Sandia
National
Laboratories

Exceptional

service

in the

national

interest

3. Pyomo Fundamentals

PYOMO

U.S. DEPARTMENT OF ENERGY

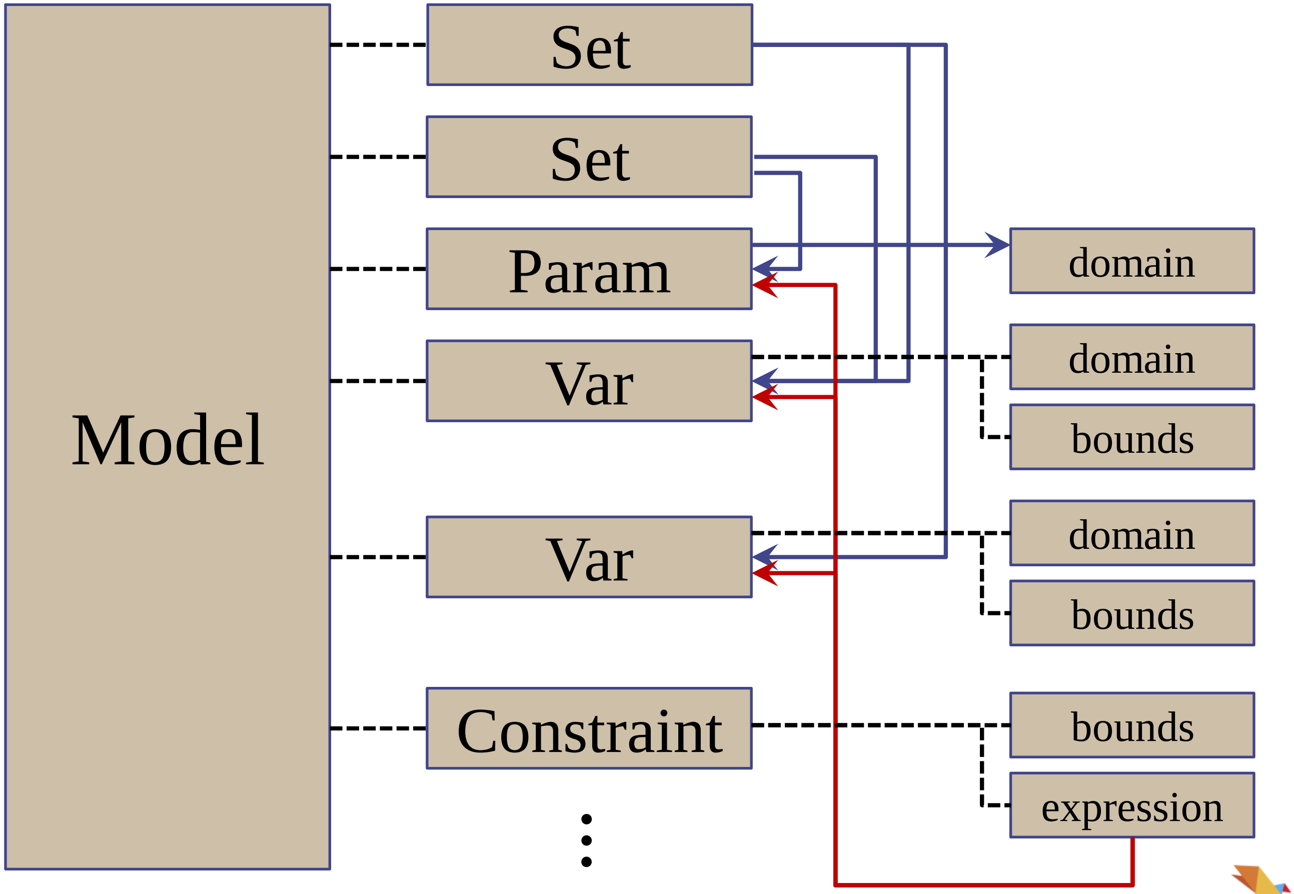
NNSA
National Nuclear Security Administration

