Optimization problems in Python using Pyomo: An Introduction

Camilo Velasquez A

Overview

- Introduction
- Pyomo Modeling: Warehouse Problem
- Pyomo Modeling: Sudoku Problem

Definition of Optimization Problem

Components:

- **Objective/Goal:** What wants to be maximized, minimized, eg.
  Eg: Minimize costs, maximize income, Is it possible/feasible?
- **Constraints/Relations:** Set of constraints or requirements that must be satisfied
  Eg: Total machines are 10, Strictly positive price, etc
- **Decision Variables:** Set of variables or parameters that can be tuned to fulfil requirements while reaching the objective
  Eg: Price of rice, Number of machines, Indicator of use machine.

This is independent of how we solve it!

- It is Pythonic, Object Oriented and Open Source!
- Customize Capability: Easy to modularize components
- Solver Agnostic: Can use multiple open source or commercial solvers (AMPL, IPOPT, GLPK, ...)
- High level API.
- Extended documentation
- Can tackle advanced optimization problems (Mixed Integer, Discrete, Nonlinear, Stochastic, Disjunctive, etc)

Sample **Applications**

- ❖ Job Scheduling
- ❖ Logistic/Transportation
- ❖ Industrial Production
- ❖ Portfolio Optimization
- ❖ Resource Allocation
- ❖ Parameter Estimation
- ❖ Blending Problems
- ❖ Network designing
- ❖ Prices Design
- ❖ Much more applications!!

Pyomo Modeling: The Warehouse Problem

- Definition
- Formulation
- Pyomo approach

Definition

- We have to attend 5 countries from 4 possible warehouses locations or cities.
- We have a limit on how many warehouses we can build.
- We have to define how much demand fraction each warehouse is going to attend from each country. Ex: Bogota will attend 60% of Peru's demand
- We will assume delivery costs proportional to the demand fraction.
- We need to minimize the costs of delivering, meeting all the demand within the countries.

Adapted from: Pyomo. Optimization Modeling in Python.

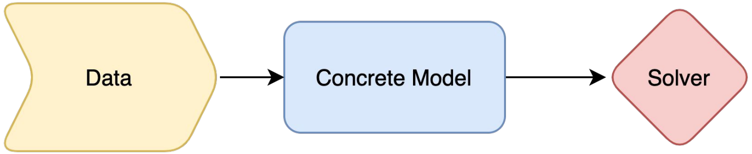
Formulation

Objective/Goal	Minimize the cost of fulfilling the demand
Constraints	Ensure that all demand is satisfied for each country Ensure that you have a limit of warehouses The demand attended from each warehouse is a fraction A warehouse is built or not (Binary) Ensure that you can only use the built warehouses
Decision Variables	Which warehouses are going to be used? How much fraction of demand must each warehouse deliver to each customer?

Pyomo Modeling Approaches

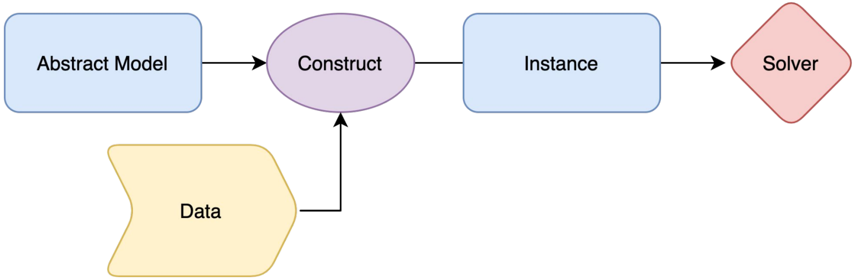
Concrete Model

$$
\begin{aligned}
\min \quad & 2x_1 + 3x_2 \\
\text{s.t.} \quad & 3x_1 + 4x_2 \geq 1 \\
& x_1, x_2 \geq 0
\end{aligned}
$$



Abstract Model

$$
\begin{aligned}
\min \quad & \textstyle\sum_{j=1}^{n} c_j x_j \\
\text{s.t.} \quad & \textstyle\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad && \forall i = 1 \ldots m \\
& x_j \geq 0 && \forall j = 1 \ldots n
\end{aligned}
$$



Concrete Model

$Cities$: Set of candidate warehouses/cities
$Countries$: Set of customer/countries
$Costs_{city,country}$: Cost of delivering product to **country** from warehouse **city**
y_{city} : 1 if **city** is selected to be warehouse. 0 otherwise.
$x_{city,country}$: Fraction of the demand for **country** served from warehouse **city**
$LimitWarehouses$: Limit of warehouses that can we can build

