Pyomo is a python based open-source package for modeling optimization problems. It makes it easy to represent optimization problems and can send it to different solvers (both open-source and commercial) to solve the problem and return the results in python. The advantage of pyomo compared to commercial software such as GAMS and AMPL is the ability to code using standard python syntax (with some modifications for pyomo constructs). Another open source package for modeling optimization problems is JuMP in Julia language.

The simplest option would be to develop the model in pyomo and call it from R using reticulate. However, it still requires writing the pyomo model in python. I want to use reticulate to write the pyomo model using R. In this blog post, I describe two examples in detail where I developed the pyomo model in R and discuss my learnings. I first discuss set-up in terms of packages needed followed by discussion of the two examples.

**Set-up**

# load libraries

library(reticulate)

library(dplyr)

library(tidyr)

library(ggplot2)

For using pyomo package through reticulate, it is necessary to have python and pyomo already installed. We can import pyomo as follows:

# Import pyomo

pyo = import("pyomo.environ", convert = FALSE)

bi = import\_builtins()

When trying different examples, I saw some issues when not using the convert = FALSE option and so started using this option when importing pyomo.

I also ran into some signal handling errors when using pyomo solvers. It seems to have worked but I don’t know yet if it has other side effects.

pyulib = import("pyutilib.subprocess.GlobalData")

pyulib$DEFINE\_SIGNAL\_HANDLERS\_DEFAULT = FALSE

**Example 1: Rosenbrock Problem (Unconstrained Nonlinear Optimization)**

The first example is a unconstrained nonlinear minimization of [rosenbrock function](https://en.wikipedia.org/wiki/Rosenbrock_function)  
\[  
\min \;\; (x – 1)^2 + 100(y – x^2)^2  
\]  
Arithmetic operators in pyomo are overloaded to generate pyomo expressions, Source:

source("operators.R")

Reticulate makes it easy to write code close to what it is in python (with certain modifications)

# Create model object

M = pyo$ConcreteModel()

# Define and initialize the variables\*

M$x = pyo$Var(initialize = 1.5)

M$y = pyo$Var(initialize = 1.5)

# Define the objective function

M$obj = pyo$Objective(expr = (1 - M$x)\*\*2 + 100 \* (M$y - M$x\*\*2)\*\*2)

We can check the model that has been created with the pprint command. While the R command M$pprint() works and shows output in the console, I couldn’t get it to show in the R markdown output. So I am using the python chunk to show the output.

r.M.pprint()

## 2 Var Declarations

## x : Size=1, Index=None

## Key : Lower : Value : Upper : Fixed : Stale : Domain

## None : None : 1.5 : None : False : False : Reals

## y : Size=1, Index=None

## Key : Lower : Value : Upper : Fixed : Stale : Domain

## None : None : 1.5 : None : False : False : Reals

##

## 1 Objective Declarations

## obj : Size=1, Index=None, Active=True

## Key : Active : Sense : Expression

## None : True : minimize : (1.0 - x)\*\*2.0 + 100.0\*(y - x\*\*2.0)\*\*2.0

##

## 3 Declarations: x y obj

Pyomo can interface with different open-source and commercial solvers depending on the type of optimization problem. Usually solvers have to be separately installed to be used with pyomo.

solver = pyo$SolverFactory("ipopt")

res = solver$solve(M, logfile = 'tmp.log')

Since the console output from solver doesn’t show in R markdown output, I wrote it to a temporary file and read it back again.

x = paste(readLines("tmp.log", encoding = "UTF-8"))

writeLines(x[2:63]) # lines 1 and 64 cause encoding issues in blog rendering

##

## Ipopt 3.12.12:

##

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## This program contains Ipopt, a library for large-scale nonlinear optimization.