CCR
Center for Computing Research

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

3. Fundamental Pyomo Components

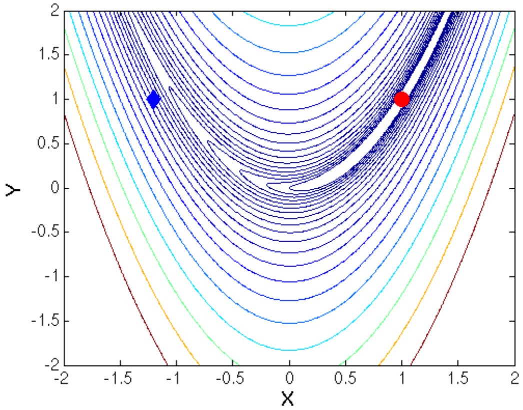
- Pyomo is an *object model* for describing optimization problems
  - The fundamental objects used to build models are *Components*



Cutting to the chase: a simple Pyomo model

- `rosenbrock.py`:

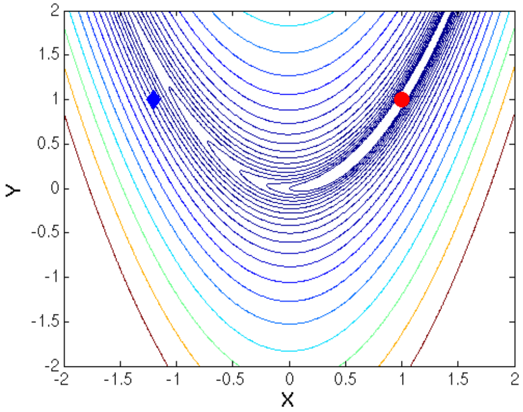
```
from pyomo.environ import *

model = ConcreteModel()

model.x = Var( initialize=-1.2, bounds=(-2, 2) )
model.y = Var( initialize= 1.0, bounds=(-2, 2) )

model.obj = Objective(
    expr= (1-model.x)**2 + 100*(model.y-model.x**2)**2,
    sense= minimize )
```



Cutting to the chase: a simple Pyomo model

- Solve the model:
  - The pyomo command

```
% pyomo solve rosenbrock.py --solver=ipopt --summary
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.00] Creating model
[    0.00] Applying solver
[    0.03] Processing results
    Number of solutions: 1
    Solution Information
      Gap: <undefined>
      Status: optimal
      Function Value: 2.98956421871e-17
    Solver results file: results.json

=====================================================
Solution Summary
=====================================================

Model unknown

  Variables:
    Variable x : Size=1 Domain=Reals
      Value=0.999999994543
    Variable y : Size=1 Domain=Reals
      Value=0.999999989052

  Objectives:
    Objective obj : Size=1
      Value=2.98956421871e-17

  Constraints:
    None

[    0.03] Applying Pyomo postprocessing actions
[    0.03] Pyomo Finished
```



Regarding *namespaces*

- Pyomo objects exist within the `pyomo.environ` namespace:

```
import pyomo.environ
model = pyomo.environ.ConcreteModel()
```

- …but this gets verbose. To save typing, we will import the core Pyomo classes into the main namespace:

```
from pyomo.environ import *
model = ConcreteModel()
```

- To clarify Pyomo-specific syntax in this tutorial, we will highlight Pyomo symbols in green

Getting Started: the *Model*

```
from pyomo.environ import *

model = ConcreteModel()
```

Every Pyomo model starts with this; it tells Python to load the Pyomo Modeling Environment

Create an instance of a *Concrete* model

- Concrete models are immediately constructed
- Data must be present *at the time* components are defined

Local variable to hold the model we are about to construct

- While not required, by convention we use "`model`"
- If you choose to name your model something else, you will need to tell the Pyomo script the object name through the command line

Populating the Model: *Variables*

```
model.a_variable = Var(within = NonNegativeReals)
```

The name you assign the object to becomes the object’s name, and must be unique in any given model.