Candidate Warehouse Location	CiudadPanama	Bogota	Lima	RioJaneiro
Customers Country				
Panama	10	130	90	420
Colombia	100	50	110	340
Peru	200	150	20	330
Guatemala	70	180	160	450
Brazil	300	380	320	40

$Costs_{city,country}$

```
# Warehouse locations ([Bogota, Lima, ...])
cities: List[City] = costs_df.columns
# Customers ([Peru, Brazil, ...])
countries: List[Country] = costs_df.index
# Costs of delivering from City to Country
# {(Lima, Brazil): 30, (Lima, Colombia): 30, ...}
costs: Dict[Tuple[City, Country], Cost] = costs_df.unstack(0).to_dict()
# Limit of Warehouses
limit_warehouses: int = 2

# Defining ConcreteModel
model = ConcreteModel(name="Warehouse Example")
# x :~ Dict[Tuple[City, Country], DemandFraction]
model.x = Var(cities, countries, bounds=(0,1))
# y :~ Dict[City, IsBuilt]
model.y = Var(cities, within=Binary)
```

Concrete Model

Minimize Costs | $$\min_{x,y} \sum_{cit \in Cities} \sum_{cou \in Countries} Costs_{city,country} x_{city,country} \quad (1)$$

$$s.t.$$

Satisfy All Demand | $$\sum_{city \in Cities} x_{city,country} = 1, \forall country \in Countries \quad (2)$$

Only use built WH | $$x_{city,country} \leq y_{city}, \forall cit \in Cities, cou \in Countries \quad (3)$$

Warehouses Limit | $$\sum_{city \in Cities} y_{city} \leq LimitWarehouses \quad (4)$$

Demand as Fraction | $$0 \leq x_{city,country} \leq 1 \quad (5)$$

WH used or not | $$y_{city} \in \{0, 1\} \quad (6)$$

Solve the model using **GLPK** |

```
# Objective (1)
model.obj = Objective(
    expr=np.sum(
        [costs[city,country]*model.x[city,country] \
         for city in cities for country in countries]),
    sense=minimize, name="Minimize Cost")
# Constraint (2)
def one_per_customer_rule(model, country):
    return np.sum([model.x[city,country] for city in cities]) == 1
model.customers_complete_frac = Constraint(
    countries, rule=one_per_customer_rule,
    name="Constraint 2 - Countries demand fulfilled")
# Constraint (3)
def warehouse_active_rule(model, city, country):
    return model.x[city,country] <= model.y[city]
model.warehouse_active = Constraint(
    cities, countries, rule=warehouse_active_rule,
    name="Constraint 3 - Can only attend from built warehouses")
# Constraint (4)
model.warehouses_limit = Constraint(
    expr=np.sum(
      [model.y[city] for city in cities]) <= limit_warehouses,
    name="Constraint 4 - Warehouse Limit")
# Solve the model and report the results
solver = SolverFactory('glpk')
solver.solve(model)
```

Solution Display

Decision variables Outcome

```
Variables:
  x : Size=20, Index=x_index
      Key                          : Lower : Value : Upper : Fixed : Stale : Domain
            ('Bogota', 'Brazil') :     0 :   0.0 :     1 : False : False :  Reals
          ('Bogota', 'Colombia') :     0 :   0.0 :     1 : False : False :  Reals
         ('Bogota', 'Guatemala') :     0 :   0.0 :     1 : False : False :  Reals
            ('Bogota', 'Panama') :     0 :   0.0 :     1 : False : False :  Reals
              ('Bogota', 'Peru') :     0 :   0.0 :     1 : False : False :  Reals
       ('CiudadPanama', 'Brazil') :     0 :   0.0 :     1 : False : False :  Reals
     ('CiudadPanama', 'Colombia') :     0 :   1.0 :     1 : False : False :  Reals
    ('CiudadPanama', 'Guatemala') :     0 :   1.0 :     1 : False : False :  Reals
       ('CiudadPanama', 'Panama') :     0 :   1.0 :     1 : False : False :  Reals
         ('CiudadPanama', 'Peru') :     0 :   1.0 :     1 : False : False :  Reals
              ('Lima', 'Brazil') :     0 :   0.0 :     1 : False : False :  Reals
            ('Lima', 'Colombia') :     0 :   0.0 :     1 : False : False :  Reals
           ('Lima', 'Guatemala') :     0 :   0.0 :     1 : False : False :  Reals
              ('Lima', 'Panama') :     0 :   0.0 :     1 : False : False :  Reals
                ('Lima', 'Peru') :     0 :   0.0 :     1 : False : False :  Reals
         ('RioJaneiro', 'Brazil') :     0 :   1.0 :     1 : False : False :  Reals
       ('RioJaneiro', 'Colombia') :     0 :   0.0 :     1 : False : False :  Reals
      ('RioJaneiro', 'Guatemala') :     0 :   0.0 :     1 : False : False :  Reals
         ('RioJaneiro', 'Panama') :     0 :   0.0 :     1 : False : False :  Reals
           ('RioJaneiro', 'Peru') :     0 :   0.0 :     1 : False : False :  Reals
  y : Size=4, Index=y_index
      Key          : Lower : Value : Upper : Fixed : Stale : Domain
            Bogota :     0 :   0.0 :     1 : False : False : Binary
      CiudadPanama :     0 :   1.0 :     1 : False : False : Binary
              Lima :     0 :   0.0 :     1 : False : False : Binary
        RioJaneiro :     0 :   1.0 :     1 : False : False : Binary
```

