



# LP Modeling Using Julia/ JuMP

---

15.S60, IAP 2014

Velibor Mistic

# Our Goal

- In OR-related research, chances are you'll need to solve a linear or mixed integer program at some point.
- In this class, we're going to learn how to model LPs and MIPs in Julia, solve them, and do interesting things with them.
- We're going to use a new programming language called **Julia** and a package for Julia called **JuMP (Julia for Mathematical Programming)**.

# Background of Julia and JuMP

- Julia:
  - “high-level, high-performance, open-source dynamic language for technical computing”
  - Dynamic nature allows rapid development, but Julia is also *fast*
    - Within factor of 2 of C/C++
  - See Lubin and Dunning (2013) (<http://arxiv.org/abs/1312.1431>) for comprehensive experiments
- JuMP:
  - Algebraic Modeling Language (AML) for Julia developed by Miles and Iain

# Outline

- Julia basics
- JuMP basics
- Facility location exercises
- Network revenue management exercises

# References

- For further information on Julia:
  - Speed tutorial: <http://learnxinyminutes.com/docs/julia/>
  - Doc: <http://docs.julialang.org/en/latest/manual/>
- For further information on JuMP:
  - Doc: <https://jump.readthedocs.org/en/release-0.2/jump.html>
  - Issues: <https://github.com/JuliaOpt/JuMP.jl/issues>
- You may find it useful to have these open as we go through the class.

# Getting started with Julia

- Open up a terminal window, and type in
  - `julia`

# Basic stuff

- Simple arithmetic:
  - $3 + 7$
  - $4.8 / 9.3$
  - $11^5$
- Comparisons:
  - $12 \leq 5$
  - $!(5 == 8)$
  - $5 \neq 8$
  - $(2014 \% 4) == 0$
- Strings:
  - "Velibor and Vishal are office mates"
  - "Velibor has  $\$(25 \% 2)$  cat"

# Variables

- Variables are easy:
  - $x = 5$
  - $y = 12$
  - $z = x - y$
  - $z$
  - $w$
- Everything has a type:
  - `typeof(z) #Int64`
  - `typeof(z + 1.5) #Float64`
  - `typeof( z >= 0) #Bool`



# Arrays

- Arrays are for sequences of values, indexed from 1 to n:
  - `v = [1, 2, 3]`
  - `v(1)` # you'll get an error – different from MATLAB
  - `v[1]` # access elements using `[ ]`; starts from 1
  - `v[0]` # gives an error!
  - `v[end]` # last element
- 2D arrays are made the same way as in MATLAB:
  - `A = [1 1 1; -1 -1 -2; 0 0 0]`
  - `B = [0.1 0.5 0.4; 0.2 0.3 0.5; 0.8 0.1 0.1]`
  - `B[2,3]`
  - `A + B`
  - `B^100`
  - `B[1,:]`

# Arrays continued

- Modifying arrays:
  - `a = [1, 2, 3]`
  - `a[end+1] = 4` # won't work!
- Push/pop commands:
  - `push!(a, 4)` # append 4 to end of a
  - `pop!(a)` # remove last element of a; 4
  - `a` # a is back to being [1, 2, 3]
  - `z = []`
  - `push!(z, 5)` # what happens?
  - `z = Int64[]`
  - `push!(z, 5)` # how about now?

## Exercise 1a

- Create the arrays A, x and b as follows:
  - `A = rand(10,3); x = rand(3,1); b = rand(10,1)`
- Write an expression that evaluates to
  - **true** if the inner product of the fourth row of A and x is greater than the fourth element of b; and
  - **false** otherwise.
- What do you find?

# Exercise 1a's lesson

- Julia is much stricter with arrays than MATLAB!
- For example:
  - `A = rand(10,3); x = rand(3,1); b = rand(10,1)`
  - `A[4,:] * x - b[4]`
  - `A[4,:] * x - b[4] < 0` # what happens?
  - `(A[4,:] * x - b[4])[1] < 0` # how about now?
- Yet another trap!
  - `z = [1:1:3]` # what is it?
  - `z[:]` # what do you get?
  - `z[:,:]` # how about now?

