Research Assistant Work

JuMP:

Introduction - Read

Read lightly over "Getting started with sets and indexing"

- Sets are often used to represent some variable in a sample with multiple observations

- Unordered sets can be used for representing non-numeric vectors, such as Strings. They are most commonly initialized through a standard vector. Other methods include using a dictionary (each key is associated with some value), and using Julia's Set functionality, which allows you to iterate through elements of a vector/dictionary and assign the result to a new set. This method will also remove duplicate elements

- Numbered sets are used for numeric variables. These are often initialized using ranges to automatically create a vector that includes each element within the range. You can use the syntax [start:step:stop] to initialize a vector that goes from start to stop in steps the size of "step".

- You can also use Tuples as sets through [(tuple\_name)] or iterating through tuple combinations. For multi-dimensional sets, you can construct containers using x[tuple\_1,tuple\_2], where tuple\_1 is the first dimension and tuple\_2 is the second dimension.

- Set operations include setdiff(x,y) which returns all elements that are not common between x and y. Another operation is union(x,y) which combines the elements of x and y vectors. Length(x) returns the number of elements in a vector.

- You can test for set membership using the "in" keyword, and combine with "for" and "if" keywords in expressions to filter and create new sets.

- For compound sets, where each element includes a function(s) of other elements, such as distance between cities, we can use a matrix to represent the set. We can set model objectives using some function of the variables. We can also use containers as indices by converting tuples to index values.

- Data conversion to a more intuitive form is useful.

Tutorial - Linear Programming

- Some (linear) optimization function according to linear constraints.

- Mixed-integer Linear Programs (MILPs) enforce some level of integrality where variables take discrete values.

- Knapsack problem: Maximize some value (c) associated with each state of i (objective function can be considered nutritional value, e.g.) according to some constraint (cost) on the number of i's that can be used through the variable w (cost per item). x is an indicator function for whether each i is activated (purchased). Can be written as a function of the cost and value vectors multiplied by a transposed "x" vector, with the summed elements of the resulting vectors corresponding to the objective and constraint functions. x is to be determined by the model here as the optimal set of decisions x for each i.

- Syntax: @variable to determine decision variables, @constraint to determine constraint function (cost), @objective to determine objective function. Use function optimize!(model) to solve program and solution\_summary(model) to summarize the results. @assert sets important variables for the model to run. Bin means a binary variable.

- Once you have created your model, it is a good idea to build a function around it that takes vectors/variables as inputs and computes the results, returning the decision variables outputted by the solver.

Time: 4 Hours

Week 2:

Diet Problem

* Objective function: Minimize cost based on some quantity of food >=0
* Constraint: Total amount of each macronutrient across all foods must be within the acceptable range. This is done for each macronutrient and summed across all foods
* Data: Create dataframe with each food, including cost and macronutrient profile (per quantity)
  + Then create dataframe with minimum and maximum bounds for each macronutrient
  + Set decision variable >=0, with the number of variables equal to the number of foods
  + We then set the objective function (min cost -> min sum(food\_cost \* x) over all food items f)
  + We then define our constraints, (min macro < sum(x\*macronutrient per unit) over all foods f < max macro, with separate constraints for each type of macronutrient)
  + Can code easily using row in eachrow(table), and then use each row for individual constraints, rather than manually creating each constraint
  + You can add or delete constraints, and use set\_integer to force a variable to take an integer

Cannery Problem

* Moving products from production plants “p” to endmarkets “m”. Demand for each endmarket is denoted “d(m)”, while the capacity for each production facility is denoted “c(p)”. The shipping cost of transferring a unit from a production facility to a market (can also be thought of as distance) is denoted “d(p,m)”. Objective is to minimize total cost (d(m,p)\*x(m,p) summed over M, summed over P).
* Constraints include production in each plant (or shipments from each plant) being less than or equal to capacity c(p), and shipments to each endmarket being at least equal to the demand in that market d(m)
* Data: In this example, data is loaded from JSON using JSON.parse function. Data provided includes markets with demand figures, production facilities with capacity figures, and distances for all combinations of production facilities and endmarkets
* Constructs a dictionary\*\* Would like to review this syntax, nested dictionary?\*\*
* Function to compute distance from plant to market \*\*want to review this function syntax\*\*
* Decision variables are initialized for all combinations of plants and markets (matrix)
* Constraint that each plant cannot ship more than capacity \*\*syntax\*\*
* Constraint that each market must receive at least its demand
* Objective is set as minimizing transportation distance, or minimize the product of units delivered and distance
  + Would likely (?) be best to minimize based on batches (or units/flight), instead of individual units, although this doesn’t seem to be an issue with the final answer

Factory Schedule Example

* Assumes we are optimizing production of some good from a set of factories “f” over the course of 12 months “m”.
* Each unit of production carries a variable cost “c(f)” per unit (depends on factory), and fixed cost “a(f)” per month.
* Z(m,f) is an indicator variable for whether the factory runs or shuts down. Where z = 0 (factory shut down), no fixed or variable costs are incurred
* Minimize the total cost, or
* Where monthly demand “d(m)” is exactly satisfied each month. In other words, production across all factories is equal to demand in each month.
* However, this model may be flawed, as a sufficiently high demand against maximum production capacities may lead to infeasibility. We can counter this by adding a new variable to account for unmet demand (purchase from other suppliers or running overtime), with an arbitrarily large cost factor
* Data: File as .txt, read into Julia as CSV and convert to dataframe type using CSV.read()
* Validating data can be a useful practice, as it helps avoid typos that lead to flawed figures. In this example, data is screened to ensure that minimum production does not exceed maximum production in any instances, and that variables are not less than 0.
* For sensitivity analysis, we analyze how the optimal objective value changes as variable costs change
* In this example, variable costs are scaled at factors between 0 and 1.5

