Computational Decision Making for Regular People

03: Math Modeling With Pyomo

October 29, 2024

Nathan Davis Barrett

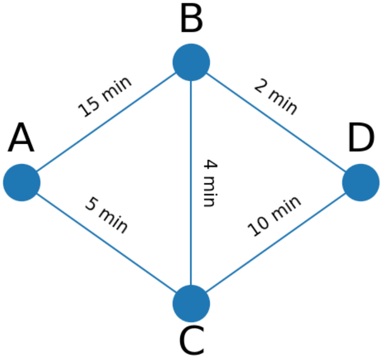
Today's Outline

1. Refresher on Math Modeling (Route Planning Problem)
2. Introduction to Pyomo
   - The Pyomo "ConcreteModel"
   - Sets
   - Parameters
   - Variables
   - Constraints
   - Objectives
   - Solvers
3. Coding and Solving the Route Planning Problem
4. Plotting / Exporting Results
5. My Recommended Pyomo Workflow (The Single Responsibility Principle)
6. "Infeasible", "Unbounded" ?

Refresher on Math Modeling

Given the roads indicated in the graph below, what's the fastest way to get from point A to point D?

1. Objective Function ($f(\overline{X})$)
2. Decision Variables ($\overline{X}$)
3. Parameters ($\overline{\alpha}$)
4. Constraints ($\mathbf{S}$)



Refresher on Math Modeling

— — — FULL FORMULATION — — —

$$\min_{X_r, Y_p} \sum_{r \in \mathbf{R}} \delta_r X_r$$

$$s.t. \quad \sum_{r \in \mathbf{R}_p} X_r = 2Y_p \quad \forall p \in \mathbf{P}^{NON-TERM}$$

$$\sum_{r \in \mathbf{R}_p} X_r = Y_p \quad \forall p \in \mathbf{P}^{TERM}$$

$$Y_p = 1 \quad \forall p \in \mathbf{P}^{TERM}$$

$$X_r, Y_p \in \{0, 1\}$$

Introduction to Pyomo

Before we jump in, please open up the Jupyter Notebook for today:
03_MathematicalModelingWithPyomo/Lecture_03_Notebook.ipynb

- "Pyomo" Stands for "Python Optimization Modeling Objects"
- It is a collection of custom data structures ("modeling objects") that help organize mathematical models in Python.
- The most important strucutres we'll deal with are as follows:
  - ConcreteModel
  - Sets
  - Parameters
  - Variables
  - Constraints
  - Objectives
  - Solvers
- Notice now, and as we move forward, how these data structures line up with the mathematical model elements we've talked about.
- We'll go over each data structure in detail.

Pyomo: ConcreteModel

- In Pyomo, everything pertaining to one mathematical model is contained in one "ConcreteModel" object.
- In your Python code, this will often be called "model" and will be defined something like this:

  ```
  import pyomo.environ as pyo

  model = pyo.ConcreteModel()
  ```
- You can think of the "model" object as the home for everything pertaining to your mathematical model: All sets, variables, constraints, objectives, etc. will live inside one model object.

Pyomo: Sets

- Recall in Lecture 01 how we used sets to reduce the amount of equations we wrote.
- We simply defined a set and then indicated that a certain Variable, Constraint, etc. should be repeated for every element in that set.
- Pyomo has a "Set" object. This is different than a built-in Python set boject.
- Pyomo "Set" objects have some special abilities such as the ability to multiply two sets together to make a new set of all combinations of the original two sets.
- Pyomo "Set" objects are defined like this:
  model.mySet = pyo.Set(initialize=[__elements__])
- Notice how "mySet" is defined as part of the "model" object.
- Example: See Jupyter Notebook

$$\mathbf{R} \rightarrow \text{model.R} = \text{pyo.Set(initialize=[...])}$$

Pyomo: Parameters

- Recall from Lecture 01 that parameters are simply pre-specified constant numbers.
- Pyomo does not have an explicit representation of these parameters.
- Instead, the value of each parameter is stored and communicated to Pyomo using regular Python variables.
- Examples: See Jupyter Notebook

$$\delta_r \rightarrow \text{delta} = \{\text{"AB"}:\ 15,\ \text{"AC"}:\ 5,\ \text{"BC"}:\ 4,\ \ldots\}$$