# Initializing arrays

- List comprehensions:
  - `oddNumbers = [ 2*i - 1 for i in 1:10]`
  - `evenNumbers = [j + 1 for j in oddNumbers]`
- Can build matrices this way:
  - `A = [ i + j for i in 1:5, j in 1:5]`

# Tuples

- Tuples are “fixed” arrays:
  - `t = (1, 2, 3)`
- Can access them like arrays, but can’t change them:
  - `t[1]` # gives back 1
  - `t[1] = 5` # error!
- Tuples can be initialized in other ways:
  - `x = tuple(1, 2, 3)`
  - `y = [1:5]`
  - `z = tuple( y... )`

# Dictionaries

- Dictionaries are like arrays, except they can be indexed by arbitrary objects:
  - `mydict = Dict()` # makes an empty dictionary.
  - `mydict["Velibor"] = "is alright"` # add key-value pairs...
  - `mydict["Vishal"] = "is awesome!"`
  - `myotherdict = ["King's Landing"=> 1, "Winterfell"=> 2, "Qarth"=>3]`
  - `myotherdict["Pyke"]` # gives an error
- Keys of dictionary:
  - `keys(mydict)`
- Values of dictionary:
  - `values(mydict)`

# Conditionals and loops

- If-then-else:

- `x = 2`
- `if x > 6`
  - `x = x + 20`
- `elseif (x < 5)`
  - `x = factorial(x)`
- `else`
  - `println("Not changing x")`
- `end`

- for loops:

- `for k = 1:5`
  - `println( k^2 )`
- `end`

- while loops:

- `while x < 2000`
  - `print("*")`
  - `x = 2 * x`
- `end`



# Functions

- Functions are defined using the keyword function:
  - ```
function convexcomb(x,y, theta)
    (1 - theta) * x + theta * y
end
```
- Functions by default return the last statement, but you can also use return:
  - ```
function convexcomb2(x, y, theta)
    if (theta < 0 || theta > 1)
        println("Bad theta!!!")
    else
        return (1 - theta) * x + theta * y
    end
end
```

## Exercise 1b

- Write a function that takes a matrix  $A$ , vectors  $x$  and  $b$ , and returns a vector containing the row indices for which  $Ax \geq b$  does not hold.
- There are many ways to do this!
- Hints:
  - `.<` performs element-wise comparisons
  - “find” function returns indices of non-zero elements for Int64 or Float64 arrays, true elements for Bit arrays

# Scripts in Julia

- Julia scripts are plain text files with the extension “.jl”.
- In a terminal window, you can run one (say script.jl) by passing it as a command line argument to Julia – e.g.,
  - `julia script.jl`

# Scripts continued

- Within a session, you can use the keyword 'using' to load everything in a script.
- E.g. suppose test.jl was:
  - ```
function VeliborsFunction(x, y)  
    return gamma(x)*gamma(y)/gamma(x+y)  
end
```
  - ```
println("Velibor's function loaded!")
```
- In Julia session:
  - ```
using test.jl
```
  - ```
VeliborsFunction(1,2)
```
  - ```
VeliborsFunction(0.5,0.5)
```



Time to JuMP!

# Why an AML?

- Algorithms like simplex work with an optimization problem that is expressed in the form

$$\begin{array}{ll}\text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0\end{array}$$

- Functions like linprog take A, b, c explicitly as inputs, solve this problem and return an optimal x.
- While it is possible to represent the data A, b, c explicitly for any given model, doing the conversion – from a mathematical description, to solver-ready form – can be quite laborious.

## Why an AML? (continued)

- An AML allows you to specify an optimization problem in a compact way that you, as a human being, can understand.
  - Optimization variables are represented by programming variables.
  - Constraints are built with expressions resemble.
  - Throughout, we can use all of the tools available to us from programming languages – conditionals, loops, functions, etc. – to build our model.
- Once the model is built in this way, the AML takes care of converting your description to solver-ready data (A, b, c).