Candidate Warehouse Location	CiudadPanama	Bogota	Lima	RioJaneiro
Customers Country				
Panama	10	130	90	420
Colombia	100	50	110	340
Peru	200	150	20	330
Guatemala	70	180	160	450
Brazil	300	380	320	40

Total Cost: 420

```
Objectives:
  obj : Size=1, Index=None, Active=True
      Key  : Active : Value
      None :   True : 420.0
```

Abstract Model

Concrete Model

```python
# Warehouse locations ([Bogota, Lima, ...])
cities: List[City] = costs_df.columns
# Customers ([Peru, Brazil, ...])
countries: List[Country] = costs_df.index
# Costs of delivering from City to Country
# {(Lima, Brazil): 30, (Lima, Colombia): 30, ...}
costs: Dict[Tuple[City, Country], Cost] = costs_df.unstack(0).to_dict()
# Limit of Warehouses
limit_warehouses: int = 2

# Defining ConcreteModel
model = ConcreteModel(name="Warehouse Example")
# x :~ Dict[Tuple[City, Country], DemandFraction]
model.x = Var(cities, countries, bounds=(0,1))
# y :~ Dict[City, IsBuilt]
model.y = Var(cities, within=Binary)
```

Abstract Model

```python
model = AbstractModel(name="Warehouse Example Abstract")
# Set: Pyomo Model component to express List or Indexables
model.cities = Set()
model.countries = Set()
model.costs = Param(model.cities, model.countries)
model.limit_warehouses = Param()

model.x = Var(model.cities, model.countries, bounds=(0,1))
model.y = Var(model.cities, within=Binary)
```

Abstract Model: Instantiate with Python

```python
# Warehouse locations ([Bogota, Lima, ...])
cities: List[City] = costs_df.columns
# Customers ([Peru, Brazil, ...])
countries: List[Country] = costs_df.index
# Costs of delivering from City to Country
# {(Lima, Brazil): 30, (Lima, Colombia): 30, ...}
costs: Dict[Tuple[City, Country], Cost] = costs_df.unstack(0).to_dict()

# Data definition
data={
        "namespace1": {
            "cities": {None: cities},
            "countries": {None: countries},
            "costs": costs,
            "limit_warehouses": {None: 2}
        },
        "namespace2": {
            "cities": {None: cities},
            "countries": {None: countries},
            "costs": costs,
            "limit_warehouses": {None: 3}
        },
    }
```

```python
instance = model.create_instance(namespace="namespace1",
                                 data=data)
# With the other namespace
instance2 = model.create_instance(namespace="namespace2",
                                  data=data)
solver = SolverFactory('cbc')
solver.solve(instance)
```

Example:
Sudoku Solver

- Definition
- Formulation
- Math Modeling
- Pyomo approach

Definition

						2		
	8				7		9	
6		2				5		
	7			6				
			9		1			
				2			4	
		5				6		3
	9		4				7	
		6						

(a) Sudoku Puzzle

9	5	7	6	1	3	2	8	4
4	8	3	2	5	7	1	9	6
6	1	2	8	4	9	5	3	7
1	7	8	3	6	4	9	5	2
5	2	4	9	7	1	3	6	8
3	6	9	5	2	8	7	4	1
8	4	5	7	9	2	6	1	3
2	9	1	4	3	6	8	7	5
7	3	6	1	8	5	4	2	9

