text{Subject to} \quad & \\ & \sum_i w_i x_i \leq w_{max} \\ & x_i \in \{0,1\}, \quad \forall i \end{aligned}$$

For JuliaBox users, see Figure 8. Solver *Cbc* is used in this example for MILP. Code can be found in the **EE553_julia_example.ipynb**.

For JuliaPro users, see Figure 9. Solver *Gurobi* is used in this example for MILP. Code can be found in the **Example2_milp.jl**.

In [2]:

```
# Example 2: MILP
#------------------------------------------------

# calling package JuMP and MILP solver Clp
using JuMP
using Cbc

# define knapsack item value in vector
v = [10, 40, 30, 50]

# define knapsack item weight in vector
w = [5, 4, 6, 3]

# define maximum allowable weight in a knapsack
w_max = 10

# define Model with solver option
m2 = Model(solver = CbcSolver())

# define binary variables whether to put item into knapsack, 1-put, 0-not put
@variable(m2, x[1:4], Bin)

# define objective function to maximize the value in the knapsack
@objective(m2, Max, sum(v[i]*x[i] for i = 1:4) )

# define knapsack weight should not exceed maximum weight
@constraint(m2, sum(w[i]*x[i] for i = 1:4) <= w_max )

# print optimization model
print(m2)
status = solve(m2)

# output results
println("Objective value: ", getobjectivevalue(m2))
println("x = ", getvalue(x))
```

```
Max 10 x[1] + 40 x[2] + 30 x[3] + 50 x[4]
Subject to
 5 x[1] + 4 x[2] + 6 x[3] + 3 x[4] ≤ 10
 x[i] ∈ {0,1} ∀ i ∈ {1,2,3,4}
Objective value: 90.0
x = [0.0,1.0,0.0,1.0]
```

Figure 8: Example 2 with JuliaBox

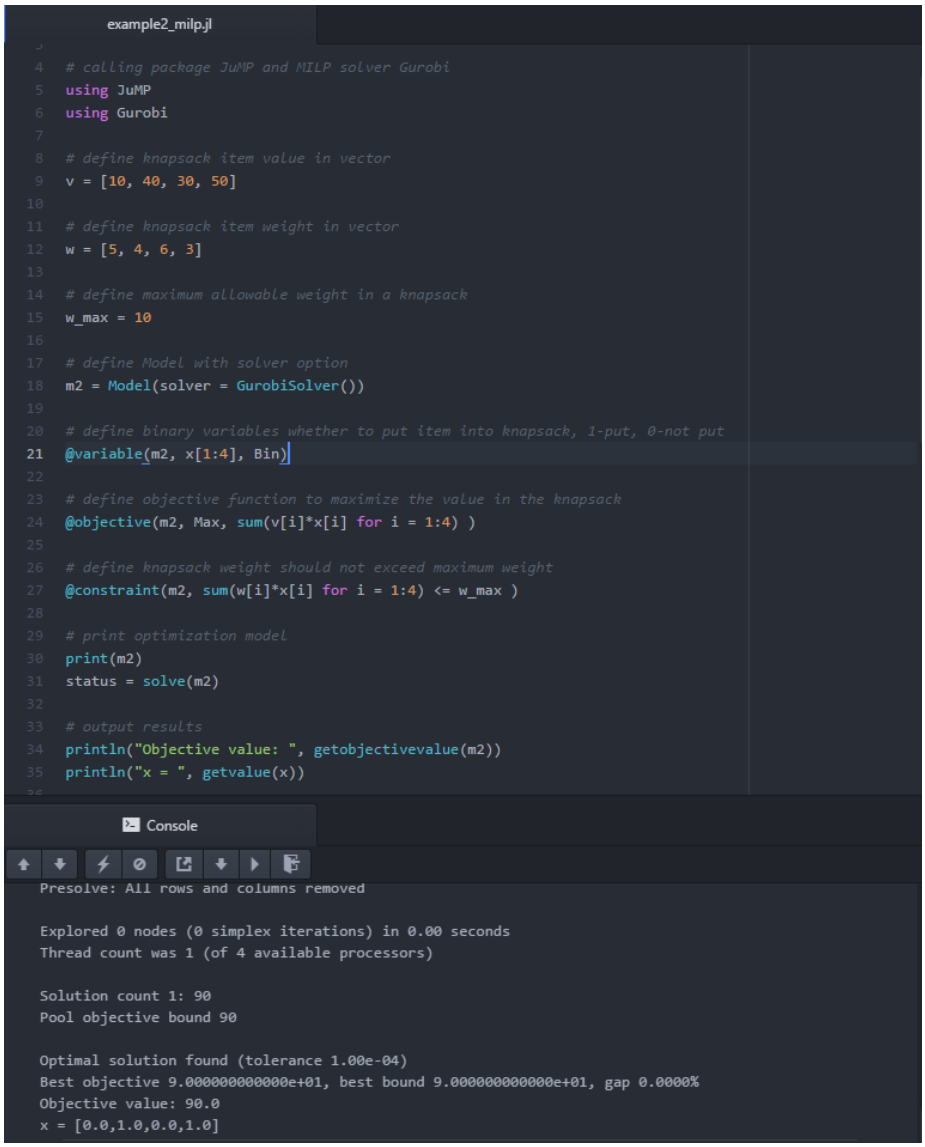

Figure 9: Example 2 with JuliaPro

8 Additional Resources

- *Julia* Official Website: http://julialang.org/
- *JuMP* Documentation: https://jump.readthedocs.io/en/latest/index.html

- JuliaOpt Examples (slightly outdated *JuMP* syntax):
  http://www.juliaopt.org/notebooks/index.html
- Stanford EE 103: http://stanford.edu/class/ee103/julia.html

References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[2] Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *arXiv:1508.01982 [math.OC]*, 2015.

[3] Miles Lubin and Iain Dunning. Computing in operations research using julia. *INFORMS Journal on Computing*, 27(2):238–248, 2015.