# Let's model a simple LP

- Run the following:
  - using JuMP
  - `m = Model()`
  - `@defVar(m, x >= 0)`
  - `@defVar(m, 0 <= y <= 3)`
  - `@addConstraint(m, x + y <= 3)`
  - `@addConstraint(m, 5x + 3y <= 11)`
  - `@setObjective(m, Max, 1x + 4y)`
  - `print(m)`
- What did we just model?



# Now solve it!

- Run the following:
  - `status = solve(m)`
  - `println("Z = ", getObjectiveValue(m))`
  - `println("x = ", getValue(x))`
  - `println("y = ", getValue(y))`

# Without an AML, in MATLAB

- In MATLAB, with `linprog`:
  - `c = [-1; -4];`
  - `A = [1 1; 5 3];`
  - `b = [3; 11];`
  - `x = linprog(c, A, b, [], [], [0;0], [inf; 3])`

# Without an AML, in Excel

| Decision variables | x | y |
|--------------------|---|---|
|                    | 0 | 3 |

| Constraints |   |    |    |
|-------------|---|----|----|
|             | 3 | <= | 3  |
|             | 9 | <= | 11 |

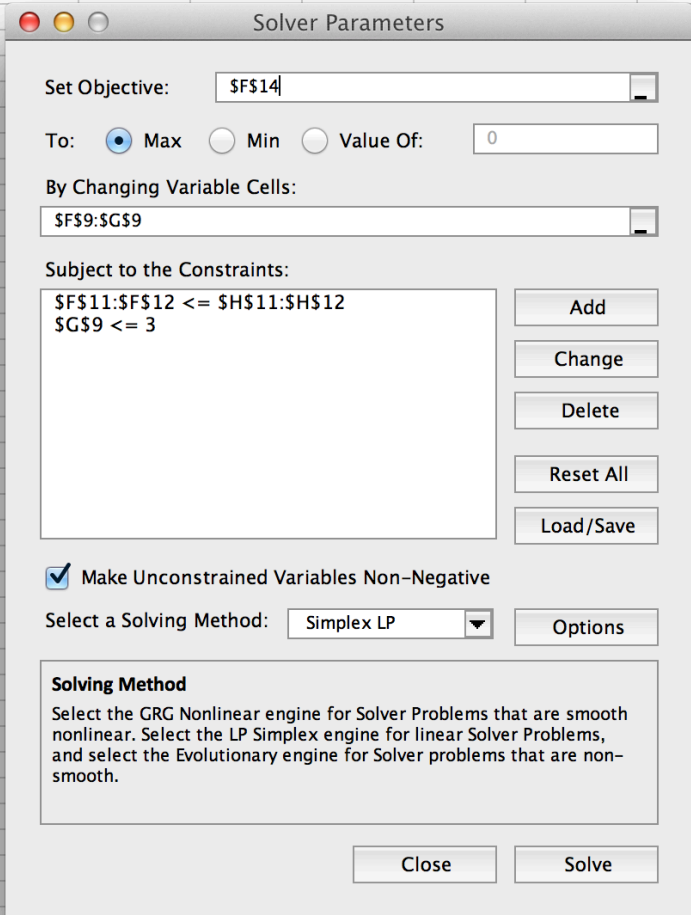
| Objective |    |
|-----------|----|
|           | 12 |

| A matrix values |   |   |
|-----------------|---|---|
|                 | 1 | 1 |
|                 | 5 | 3 |

| Obj. coefficients |   |   |
|-------------------|---|---|
|                   | 1 | 4 |

The Solver Parameters dialog box is shown, configured for a linear programming problem. The 'Set Objective' field is set to '\$F\$14'. The 'To' options are 'Max', 'Min', and 'Value Of: 0'. The 'By Changing Variable Cells' field is set to '\$F\$9:\$G\$9'. The 'Subject to the Constraints' list contains two constraints: '\$F\$11:\$F\$12 <= \$H\$11:\$H\$12' and '\$G\$9 <= 3'. The 'Make Unconstrained Variables Non-Negative' checkbox is checked. The 'Select a Solving Method' dropdown is set to 'Simplex LP'. The 'Solving Method' section provides instructions on selecting the appropriate engine for different problem types. The 'Close' and 'Solve' buttons are at the bottom.