## Ipopt is released as open source code under the Eclipse Public License (EPL).

##

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

##

## This is Ipopt version 3.12.12, running with linear solver mumps.

## NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

##

## Number of nonzeros in equality constraint Jacobian...: 0

## Number of nonzeros in inequality constraint Jacobian.: 0

## Number of nonzeros in Lagrangian Hessian.............: 3

##

## Total number of variables............................: 2

## variables with only lower bounds: 0

## variables with lower and upper bounds: 0

## variables with only upper bounds: 0

## Total number of equality constraints.................: 0

## Total number of inequality constraints...............: 0

## inequality constraints with only lower bounds: 0

## inequality constraints with lower and upper bounds: 0

## inequality constraints with only upper bounds: 0

##

## iter objective inf\_pr inf\_du lg(mu) ||d|| lg(rg) alpha\_du alpha\_pr ls

## 0 5.6500000e+01 0.00e+00 1.00e+02 -1.0 0.00e+00 - 0.00e+00 0.00e+00 0

## 1 2.4669972e-01 0.00e+00 2.22e-01 -1.0 7.40e-01 - 1.00e+00 1.00e+00f 1

## 2 1.6256267e-01 0.00e+00 2.04e+00 -1.7 1.48e+00 - 1.00e+00 2.50e-01f 3

## 3 8.6119444e-02 0.00e+00 1.08e+00 -1.7 2.36e-01 - 1.00e+00 1.00e+00f 1

## 4 4.3223836e-02 0.00e+00 1.23e+00 -1.7 2.61e-01 - 1.00e+00 1.00e+00f 1

## 5 1.5610508e-02 0.00e+00 3.54e-01 -1.7 1.18e-01 - 1.00e+00 1.00e+00f 1

## 6 5.3544798e-03 0.00e+00 5.51e-01 -1.7 1.67e-01 - 1.00e+00 1.00e+00f 1

## 7 6.1281576e-04 0.00e+00 5.19e-02 -1.7 3.87e-02 - 1.00e+00 1.00e+00f 1

## 8 2.8893076e-05 0.00e+00 4.52e-02 -2.5 4.53e-02 - 1.00e+00 1.00e+00f 1

## 9 3.4591761e-08 0.00e+00 3.80e-04 -2.5 3.18e-03 - 1.00e+00 1.00e+00f 1

## iter objective inf\_pr inf\_du lg(mu) ||d|| lg(rg) alpha\_du alpha\_pr ls

## 10 1.2680803e-13 0.00e+00 3.02e-06 -5.7 3.62e-04 - 1.00e+00 1.00e+00f 1

## 11 7.0136460e-25 0.00e+00 1.72e-12 -8.6 2.13e-07 - 1.00e+00 1.00e+00f 1

##

## Number of Iterations....: 11

##

## (scaled) (unscaled)

## Objective...............: 1.5551321399859192e-25 7.0136459513364959e-25

## Dual infeasibility......: 1.7239720368203862e-12 7.7751138860599418e-12

## Constraint violation....: 0.0000000000000000e+00 0.0000000000000000e+00

## Complementarity.........: 0.0000000000000000e+00 0.0000000000000000e+00

## Overall NLP error.......: 1.7239720368203862e-12 7.7751138860599418e-12

##

##

## Number of objective function evaluations = 18

## Number of objective gradient evaluations = 12

## Number of equality constraint evaluations = 0

## Number of inequality constraint evaluations = 0

## Number of equality constraint Jacobian evaluations = 0

## Number of inequality constraint Jacobian evaluations = 0

## Number of Lagrangian Hessian evaluations = 11

## Total CPU secs in IPOPT (w/o function evaluations) = 0.037

## Total CPU secs in NLP function evaluations = 0.000

##

## EXIT: Optimal Solution Found.

The solution of the model can be access using the defined variables of the model

# Display model solution

M$x(); M$y()

## 1.0000000000008233

## 1.0000000000016314

**Example 2: Transportation Problem (Linear Programming)**

Given a set of markets with demands and plants that can supply the demands, the goal in the transportation problem is to determine minimum cost shipping scenario that meets market demands while satisfying supply capacity of plants. The decision variables and parameters of the problem are:

