Lecture 2 – Modeling in Julia/JuMP

Module 2 – Julia

CS/ISyE/ECE 524

Using Julia/JuMP to solve a model

- Julia is a **programming language**
- JuMP (Julia Mathematical Programming) is a **modeling language** written in Julia
- To use JuMP to solve an instance of a model, you must specify a **solver** (Clp, ECOS, Gurobi, …)

```
using JuMP, Clp, ECOS, Gurobi
m = Model(Clp.Optimizer)
m = Model(Gurobi.Optimizer)

m = Model()
set_optimizer(m, Clp.Optimizer)
optimize!(m)
set_optimizer(m, Gurobi.Optimizer)
optimize!(m)
```

Set a solver when creating model

Or create model and set a solver later

Getting started with Julia/JuMP

- To use any package (including the Package package) you have to tell Julia to "*use*" it:

```
using Pkg, JuMP, Clp, …
```

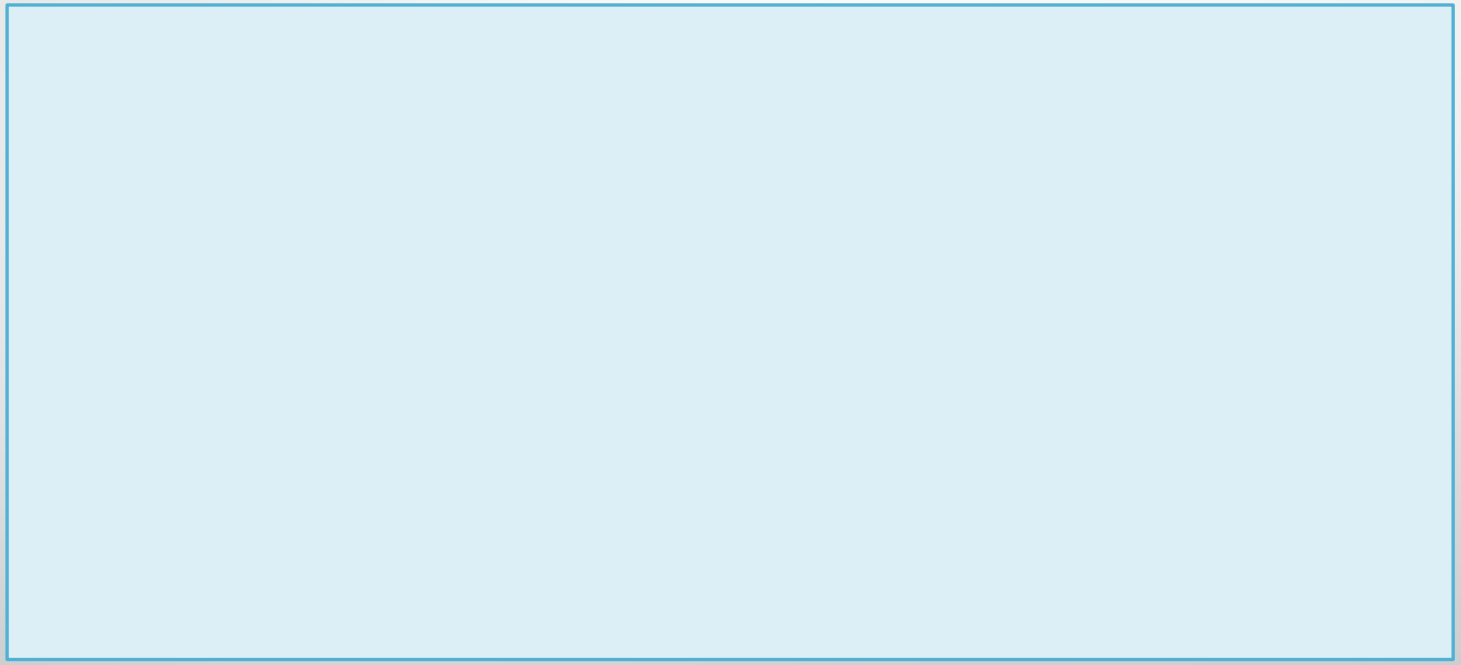
- *Every* solver must be installed before you can use it

```
Pkg.add("Clp")
```

- Installation takes a couple minutes, but only needs to be done once
- Julia precompiles newly installed/updated packages the first time you use it (5-30sec)
- Keep all packages up to date w

```
Pkg.update()
```

Remember Top Brass?



Let’s (finally) solve it

Solving Top Brass with different solvers:

Top Brass in Julia

Do we get the same answer with Clp, ECOS, and SCS?

Compare solver performance with

```
@time(optimize!(m))
```

What happens if we use a solver that isn't "optimized" for our model type?

Solving Top Brass in Juli

Note: This may be a helpful walkthrough if you don't understand the example posted on the previous slide. This (and all other Julia walkthroughs) is **optional!**

- Video for "Top Brass" is posted on Canvas (under Kaltura Gallery)

Separating the *data* from the *structure*

We just built an instance of a **linear program (LP)**

- What makes this a “linear” program?

Our complete model had **decision variables** and **parameters** (data)

- By changing *parameters*, we get different *instances*
- Typically, we want to separate the data (parameters) from the algebraic structure (model) (why?)

General modeling

$$\underset{x_f, x_s}{\text{maximize}} \; 12x_f + 9x_s$$

subject to

$$4x_f + 2x_s \leq 4800$$
$$1x_f + 1x_s \leq 1750$$
$$0 \leq x_f \leq 1000$$
$$0 \leq x_s \leq 1500$$

$\longrightarrow$

$$\underset{x_f, x_s}{\text{maximize}} \; c_1 x_f + c_2 x_s$$

subject to

$$a_{11}x_f + a_{12}x_s \leq b_1$$
$$a_{21}x_f + a_{22}x_s \leq b_2$$
$$l_1 \leq x_f \leq u_1$$
$$l_2 \leq x_s \leq u_2$$

By changing **parameters**, we get new **instances**

Some tips for writing modular code

Reminder: it is a good idea to separate the *data* from the *model* (more general)

Use ***dictionaries*** for more modular code

Use ***expressions*** for more readable code

NamedArrays can be used to index over sets

To see why more modular code is better, try adding a new type of trophy

Top Brass Modular.ipynb

Top Brass Compact.ipynb

Modular Top Brass

Note: This may be a helpful walkthrough if you don't understand the example posted on the previous slide. This (and all other Julia walkthroughs) is **optional!**

- Video for "Top Brass Modular" is posted on Canvas (under Kaltura Gallery)

Note: This may be a helpful walkthrough if you don’t understand the example posted on the previous slide. This (and all other Julia walkthroughs) is **optional!**

- Video for “Top Brass Compact” is posted on Canvas (under Kaltura Gallery)

Julia Module Learning Outcomes

Now you should…

- Understand why we use Julia in this class
- See how to run the most recent versions of Julia, IJulia, and JuMP
- Begin to understand basic functionality and important tips & tricks for using Julia effectively
- See how we build and solve an optimization model in Julia/JuMP with an example