# Building more general models

- More general ways to build things:
  - `m2 = Model()`
  - `@defVar(m2, x[1:10, 1:2] >= 0)`
  - `@addConstraint(m2, sum{ x[i,j] , i = 1:10, j=1:2} <= 1)`
- `primes = [2, 3, 5, 7]`
- `@addConstraint(m2, sum{x[i,j] , i in primes, j = 1:2} >= 0.2)`
- `@addConstraint(m2, sum{x[i,j], i = 1:10, j = 1:2; i + j <= 5} >= 0.05)`
- `myTest(i,j) = i + j <= 5    # what did we do here?`
- `@addConstraint(m2, sum{x[i,j], i = 1:10, j=1:2; myTest(i,j)} >= 0.05)`

# Solver options

- The Model() constructor accepts a specification of a solver – e.g.,
  - using JuMP, Gurobi
  - `m = Model(solver = GurobiSolver() )`
- The solver constructor (GurobiSolver()) accepts parameters. Parameter names/values follow the same naming/meaning as within the solver – e.g.,
  - `m = Model(solver = GurobiSolver(Method = 2, Crossover = 0) )`
  - `m = Model(solver = GurobiSolver(Presolve = 0))`

# Facility Location

minimize 
$$\sum_{i=1}^M \sum_{j=1}^N d_{ij} x_{ij}$$

subject to 
$$x_{ij} \leq y_j, \quad \forall i = 1, \dots, M, \quad j = 1, \dots, N,$$

$$\sum_{j=1}^N x_{ij} = 1, \quad \forall i = 1, \dots, M,$$

$$\sum_{j=1}^N y_j \leq K,$$

$$0 \leq x_{ij} \leq 1, \quad \forall i = 1, \dots, M, \quad j = 1, \dots, N,$$

$$y_j \in \{0, 1\}, \quad \forall j = 1, \dots, N.$$

# Without an AML...

- In MATLAB, using linprog?
  - Think about how you would need to keep track of indices of many rows and columns of A matrix will have double indices (rows because of  $x_{ij} \leq y_j$ , columns because of  $x_{ij}$ ).
  - This can be nasty to debug (trust me).
- In Excel, using Solver?
  - Could potentially handle it – have large plots of spreadsheet space where rows and columns correspond to  $i$  and  $j$ .
  - But what if you have new data? What if the dimensions of the problem change?

## Exercise 2

- Suppose that we live in a 1D world, where customers and facilities are located on a line as follows:
  - `customerLocs = [3, 7, 9, 10, 12, 15, 18, 20]`
  - `facilityLocs = [1, 5, 10, 12, 24]`
  - `d_ij` = absolute value distance between customer *i* and facility *j*
- Model and solve this problem in Julia using JuMP!  
(Set  $K = 3$ .)
- HINT: to make a variable binary, use “Bin” in `@defVar`:
  - `@defVar(m, z[1:5], Bin)`



## Exercise 3

- In MIPs, it is sometimes useful to know not only the best solution, but also the second best, third best and so on.
- Modify your code so that, after the first solve, the model is solved another three times.
- Each time, add a constraint that cuts off the current facility decision, re-solve the model and output the objective value.