Pyomo: Variables

- Recall in Lecture 01 how we define decision variables over a set and a domain:

$$X_r \quad \forall r \in \mathbf{R} \, (\textbf{Binary Variable})$$

- Pyomo has a "Var" object that represents a decision variable.
- Pyomo "Var" objects are defined like this:
  model.myVar = pyo.Var(model.mySet,domain=pyo.myDomain)
- Notice how "myVar" is defined as part of the "model" object.
- Typical domains are: pyo.Binary, pyo.Reals, pyo.Integers, pyo.NonNegativeReals, etc.
- To access an individual element within a variable that is defined over a set, use a similar bracket notation used for Python Lists:
  model.myVar[myIndex]
- Example: See Jupyter Notebook

$$X_r \quad \forall r \in \mathbf{R} \, (\textbf{Binary Variable}) \rightarrow$$

model.X = pyo.Var(model.R,domain=pyo.Binary)

Pyomo: Constraints

- Recall in Lecture 01 how we define constraints over a set:

$$\text{Some Relationship} \quad \forall r \in \mathbf{R}$$

- Pyomo has a "Constraint" object that represents each constraint.
- Pyomo "Constraint" objects take a little bit more setup in order to define.
  1. Define a Python function that takes in the following things are arguments: the "model" object, an individual element (or combination of elements) within the set (or combination of sets) of which this constraint is to be defined.
  2. This function should return a relation between different algebraic expressions using Python's built-in relation syntax (==, <=, or >=)
  3. The "Constraint" object can then be defined like this:
     model.myConstr = pyo.Constraint(model.mySet,rule=myFunction)
  4. Pyomo will then call the function ("myFunction") again and again for each element in the set ("mySet") and store the resulting relation in the constraint object ("myConstr").
- Example: See Jupyter Notebook

Pyomo: Objective

- Recall in Lecture 01 that an objective is 1) a mathematical expression, 2) an indication of which decision variables to vary, and 3) an associated "max" or "min" keyword:

$$\max_{X_r, Y_p} \sum_{r \in \mathbf{R}} \delta_r X_r$$

- Pyomo has an "Objective" object that represents this objective.
- Pyomo "Objective" objects are defined like this:
  model.myObj = pyo.Objective(expr=myExpression,sense=pyo.mySense)
- Note that since there are often lots of decision variables you want to vary, Pyomo assumes you want to vary all of the decision variables that you've defined using model.myVar = pyo.Var(...). Because of that you do not need to re-mention them here.
- Here "mySense" is just "maximize" or "minimize" depending on what you want to do.
- As with all Pyomo objects, notice how "myObj" is defined as part of the "model" object.
- Example: See Jupyter Notebook

$\max_{X_r, Y_p} \sum_{r \in \mathbf{R}} \delta_r X_r \rightarrow$

model.myObj = pyo.Objective(expr=sum(delta[r] * model.X[r] for r in model.R),sense=pyo.maximize)

Pyomo: Solver

- Now that we have our mathematical model defined, we need some way to solve it.
- Lots of really smart mathematicians have come up with clever algorithms to solve these problems.
- Pyomo developers have coded these algorithms into "solvers" for us to use.
- Some solvers are used by professionals to game the stock market, manage the electrical grid, etc. Those solvers can be very expensive.
- Other solvers are free (but considerably slower). Those solvers should be suitable for what we'll do in this class.
- Each solver is limited to a certain style of problem: Linear, Quadratic, Mixed-Integer (Binary), Generic Nonlinear, etc.
- Most of the problems we'll do in this class are "Mixed-Integer Linear" or "Mixed-Integer Non-Linear". So we'll use the free "SCIP"[1] solver for what we do in this class.
- If you followed the instructions at the end of Lecture 2, the SCIP solver should already be installed. If not, please visit https://www.scipopt.org/index.php#download

[1] https://www.scipopt.org/

Pyomo: Solver

- In Pyomo, the solver object exists independently from the ConcreteModel object.
- Ideologically speaking, a solver acts on a model to find an optimal solution.
- A Pyomo solver must first be created:
  solver = pyo.SolverFactory("scip")
- The main argument to the SolverFactory function is the name of the solver. If you want to switch between solvers, all you need to do is type the name of a different solver here.
- Once a solver object has been created, it can act on a given model using the "solver.solve" function:
  solver.solve(model,tee=True)
- There are a lot of different arguments you could pass to the solver.solve function to do a variety of things:
  - Print out updates about the solver's progress (tee=True)
  - Set a time limit for the solver
  - Set a threshold of acceptable accuracy of the optimal solution
  - Specify advanced solver settings
  - etc.
- But these additional arguments change depending on which solver you're using and can get pretty complicated. So we'll just use "tee=True".