(b) Solution

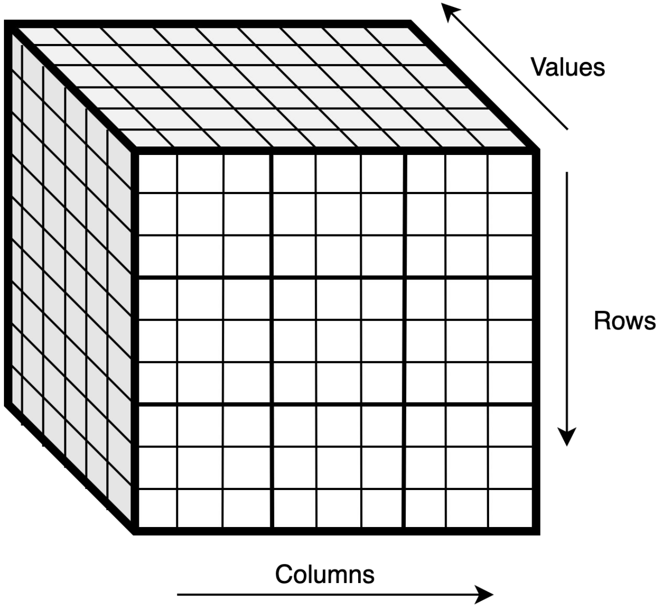
Get whether a Sudoku is solvable, and get all feasible solutions.

Taken from: Pyomo. Optimization Modeling in Python.

Formulation

Objective/Goal	Anything would work, we just need to fulfill the constraints
Constraints	All values should be unique within a row All values should be unique within a column All values should be unique within a SubSquare (3x3 square) A cell can only contain one value
Decision Variables	Which values are used in each cell?

Modeling



A Cube of Binary (Indicator) variables.

$y[r, c, v] = 1$ if v is the value in r, c. 0 otherwise

Constraints

$$\sum_{c \in Cols} y[r, c, v] = 1, \forall r \in Rows, \forall v \in Values \quad (1)$$

$$\sum_{r \in Rows} y[r, c, v] = 1, \forall c \in Cols, \forall v \in Values \quad (2)$$

$$\sum_{r,c \in SubSquares[i]} y[r, c, v] = 1, \forall i \in SubSquares \quad (3)$$

$$\sum_{v \in Values} y[r, c, v] = 1, \forall r \in Rows, \forall c \in Cols \quad (4)$$

Abstract Model

```python
model = pyo.AbstractModel()
# Starting board as internal variable
model.board = pyo.Set()
# Create sets for constraints
model.rows = pyo.RangeSet(1,9)
model.cols = pyo.RangeSet(1,9)
model.subsquares = pyo.RangeSet(1,9)
model.values_ = pyo.RangeSet(1,9)
# Create Y as Binary: row, col, val
model.y = pyo.Var(model.rows, model.cols, model.values_,
                  within=pyo.Binary)
```

```python
# This is a feasability problem (Objective doesn't matter which)
model.obj = pyo.Objective(expr=1.0)

def row_constraint(model, r, v):
    return sum(model.y[r,c,v] for c in model.cols) == 1
model.row_constraint = pyo.Constraint(
    model.rows, model.values_,
    rule=row_constraint,
    name="Constraint 1 - Unique vals per rows")

def col_constraint(model, c, v):
    return sum(model.y[r,c,v] for r in model.rows) == 1
model.col_constraint = pyo.Constraint(
    model.cols, model.values_,
    rule=col_constraint,
    name="Constraint 2 - Unique vals per Columns")

def subsquare_constraint(subsq_to_row_col):
    def _sq_constraint(model, s, v):
        return sum(model.y[r,c,v] for (r, c) in subsq_to_row_col[s]) == 1
    return _sq_constraint
model.subsq_constraint = pyo.Constraint(
    model.subsquares, model.values_,
    rule=subsquare_constraint(subsq_to_row_col),
    name="Constraint 3 - Unique vals per Subsquare")

def value_constraint(model, r, c):
    return sum(model.y[r,c,v] for v in model.values_) == 1
model.value_constraint = pyo.Constraint(
    model.rows, model.cols, rule=value_constraint,
    name="Constraint 4 - Only one value per Cell")
```

Abstract Model: Sudoku Board

Example Board

```
board: List[Tuple[Row, Column, Value]] = [
    (5, 1, 5),
    (6, 1, 8),
    (7, 1, 9),
    (8, 1, 2),
    (2, 2, 1),
    (3, 2, 3),
    ...
    (3, 9, 5),
    (4, 9, 4),
    (5, 9, 1)]
```

Column

Row	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	4	0	0
2	0	1	5	0	6	0	0	0	9
3	0	3	8	0	4	9	0	6	5
4	0	0	2	0	9	0	0	0	4
5	5	0	0	0	0	0	0	0	1
6	8	0	0	0	2	0	9	0	0
7	9	6	0	8	3	0	1	7	0
8	2	0	0	0	1	0	5	9	0
9	0	0	3	0	0	2	0	4	0