## Exercise 3 Hint

- Suppose our current solution is

$$\{\bar{y}_j\}_{j=1}^N$$

- Suppose

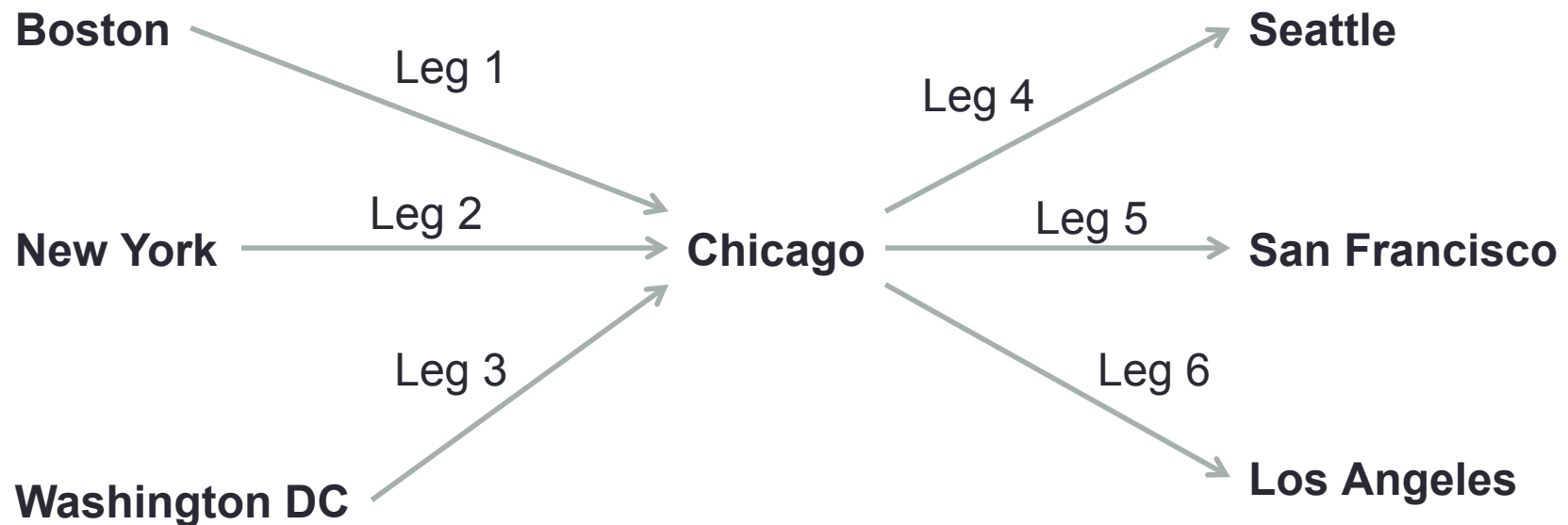
$$J_1 = \{j \mid \bar{y}_j = 1\} \quad J_0 = \{j \mid \bar{y}_j = 0\}$$

- Then the following constraint cuts off the current solution:

$$\sum_{j \in J_1} y_j + \sum_{j \in J_0} (1 - y_j) \leq N - 1$$

# Network revenue management

- Suppose we are running an airline that operates the following network:



# Network revenue management - II

- We sell fares that may use capacity on one or two of the links in the network.
  - E.g.: we may sell Boston  $\rightarrow$  Chicago, Chicago  $\rightarrow$  LA, or Boston  $\rightarrow$  LA *via* Chicago.
- For now, we'd like to decide how many requests of each fare to accept, so as to maximize our revenue, subject to:
  - How much capacity is on each flight.
  - How many requests we expect of each fare, given that we are selling over  $T$  days.

## LO Formulation

$$A_{\ell,f} = \begin{cases} 1 & \text{if fare } f \text{ uses link/leg } \ell, \\ 0 & \text{otherwise.} \end{cases}$$

