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

Week 3:

Approximating Non-linear Functions

* Approximating non-linear functions with a mixed-integer linear program
* If a function that you are approximating is convex, you can minimize down onto it using an outer approximation
  + We can use the following inequality due to concave-up nature of function:
    - F(x) >= f(x1) + df(x1)(x-x1)
    - As we add more planes, the error between the true function and the piecewise linear approximation decreases
* If we want to approximate a concave (downward acceleration) function, we can maximize “up” to it using the following inequality:
  + f(x) <= f(x1) + df(x1)(x-x1)
  + Example is 1/x
  + Adding more points of x1 leads to more accurate approximation
* You can also use an inner approximation method for non-linear functions. \*\* Model formulation, not sure how lambda works\*\*
  + \*\*Essentially plots points across the function and connects them linearly, then sets minimum/maximum for the function\*\*
  + Can be done for both concave and convex
* Piecewise linear approximation
  + If a model is non-convex or non-concave (think: sin/cos), then traditional inner/outer approximation becomes infeasible. Outer breaks down where concave up switches to concave down, and inner does not stay near the line (?)
  + We can force inner approximation to stay on the linear lines by using SOS2, that ensures at most two elements of lambda can be non-zero, and if they are, they must be adjacent

Facility Location Problem

* Uncapacitated facility location
  + We are given a set of clients M = {1,…,m} and a set of sites N= {1,…,n} where a facility can be built
  + Decision variables are split into 2 categories:
    - Binary variable y(j) indicates whether factory “j” is built or not
    - Binary variable x(i,j) indicates whether client “i” is assigned to facility “j”
    - Objective: Minimize the total cost of serving all clients. This cost breaks down into 2 components:
      * Fixed cost of building a facility f(j)
      * Cost of serving clients from assigned facility (in this case, the Euclidean distance between the two) c(i,j)
    - Constraints:
      * Each customer must be served by exactly one facility
      * A facility cannot serve any client unless it is open
    - MILP Formation
      * Minimize total cost, or c(i,j)\*x(i,j) for all factory-client combinations + f(j)\*y(j) for all factory locations
      * Ensure that each client can only be served once (each client is served by exactly one facility)
  + Capacitated facility location
    - Introduces capacity constraints, where each client “i” has some demand a(i) >= 0, and each facility “j” has a finite capacity q(j) which cannot be exceeded
    - Additional constraint ensures that the sum of demand from each customer (0 unless x(i,j) = 1) for a facility must be less than or equal to the capacity of the built facility

Financial modelling problems

* Cashflow Requirements
  + Given a set of cash flow requirements (mandatory payments) for a company, and a set of capital sources, we can create a linear model to forecast how we can use each source of capital to meet our requirements and maximize our total money by some point in time
* Combinatorial Auctions
  + The set of items available in the auction is M = {1,2,…,m}, and a bid is defined as variable B(j) = (S(j), p(j)) where S(j) is a set of items from M that are being bid on, and p(j) is the price offer for the set
  + Suppose auctioneer receives “n” bids, the goal is to maximize revenue from all bids
  + Objective function is to maximize revenue, or the sum of offer price from all accepted bids. You can use an indicator function y(j) for whether you accept a bid
  + Subject to only being able to accept one bid per item

Geographical Clustering

* + Goal is to cluster a set of “n” cities into “k” groups that minimizes the total pairwise distance between cities and ensuring that the variance in the total population is relatively small between cities in the same group
  + Starts with a function to compute distance (haversine)
  + Creates lower triangular matrix by calling distance function on each city \*\*why a triangular matrix?\*\*
  + Building the model:
    - Add constraint: Each city must be part of exactly one group
    - Set a maximum difference between group population and target population. When this constraint is relaxed, your cities will be closer together but the variance in populations will be higher
    - We compute the total distance between cities in a group by iterating over all cities using an indicator function z(i,j) for cities within the same group
    - Objective is to minimize dot product of z with the distance matrix