* Decision Variable: \(x\_{ij}\) – quantity shipped from plant \(i\) to market \(j\)
* Parameters:
  + \(P\) – number of plants, \(M\) – number of markets
  + \(c\_{ij}\) – cost of shipping per unit from plant \(i\) to market \(j\)
  + \(Cap\_i\) – capacity of plant \(i\)
  + \(Dem\_j\) – demand in market \(j\)

The mathematical formulation of the problem is:  
\[  
\mbox{(Minimize Cost)}\;\;\min \;\; \sum\_{i=1}^P\sum\_{j=1}^Mc\_{ij}x\_{ij} \\  
\mbox{(Supply from a plant below capacity)} \;\;\sum\_{j=1}^Mx\_{ij} \leq Cap\_i, \;\; i=1,2,\ldots,P \\  
\mbox{(Supply to a market meets demand)}\;\;\sum\_{i=1}^Px\_{ij} \geq Dem\_j, \;\; j=1,2,\ldots,M \\  
\mbox{(shipments are non-negative)}\;\;x\_{ij}\geq 0, \;\; i=1,2,\ldots,P, \;\; j=1,2,\ldots,M  
\]

Cost is a dictionary where keys are tuples of the form (plant, market). I couldn’t figure out how to specify a R list that would translate to such a dictionary in python. So I used a round about way by converting a tibble to a pandas dataframe to a dictionary.

# shipping cost between plants and markets

costdf = tribble(

~plants, ~mkts, ~dtab,

"seattle", "new-york", 2.5,

"seattle", "chicago", 1.7,

"seattle", "topeka", 1.8,

"san-diego", "new-york", 2.5,

"san-diego", "chicago", 1.8,

"san-diego", "topeka", 1.4

)

costdf = costdf %>% mutate(cost = dtab \* 90)

dict\_from\_df = function(df) {

names(df) = c("x", "y", "z")

dfpy = r\_to\_py(df)$set\_index(c("x", "y"))

dfdict = dfpy$to\_dict()["z"]

return(dfdict)

}

cost = dict\_from\_df(costdf %>% select(plants, mkts, cost))

cost

## {('seattle', 'new-york'): 225.0, ('seattle', 'chicago'): 153.0, ('seattle', 'topeka'): 162.0, ('san-diego', 'new-york'): 225.0, ('san-diego', 'chicago'): 162.0, ('san-diego', 'topeka'): 125.99999999999999}

Next, we create a model object and define the parameters and decision variables

# create the model object

M = pyo$ConcreteModel()

# set the model parameters

M$plants = pyo$Set(initialize = c("seattle", "san-diego"))

M$mkts = pyo$Set(initialize = c("new-york", "chicago", "topeka"))

M$cap = pyo$Param(M$plants, initialize = list("seattle" = 350, "san-diego" = 600))

M$dem = pyo$Param(M$mkts, initialize = list("new-york" = 325, "chicago" = 300, "topeka" = 275))

M$cost = pyo$Param(M$plants, M$mkts, initialize = cost)

# define the model decision variables (shipment quantities between a plant and market)

# bounds specify that x>=0

M$x = pyo$Var(M$plants, M$mkts, bounds = tuple(0, NULL))

Next, we will set the constraint on supply from plants being below their capacities. In pyomo, the constraints can be described with a function that describes the rule to construct the constraint. In python pyomo, the supply constraint will be listed as follows:

def supply\_rule(m, i):

expr = sum(m.x[i, j] for j in mkts) <= cap[i]

return expr

M.supply\_cons = pyo.Constraint(plants, rule = supply\_rule)

The function supply\_rule is a function that tells what the constraint expression is for a particular plant \(i\). Then M.supply\_cons line ensures that the constraint is generated for each of the plants. Python shines here since this is represented succintly using list comprehensions. Since I am not sure how to do an equivalent representation using R, I used loops in the function to construct the constraint. I also wasn’t sure if using R function to represent the rule will work when passing the R function to create the supply constraint. But it worked fine and I guess that’s the magic of reticulate. Below is the supply constraint constructed in R.