Transportation Problem

* Set of factories and retail stores, aim is to minimize cost. Factory set is defined as a vector of instances “i” (origins), and retail stores are represented by a vector of instances “j” (destinations)
* Maximum supply (capacity) at each factory is “s(i)”, and demand from each store is “d(j)”
* Decision variable is x(i,j), representing the number of units from “i” to “j”, with per-unit shipping cost c(i,j)
* Minimize total cost, or cost\*units for each factory-endmarket pair

Multi-commodity Flow

* The multi-commodity problem is an extension to the transportation problem with multiple types of products. We add the extra parameter, “k” (instance of vector of product types) to existing constraints, and create a new constraint that limits the total number of units (across products) that can be transported between i and j pairs

Network Multi-commodity Flow Problem

* Similar to the multi-commodity flow problem, except that instead of having factories and endmarkets as distinct nodes, each node has both a supply capacity and demand level (although either can also be zero). Each node is represented by some instance “i” in vector N. Instances (i,j) in E represents the connections between all nodes, with shipment cost c(i,j,p), where “p” represents different commodity (or product) types
* Supply capacity for each node “u(I,p) and shipping capacity “u(I,j) between nodes cannot be exceeded
* Deicision variables are “x(i,j,p)”, or the quantity of product “p” being transported from “i” to “j”, and s(i,p), or the production quantity of each factory s(i,p)
* Total demand must be met by the sum of supply and net inflow

Tips and Tricks

* Absolute Value function, where t>=|x|
  + Option 1: Create two linear inequality constraints using both x and negative x. Therefore, regardless of x’s sign, the absolute value will be tested
  + Option 2: Use two non-negative variables and create expressions t+x and t-x. Can be used to test t>=-x (negative x) and t>=x (positive x)
  + Option 3: Using MOI.NormOneCone function
* L1-norm (Manhattan distance/vector magnitude sum)
  + To calculate the minimum Manhattan distance, use MOI.NormOneCone function
* Infinity-norm (Largest magnitude among each element of a vector, regardless of direction)
  + To calculate the smallest infinity-norm in a set of vectors \*\*not sure of this\*\*, use MOI.NormInfinityCone function
* Max
  + To model t>=max{x,y}, create constraints for both t>=x and t>=y and condition on both of them being true
* Min
  + To model t<=min{y,x}, use the same approach as above except with t<=y,x conditions and require both to be true
* Modulo (common divisor and remainder -> 5%2 = 2 (remainder 1)
  + Set constraint where y is between 0 and n-1 (where y is the remainder)
  + And set a constraint where z is any number greater than or equal to 0 (common divisor)
  + Set x greater than or equal to 0 (dividend)
  + Set x == n\*z + y, or ensure that x is equal to the remainder plus the divisor\*modulo quotient
* Boolean Operators
  + Or (x3=x1 OR x2, where x is binary)
    - Create 3 element binary vector (0 or 1), and set a constraint where x[3] must be <= x[1], x[2] <= x[3], and x[3] <= x[1] + x[2]
  + And (x3 = x1 AND x2), where x is binary
    - Create 3 element binary vector where x[3] <= x[1], x[3] <= x[2], and x[3] >= x[1] + x[2] -1
  + Not (x1 not x2)
    - Set constraint where x[1] + x[2] = 1, where x is a binary vector
  + Implies (x1, therefore x2)
    - Set x[1] <= x[2], such that if x[1] is 0, then x[2] can be 1 or 0. If x[1] is 1, then x[2] must be 1
  + Disjunctions (“or”)
    - Use “big-M” multiplied by a binary variable to relax one of the constraints
    - Introduce binary variable y that takes values 1 or 0, activating “big M” (large constant) to make one of the expressions true, and only testing the other expression
  + Indicator constraints, where an indicator activates a constraint
    - To model x1+x2 <= 1 if z=1, use “-->” operator, where z is a binary variable and x is a 2 element vector in the function z-->{sum(x) <=1})
    - For modelling the same constraint when z=0, use !z --> {sum(x)<=1})
    - If the --> operator does not work, use big M
  + Semi-continuous Variables
    - Can be modelled using the semicontinuous() function, or using a binary reformulation
  + Semi-integer Variables
    - Can use the semiinteger() function, or using binary reformulation with x being initialized as an integer type
  + Special Ordered Sets of Type 1 (set of variable which at most 1 can take on a non-zero value)
    - Use MathOptInterface.SOS1
  + Special Ordered Sets of Type 2 (same as type 1, except two elements can be non-zero but they must be consecutive)
    - Use MathOptInterface.SOS2
  + Piecewise Linear Approximations
    - SOSII constraints are most often used
    - We create a function with a set of x-values and y-values
    - Set N equal to the number of points you have
    - Model in constraint using combination of lambda, y-hat and x-hat \*\*unsure how this calculation works\*\*

Time: 8.5 hours