“`within`” is optional and sets the variable domain (“`domain`” is an alias for “`within`”)

Several pre-defined domains, e.g., “`Binary`”

```
model.a_variable = Var(bounds = (0, None))
```

Same as above: “`domain`” is assumed to be `Reals` if missing

Defining the *Objective*

```
model.x = Var( initialize=-1.2, bounds=(-2, 2) )
model.y = Var( initialize= 1.0, bounds=(-2, 2) )

model.obj = Objective(
    expr= (1-model.x)**2 + 100*(model.y-model.x**2)**2,
    sense= minimize )
```

If “sense” is omitted, Pyomo assumes minimization

Note that the Objective expression is not a *relational expression*

“expr” can be an expression, or any function-like object that returns an expression

Defining the Problem: *Constraints*

```
model.a = Var()
model.b = Var()
model.c = Var()
model.c1 = Constraint(
    expr = model.b + 5 * model.c <= model.a )
```

"expr" can be an expression, or any function-like object that returns an expression

```
model.c2 = Constraint(expr = (None, model.a + model.b, 1))
```

"expr" can also be a tuple:

- 3-tuple specifies (LB, expr, UB)
- 2-tuple specifies an equality constraint.

In general, we do not recommend this notation

Lists of Constraints

```
model.a = Var()
model.b = Var()
model.c = Var()
model.limits = ConstraintList()

model.limits.add(30*model.a + 15*model.b + 10*model.c <= 100)
model.limits.add(10*model.a + 25*model.b +  5*model.c <= 75)
model.limits.add(6*model.a  + 11*model.b +  3*model.c <= 30)
```

"add" adds a single new constraint to the list.
The constraints need not be related.

Higher-dimensional components

- (Almost) All Pyomo *components* can be *indexed*
  - *All* non-keyword arguments are assumed to be *indices*
  - Individual indices may be multi-dimensional (e.g., a list of pairs)

```
<Type>(<IDX1>, <IDX2>, […] <keyword>=<value>, …)
```

- Indexed variables

```
model.a_vector = Var(IDX)
model.a_matrix = Var(IDX_A, IDX_B)
```

The indexes are any iteratable object, e.g., list or Set

- ConstraintList is a special case with an implicit index

- *Note:* while indexed variables look like matrices, they are not.
  - In particular, we do not support matrix algebra (*yet…)*

Manipulating indices: list comprehensions

```
model.IDX = range(10)
model.a = Var()
model.b = Var(model.IDX)
model.c1 = Constraint(
    expr = sum(model.b[i] for i in model.IDX) <= model.a )
```

Python *list comprehensions* are very common for working over indexed variables and nicely parallel mathematical notation:

$$\sum_{i \in IDX} b_i \leq a$$

Concrete Modeling

Putting It All Together: *Concrete p-Median*

- Determine the set of P warehouses chosen from N candidates that minimizes the total cost of serving all customers M where $d_{n,m}$ is the cost of serving customer m from warehouse location n.

$$\begin{array}{llll}
\min & \sum_{n\in N,m\in M} d_{n,m}x_{n,m} & & \text{(minimize total cost)} \\
s.t. & \sum_{n} x_{n,m} = 1 & \forall\ m \in M & \text{(guarantee all customers served)} \\
 & x_{n,m} \le y_n & \forall\ n \in N, m \in M & \text{(customer n can only be served from warehouse m if warehouse m is selected)} \\
 & \sum_{n\in N} y_n = P & & \text{(select P warehouses)} \\
 & 0 \le x \le 1 & y \in \{0,1\}^N &
\end{array}$$