Pyomo: Accessing Results

- Once the solver.solve function has been executed, assuming the solver was able to find a solution, the solution will be stored within the ConcreteModel itself.
- You can evaluate the value of any variable or expression using the "pyo.value" function.
  - myVarValue = pyo.value(model.myVar[myIndex])
  - myExpressionValue = pyo.value(model.myVar[myIndex1] + model.myVar[myIndex2])
- These results can then be plotted, saved to an excel file, etc. (See Lecture 02)

Full-Blown Example Problem

We'll solve the Route Planning Problem we've been discussing.

Please open up the Jupyter Notebook for Lecture 03.

The Single Responsibility Principle

There is a notion in the world of software developers that
Every function, script, data structure, etc. should have one single Responsibility.

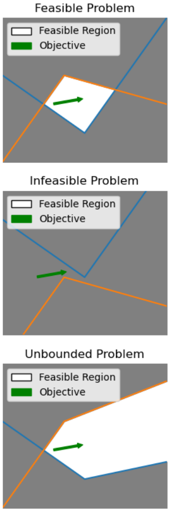
- There are a lot of things to go on in constructing, solving, and interpreting mathematical models.
- Having them all jammed into one Python script can make your code hard to read and more prone to errors.
- I highly recommend breaking each part of this process into it's own function or script.
- One huge advantage of doing this is that if, down the road, you need to run just a small potion of your code, you can just call that one function or script instead of re-running the whole thing or trying to copy and paste pieces of your code that might not work if taken out of context.

My Recommended Pyomo Workflow

- In order to make my code clean and easy to read, I break up my code like this:
  - **Parameters Class**: First define a custom data type (a.k.a. Class) called "Parameters" that houses all the parameters needed to assemble an instance of a model.
  - **AssembleModel Function**: Next define a function called "AssembleModel" that takes in a Parameters object and assembles a Pyomo ConcreteModel object equipped with all sets, variables, constraints, and objective defined. It should then return this model object.
  - **ExecuteOptimization Function**: Next define a function called "ExecuteOptimization" that takes in a fully assembled Pyomo ConcreteModel object and executes the optimization of this model (e.g. creates a Solver object and calls solver.solve)
  - **ExtractResults Function**: Next define a function called "ExtractResults" that takes in a full assembled and optimized Pyomo ConcreteModel object and extracts the results from that object to a format you'd like (generally excel or a plot).
  - **Main Function**: Finally, define a function called "main" that takes no arguments but first defines a Parameters object, uses this Parameters object to assemble a Pyomo Model (using AssembleModel). Then pass this assembled model to ExecuteOptimization, and extract the results using ExtractResults.
- Let's see how this looks for the Route Planning model we've already made (See Jupyter Notebook).

Infeasible? Unbounded?

- Infeasible: There are no solutions that can satisfy all of the constraints you have in your model
  - Try temporarily removing constraints (one-by-one) to try to find the one that is causing problems. That's a good place to start looking...
- Unbounded: The solver says you can keeping traveling forever in the direction indicated by your objective. Thus your optimal objective function value will be infinity.
  - Try adding bounding constraints on your variables (e.g. $X <= 1,000,000$) to make the problem bounded again. Then look at the new solution to see which variable is becoming so large. That's a good place to start looking...



Next Class: Patterns in Formulations

- In the next class we'll tackle how to represent certain real-world behaviors in mathematical models.
  - How can I neatly optimize over a series of time periods, accounts, options, or all of them all at once?
  - How can I handle uncertainty in my model parameters?
  - What if I want different parameters to be selected if I make a certain decision?
  - How can I handle "nonlinear" behaviors like "min", "max", and binary activation in a (mixed-integer) linear way? (I want my models to not take forever to solve)