Network Flow Problems

* A flow network is a directed graph with nodes and edges, where each edge has some capacity and receives a flow. Flow cannot exceed capacity
* Shortest Path Problem
  + Suppose that each arc (i,j) is assigned some scalar cost a(i,j). We want to find a path through all the nodes that minimizes cost
  + Minimize, for all pairs of (i,j) the cost of the pair times an indicator function for whether the pair is activated
  + \*\*Don’t understand constraint including purpose of “b” and what the variable “k” represents\*\*
* Assignment Problem
  + There are n people and n objects, where “i” represents each person and “j” represents each object. There is some benefit a(i,j) of matching person i with object j. We want to maximize this benefit
  + Additional constraint is that person i can only be assigned to object j if the pair (i,j) belongs to a given set of pairs in A
  + Want to find the set of person-object pairs that maximizes total benefit
  + Each object can only be assigned once, and \*\*each person (object) must be assigned to one object (person) \*\*
* Max-flow Problem
  + We have a graph with two special nodes, a sink (that only consumes) and a source (that only produces)
  + Objective is to move as much flow as possible from source to sink

Multi-objective Knapsack

* A multi-objective problem has multiple objective functions
* We can modify our original knapsack problem by adding an additional objective. Rather than just maximizing our total profit, we can also look to maximize another function: Total desirability rating
* \*\*How do these functions work? What is the behavior in a scenario with tradeoffs? Different forms?\*\*
* Multi-objective function models provide a range of solutions to illustrate trade-offs b between functions. Best way is to plot the decision variables for each solution. This allows the decision maker to choose a solution that fits their personal preferences

Sudoku

* A popular number puzzle. Rules:
  + The numbers 1-9 must appear in each 3x3 square
  + The numbers 1-9 must appear in each row
  + The numbers 1-9 must appear in each column
  + Objective is to satisfy these rules, in other words not a maximization or minimization problem
  + We can model this problem where all decision variables are binary, or 0-1 integer programming
  + We will initialize a binary variable as an indicator for each possible number in each possible cell occurring. This takes the form x[i,j,k], as an indicator for whether the value k is present at row i and column j.
  + We will then create a constraint that each column and row must contain one of each character
  + We can then create a constraint that each 3x3 square must include each of the 9 digits

N-Queens Problem

* Involves placing a number of queens on an N\*N chessboard such that none of the queens can attack each other
* We can create a variable that represents a matrix version of the chessboard, where each value takes 1 if a queen sits on the board and 0 otherwise
* We can initialize the following constraints to make the model work:
  + There must be exactly one queen in each row/column
  + There can only be one queen on any given diagonal

Constraint Programming

* JuMP supports a range of constraint-programming type constraints through sets in MathOptInterface (MOI)
* Due to the ability to convert models to MIPs, variables can only take on integer forms \*\*is this correct?\*\*
* MOI applications for constraints
  + MOI.AllDifferent set ensures that every element in a list takes a different integer value
  + MOI.BinPacking set can be useful for placing a set of items into different bins given some capacity for each bin
  + MOI.Circuit set is used for constructing a tour of a list of N variables. (From the textbook) “They will each be assigned an integer from 1 to N that describes the successor to each variable in the list” \*\*I don’t understand this. What is the successor?\*\*
  + MOI.CountAtLeast is used to test whether n elements of a set of variables belong to some set of values. For example, we want to ensure that at least one of x[1] and x[2] in our model takes on the value of 3
    - We can use a partition for testing multiple sets of variables simultaneously. The partition is a vector with the number of elements per variable set (or the number of variables per set)
    - Then we can define the value set that the variable elements must be a part of
    - And the number, n, of elements per variable set that must be in the value set
  + MOI.CountBelongs is used to count how many elements in a set of elements belong to some set of values
  + MOI.CountDistinct set is used to count the number of distinct elements in a set of variables
  + MOI.CountGreaterThan is used to limit the number of variables in a set that have a value equal to another variable
    - We compute some value n that is strictly greater than the number of times that variable set y appears in x
    - \*\*Don’t understand the application of this\*\*
  + MOI.Table is used to select a single row in a matrix / 2d list
  + \*\*How do you select the row #?\*\*

Callbacks

* \*\*Don’t understand callbacks\*\*

Sensitivity Analysis of an LP

* lp\_sensitivity\_report function can create a sensitivity report of an LP similar to excel solver
* Sensitivity analysis of an LP asks 2 questions
  + How much can the objective coefficients change before a different solution becomes optimal
  + How much can the right-hand side of a constraint change before the optimal solution changes
  + Sensitivity Report object
    - Provides a tuple for each variable in the objective function with lower and upper bounds that illustrate how much the objective coefficient can change before the optimal solution changes
    - A tuple for each constraint that explains how much the RHS can change before the optimal solution changes
    - We can use a function to create a dataframe for each constraint or variable with upper and lower bounds, reduced cost, and other columns

Week 3 Hours: 10