Concrete p-Median (1)

```
from pyomo.environ import *

N = 3
M = 4
P = 3
d = {(1, 1): 1.7, (1, 2): 7.2, (1, 3): 9.0, (1, 4): 8.3,
     (2, 1): 2.9, (2, 2): 6.3, (2, 3): 9.8, (2, 4): 0.7,
     (3, 1): 4.5, (3, 2): 4.8, (3, 3): 4.2, (3, 4): 9.3}

model = ConcreteModel()
model.Locations = range(N)
model.Customers = range(M)

model.x = Var( model.Locations, model.Customers,
               bounds=(0.0,1.0) )

model.y = Var( model.Locations, within=Binary )
```

Concrete p-Median (2)

```
model.obj = Objective( expr = sum( d[n,m]*model.x[n,m]
    for n in model.Locations for m in model.Customers ) )

model.single_x = ConstraintList()
for m in model.Customers:
    model.single_x.add(
        sum( model.x[n,m] for n in model.Locations ) == 1.0 )

model.bound_y = ConstraintList()
for n in model.Locations:
    for m in model.Customers:
        model.bound_y.add( model.x[n,m] <= model.y[n] )

model.num_facilities = Constraint(
    expr=sum( model.y[n] for n in model.Locations ) == P )
```

Solving models: *the* pyomo *command*

- pyomo (`pyomo.exe` on Windows):
  - Constructs model and passes it to an (external) solver

```
pyomo solve <model_file> [<data_file> …] [options]
```

  - Installed to:
    - `[PYTHONHOME]\Scripts` [Windows; `C:\Python27\Scripts`]
    - `[PYTHONHOME]/bin` [Linux; `/usr/bin`]

- Key options (*many* others; see `--help`)

`--help`	Get list of all options
`--help-solvers`	Get the list of all recognized solvers
`--solver=<solver_name>`	Set the solver that Pyomo will invoke
`--solver-options="key=value[ …]"`	Specify options to pass to the solver as a space-separated list of keyword-value pairs
`--stream-solver`	Display the solver output during the solve
`--summary`	Display a summary of the optimization result
`--report-timing`	Report additional timing information, including construction time for each model component

In Class Exercise: Concrete Knapsack

$$\max \quad \sum_{i=1}^{N} v_i x_i$$

$$s.t. \quad \sum_{i=1}^{N} w_i x_i \leq W_{\max}$$

$$x_i \in \{0,1\}$$

Item	**Weight**	**Value**
`hammer`	5	8
`wrench`	7	3
`screwdriver`	4	6
`towel`	3	11

`Max weight:` 14

Syntax reminders:

```
from pyomo.environ import *
ConcreteModel()
Var( [index, …], [within=domain], [bounds=(lower,upper)] )
ConstraintList()
    c.add( expression )
Objective( sense={maximize|minimize},
           expr=expression )
```

Concrete Knapsack: *Solution*

```
from pyomo.environ import *

v = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
W_max = 14

model = ConcreteModel()
model.ITEMS = v.keys()
model.x = Var( model.ITEMS, within=Binary )

model.value = Objective(
   expr = sum( v[i]*model.x[i] for i in model.ITEMS ),
   sense = maximize )

model.weight = Constraint(
   expr = sum( w[i]*model.x[i] for i in model.ITEMS ) <= W_max )
```

Abstract Modeling

Concrete *vs.* Abstract Models

- Concrete Models: data first, then model
  - 1-pass construction
  - All data must be present before Python starts processing the model
  - Pyomo will construct each component in order at the time it is declared
  - Straightforward logical process; easy to script.
  - Familiar to modelers with experience with GAMS

- Abstract Models: model first, then data
  - 2-pass construction
  - Pyomo stores the basic model declarations, but does not construct the actual objects
    - Details on how to construct the component hidden in functions, or *rules*
    - e.g., it will declare an indexed variable “x”, but will not expand the indices or populate any of the individual variable values.
  - At “creation time”, data is applied to the abstract declaration to create a concrete instance (components are still constructed in declaration order)
  - Encourages generic modeling and model reuse
    - e.g., model can be used for arbitrary-sized inputs
  - Familiar to modelers with experience with AMPL