# supply from plant <= plant capacity

supply\_rule = function(m, i) {

supply = 0

for (j in bi$list(m$mkts)) {

supply = supply + m$x[tuple(i, j)]

}

expr = supply <= m$cap[[i]]

return(expr)

}

M$supply\_cons = pyo$Constraint(M$plants, rule = supply\_rule)

We can check if the supply constraint is constructed correctly

r.M.supply\_cons.pprint()

## supply\_cons : Size=2, Index=plants, Active=True

## Key : Lower : Body : Upper : Active

## san-diego : -Inf : x[san-diego,new-york] + x[san-diego,chicago] + x[san-diego,topeka] : 600.0 : True

## seattle : -Inf : x[seattle,new-york] + x[seattle,chicago] + x[seattle,topeka] : 350.0 : True

Similarly, we can construct the demand constraint

# supply to a market meets its demand

demand\_rule = function(m, j) {

demand = 0

for (i in bi$list(m$plants)) {

demand = demand + m$x[tuple(i, j)]

}

expr = demand >= m$dem[[j]]

return(expr)

}

M$demand\_cons = pyo$Constraint(M$mkts, rule = demand\_rule)

r.M.demand\_cons.pprint()

## demand\_cons : Size=3, Index=mkts, Active=True

## Key : Lower : Body : Upper : Active

## chicago : 300.0 : x[seattle,chicago] + x[san-diego,chicago] : +Inf : True

## new-york : 325.0 : x[seattle,new-york] + x[san-diego,new-york] : +Inf : True

## topeka : 275.0 : x[seattle,topeka] + x[san-diego,topeka] : +Inf : True

Next, we construct the objective of total shipment cost and set it to be minimized.

# objective to minimize shipping cost

objective\_rule = function(m) {

totcost = 0

for (i in bi$list(m$plants)) {

for (j in bi$list(m$mkts)) {

totcost = totcost + m$cost[tuple(i, j)] \* m$x[tuple(i, j)]

}

}

return(totcost)

}

M$objective = pyo$Objective(rule = objective\_rule, sense = pyo$minimize)

Since this a linear program, we use the open-source solver [glpk](https://www.gnu.org/software/glpk/) to solve this.

# solve using solver glpk

opt = pyo$SolverFactory('glpk')

results = opt$solve(M, logfile = "tmp.log")

Optimal shipping cost based on solving this problem is:

r.M.objective()

## 153675.0

Next we display the shipment quantities. It is a bit more verbose since we worked with the option of convert = FALSE resulting in explicitly extracting desired values.

res\_L = list()

k = 1

for(i in bi$list(M$plants)) {

for(j in bi$list(M$mkts)) {

res\_L[[k]] = list(plant = i, mkt = j, qty = py\_to\_r(M$x[tuple(i, j)]()))

k = k + 1

}

}

res\_df = bind\_rows(res\_L)

res\_df

## # A tibble: 6 x 3

## plant mkt qty

##

## 1 seattle new-york 0

## 2 seattle chicago 300

## 3 seattle topeka 0

## 4 san-diego new-york 325

## 5 san-diego chicago 0

## 6 san-diego topeka 275

**Summary**

Here I covered two examples to show how to develop a pyomo model from R using the reticulate package. While it might still be easier to develop the pyomo model in python (since it was meant to be that way), I found that it is possible to develop pyomo models in R also fairly easily albeit with some modifications (some maybe less elegant compred to the python counterpart). It may still be better to develop more involved pyomo models in python but reticulate offers a way to develop simple to intermediate levels models directly in R. I am summarizing key learnings:

* Need to overload arithmetic operators to enable things like addition etc. between pyomo objects
* Use the option convert = FALSE to retain pyomo objects as python objects potentially avoid issues that are hard to troubleshoot.
* Lack of list comprehension in R makes some of the constraint specifications more verbose but still works.
* Need to be careful about indexing (sometimes need to explicitly specify a tuple and sometimes not)