$R_f$  = revenue from fare  $f$

$p_f$  = probability of daily request  
being of fare  $f$

$T$  = length of horizon

# LO Formulation

$$\text{maximize} \quad \sum_{f \in \mathcal{F}} r_f x_f$$

$$\text{subject to} \quad \sum_{f \in \mathcal{F}} A_{\ell, f} x_f \leq c_{\ell}, \quad \forall \ell \in \mathcal{L},$$

$$0 \leq x_f \leq T \cdot p_f, \quad \forall f \in \mathcal{F}.$$

# Data

- We have the following data:
  - legs = [1, 2, 3, 4, 5, 6]
  - fareLegs = [ (1), (2), (3), (4), (5), (6), (1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6)]
  - fareProbabilities = [0.06, 0.096, 0.046, 0.073, 0.159, 0.067, 0.043, 0.019, 0.112, 0.075, 0.031, 0.044, 0.012, 0.0210, 0.1130]
  - fareRevenues = [40, 30, 30, 10, 40, 10, 190, 80, 90, 70, 60, 190, 60, 50, 10]
  - legCapacities = [20, 20, 20, 20, 20, 20]
  - T = 100

## Exercise 4

- Formulate the LO problem in Julia and JuMP, and solve it. What is the optimal revenue?
- Hint: what happens when you try this:
  - `z = (1, 2, 3)` # what is the type of `z`?
  - `2 in z` # what happens?
  - `4 in z` # what happens?



## Exercise 5

- Create a function to solve the problem, that takes
  - legs, fareLegs, fareProbabilities, fareRevenues, legCapacities, T
- as inputs.
- You may use Ex5\_NRM\_Function\_test.jl to test it.

# Constraint references

- Like variables, constraints can have a “name” – in JuMP, this is a *constraint reference*.
- Constraint references are created using the `@defConstrRef` macro.
- The output of `@addConstraint` can then be assigned to a constraint reference.
- Example:
  - `m = Model(:Max)`
  - `@defVar(m, x[1:4] >= 0)`
  - `@defConstrRef myCons[1:4]`
  - `for i = 1:4`  
`myCons[i] = @addConstraint(m, x[i] <= 10)`

# Dual information

- JuMP also provides a function, `getDual`, for obtaining dual information.
- If `cRef` is a constraint reference, then `getDual(cRef)` returns the shadow price of the constraint.
- If `var` is a variable, then `getDual(var)` returns the reduced cost of the variable.

## Exercise 6

- Modify the function from Exercise 5 to return the shadow prices of the leg constraints in an array, as well as the optimal objective.
- You may use `Ex6_NRM_DualFunction_test.jl` to test it.

# Online NRM

- In reality, the problem is an *online* problem.
- On each day, there is a request for a fare. We must decide whether to accept it or reject it.
  - Accepting a fare now gives us revenue, but uses capacity that could be used on more lucrative fares later.
  - Rejecting a fare keeps our capacity intact, but we do not earn any revenue.
- This is a notoriously difficult stochastic control problem.

# Bid-price control

- Let

$\lambda_\ell(\vec{c}, t)$  = dual price of leg  $\ell$ 's capacity constraint,  
when the remaining capacity is  $\vec{c}$   
and  $t$  days remain.

- In BPC, the action to take (when there is sufficient capacity) is given by comparing the dual prices to the revenue of the fare:

$$\pi_t(f) = \begin{cases} \text{Accept} & \text{if } \sum_{\ell \in \mathcal{L}} A_{\ell, f} \lambda_\ell(\vec{c}, t) \leq r_f, \\ \text{Reject} & \text{otherwise.} \end{cases}$$

## Exercise 7

- Ex7\_NRM\_SimulateControl.jl contains two simulation functions – one for BPC and one for a greedy policy. The BPC function is missing two lines of code that you need to fill in.
- To test it, use Ex7\_NRM\_SimulateControl\_test.jl.
- How does BPC compare to the greedy policy?

# Thank you for listening!

- For further information on Julia:
  - Speed tutorial: <http://learnxinyminutes.com/docs/julia/>
  - Doc: <http://docs.julialang.org/en/latest/manual/>
- For further information on JuMP:
  - Doc: <https://jump.readthedocs.org/en/release-0.2/jump.html>
  - Issues: <https://github.com/JuliaOpt/JuMP.jl/issues>
  - Contact Miles and Iain
- If you find JuMP useful, please cite Miles and Iain's paper!
  - M. Lubin and I. Dunning, "Computing in Operations Research using Julia", under review.