Generating and Managing Indices: *Sets*

- Any iterable object can be an index, e.g., lists:
  - `IDX_a = [1,2,5]`
  - `DATA = {1: 10, 2: 21, 5:42};`
    `IDX_b = DATA.keys()`

- Sets: objects for managing multidimensional indices
  - `model.IDX = Set( initialize = [1,2,5] )`

Note: capitalization matters:
  `Set` = Pyomo class
  `set` = native Python set

Like indices, Sets can be initialized from any iterable

  - `model.IDX = Set( [1,2,5] )`

Note: This doesn't do what you want.
  This creates a 3-member *indexed set*, where each set is *empty.*

Sequential Indices: *RangeSet*

- Sets of sequential integers are common
  - `model.IDX = Set( initialize=range(5) )`
  - `model.IDX = RangeSet( 5 )`

Note: `RangeSet` is 1-based.
This gives [1, 2, 3, 4, 5]

Note: Python `range` is 0-based.
This gives [0, 1, 2, 3, 4]

- You can provide lower and upper bounds to RangeSet
  - `model.IDX = RangeSet( 0, 4 )`

This gives [0, 1, 2, 3, 4]

Manipulating Sets

- Sets support efficient higher-dimensional indices

```
model.IDX = Set( initialize=[1,2,5] )
model.IDX2 = model.IDX * model.IDX
```

This creates a *virtual* 2-D “matrix” Set

Sets also support union (&), intersection (|), difference (-), symmetric difference (^)

- Creating sparse sets

```
model.IDX = Set( initialize=[1,2,5] )
def lower_tri_filter(model, i, j):
      return j <= i
model.LTRI = Set( initialize = model.IDX * model.IDX,
                  filter = lower_tri_filter )
```

The `filter` should return *True* if the element is in the set; *False* otherwise.

Deferred construction: *Rules*

- Abstract modeling constructs the model in two passes:
  - Python parses the model declaration
    - creating “empty” Pyomo components in the model
  - Pyomo loads and parses external data

- Components are constructed in declaration order
  - The instructions for *how* to construct the object are provided through a function, or *rule*
  - Pyomo calls the rule for each component index
  - *Rules* can be provided to virtually all Pyomo components (even when using Concrete models)

- Naming conventions
  - the component name prepended with “`_`” (`c4` → `_c4`)
  - the component name with “`_rule`” appended (`c4` → `c4_rule`)
  - each rule is called “`rule`” (Python implicitly overrides each declaration)

Indexed Constraints

```
model.IDX = Set( initialize=range(5) )
model.a = Var( model.IDX )
model.b = Var()
def c4_rule(model, i):
     return model.a[i] + model.b <= 1
model.c4 = Constraint( model.IDX, rule=c4_rule )
```

For indexed constraints, you provide a "`rule`" (function) that returns an expression (or tuple) for each index.

```
model.IDX2 = model.IDX * model.IDX
def c5_rule(model, i, j, k):
     return model.a[i] + model.a[j] + model.a[k] <= 1
model.c5 = Constraint( model.IDX2, model.IDX, rule=c5_rule )
```

Each dimension of each index is a separate argument to the rule

Importing Data: *Parameters*

- Scalar numeric values

```
model.a_parameter = Param( initialize = 42 )
```

Provide an (initial) value of 42 for the parameter

- Indexed numeric values

```
model.a_param_vec = Param( IDX,
                           initialize = data,
                           default = 0 )
```

Providing "`default`" allows the initialization data to only specify the "unusual" values

"`data`" *must* be a dictionary(*) of index keys to values because all sets are assumed to be *unordered*

(*) – actually, it must define __getitem__(), but that only really matters to Python geeks