Additional Resources

```python
# Fix initial board values
def build_model(model):
    # Fix variables based on the current board
    for (r,c,v) in model.board:
        model.y[r,c,v].fix(1)

# Remove previously seen solutions
def add_integer_cut(model):
    if not hasattr(model, "integer_cuts"):
        model.integer_cuts = pyo.ConstraintList()
    # Add the integer cut, corresponding to the current solution in the model
    # Note that if it is exactly one of the previous solution it would not satisfy the constraint
    # To satisfy the constraint, at least 1 number should be different
    cut_expr = 0.0
    for r in model.rows:
        for c in model.cols:
            for v in model.values_:
                if not model.y[r,c,v].fixed:
                    # Note, it may not be exactly 1 (Precision error)
                    if model.y[r,c,v].value >= 0.5:
                        cut_expr += (1.0 - model.y[r,c,v])
                    else:
                        cut_expr += model.y[r,c,v]
    model.integer_cuts.add(cut_expr >= 1)
```

Define 2 Sets: S_0 and S_1 :

S_0 : Indices for those variables whose current solution is 0.

S_1 : Indices for those variables whose current solution is 1.

$$\sum_{r,c,v \in S_0} y[r,c,v] + \sum_{r,c,v \in S_1} (1 - y[r,c,v]) \geq 1 \quad (5)$$

Solving a Sudoku

```
instance = model.create_instance(namespace="sudoku3", data=data)
build_model(instance)
solutions = []
while True:
    with pyo.SolverFactory("glpk") as opt:
        results = opt.solve(instance)
        if results.solver.termination_condition != pyo.TerminationCondition.optimal:
            print("All board solutions have been found")
            break
    add_integer_cut(instance)
    solutions.append(instance.clone())
print(f"Number of solutions: {len(solutions)}")
------------------------------------------------------------------------------------
>>> All board solutions have been found
>>> Number of solutions: 2
```

Solutions

0	0	0	0	0	0	4	0	0
0	1	5	0	6	0	0	0	9
0	3	8	0	4	9	0	6	5
0	0	2	0	9	0	0	0	4
5	0	0	0	0	0	0	0	1
8	0	0	0	2	0	9	0	0
9	6	0	8	3	0	1	7	0
2	0	0	0	1	0	5	9	0
0	0	3	0	0	2	0	4	0

6	2	9	3	5	7	4	1	8
4	1	5	2	6	8	7	3	9
7	3	8	1	4	9	2	6	5
3	7	2	5	9	1	6	8	4
5	9	6	7	8	4	3	2	1
8	4	1	6	2	3	9	5	7
9	6	4	8	3	5	1	7	2
2	8	7	4	1	6	5	9	3
1	5	3	9	7	2	8	4	6

6	2	9	7	5	3	4	1	8
4	1	5	2	6	8	7	3	9
7	3	8	1	4	9	2	6	5
3	7	2	5	9	1	6	8	4
5	9	6	4	8	7	3	2	1
8	4	1	3	2	6	9	5	7
9	6	4	8	3	5	1	7	2
2	8	7	6	1	4	5	9	3
1	5	3	9	7	2	8	4	6

Summary

- **Definition of optimization problems:** Objectives, Constraints, Variables
- **Pyomo Overview:** Open Source, Solver Agnostic, Nice docs, Tackle Multiple Optimization problems.
- **Pyomo Examples:**
    - Define multiple optimization problems (MILP, MINLP, etc)
    - Use different solvers
    - Display the solutions
- **Pyomo has much more!!**
    - Non-linear, Generalized disjunctive, MINLP, Quadratic Programming, Bilevel programming, Differential Algebraic Equations, etc
    - Sensitivity Analysis, Block Modeling, ...

THANKS!

Repo:

https://github.com/cvelas31/pyomo_examples

Contact:

linkedin.com/in/cvelas31

Factored

WE ARE HIRING!

References:

- http://www.pyomo.org/documentation
- https://pyomo.readthedocs.io/en/stable/index.html#
- osti.gov/servlets/purl/1110661
- https://github.com/Pyomo/pyomo
- Hart, William E., Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. *Pyomo – Optimization Modeling in Python*. Second Edition. Vol. 67. Springer, 2017.
- https://github.com/jckantor/ND-Pyomo-Cookbook

Code and presentation is located in the this repo:

- https://github.com/cvelas31/pyomo_examples

Feel free to use it, add things, etc.

Pyomo Github Commit Contributions