Data Sources

- Data can be imported from “`.dat`” file
  - Format similar to AMPL style
  - Explicit data from “`param`” declarations
  - External data through “`load`” declarations:
    - Excel

```
load ABCD.xls range=ABCD : Z=[A, B, C] Y=D ;
```

    - Databases

```
load “DBQ=diet.mdb” using=pyodbc query=“SELECT FOOD, cost,
  f_min, f_max from Food” : [FOOD] cost f_min f_max ;
```

- External data overrides “`initialize=`” declarations

Abstract p-Median (pmedian.py, 1)

```
from pyomo.environ import *

model = AbstractModel()


model.N = Param( within=PositiveIntegers )
model.P = Param( within=RangeSet( model.N ) )
model.M = Param( within=PositiveIntegers )


model.Locations = RangeSet( model.N )
model.Customers = RangeSet( model.M )


model.d = Param( model.Locations, model.Customers )


model.x = Var( model.Locations, model.Customers, bounds=(0.0, 1.0) )
model.y = Var( model.Locations, within=Binary )
```

Abstract p-Median (pmedian.py, 2)

```
def obj_rule(model):
    return sum( model.d[n,m]*model.x[n,m]
       for n in model.Locations for m in model.Customers )
model.obj = Objective( rule=obj_rule )

def single_x_rule(model, m):
    return sum( model.x[n,m] for n in model.Locations ) == 1.0
model.single_x = Constraint( model.Customers, rule=single_x_rule )

def bound_y_rule(model, n,m):
    return model.x[n,m] - model.y[n] <= 0.0
model.bound_y = Constraint( model.Locations, model.Customers,
                            rule=bound_y_rule )

def num_facilities_rule(model):
    return sum( model.y[n] for n in model.Locations ) == model.P
model.num_facilities = Constraint( rule=num_facilities_rule )
```

Abstract p-Median (pmedian.dat)

```
param N := 3;

param M := 4;

param P := 2;

param d:  1     2     3     4 :=
     1   1.7   7.2   9.0   8.3
     2   2.9   6.3   9.8   0.7
     3   4.5   4.8   4.2   9.3 ;
```

In Class Exercise: Abstract Knapsack

$$\max \quad \sum_{i=1}^{N} v_i x_i$$

$$s.t. \quad \sum_{i=1}^{N} w_i x_i \le W_{\max}$$

$$x_i \in \{0,1\}$$

Item	Weight	Value
`hammer`	5	8
`wrench`	7	3
`screwdriver`	4	6
`towel`	3	11

`Max weight:` 14

Syntax reminders:

```
AbstractModel()
Set( [index, …], [initialize=list/function] )
Param( [index, …], [within=domain], [initialize=dict/function] )
Var( [index, …], [within=domain], [bounds=(lower,upper)] )
Constraint( [index, …], [expr=expression|rule=function] )
Objective( sense={maximize|minimize},
           expr=expression|rule=function )
```

Abstract Knapsack: *Solution*

```
from pyomo.environ import *

model       = AbstractModel()
model.ITEMS = Set()
model.v     = Param( model.ITEMS, within=PositiveReals )
model.w     = Param( model.ITEMS, within=PositiveReals )
model.W_max = Param( within=PositiveReals )
model.x     = Var( model.ITEMS, within=Binary )

def value_rule(model):
    return sum( model.v[i]*model.x[i] for i in model.ITEMS )
model.value = Objective( rule=value_rule, sense=maximize )

def weight_rule(model):
    return sum( model.w[i]*model.x[i] for i in model.ITEMS ) \
        <= model.W_max
model.weight = Constraint( rule=weight_rule )
```

Abstract Knapsack: *Solution Data*

```
set ITEMS := hammer wrench screwdriver towel ;

param:  v w :=
    hammer       8 5
    wrench       3 7
    screwdriver  6 4
    towel       11 3;

param W_max := 14;
```