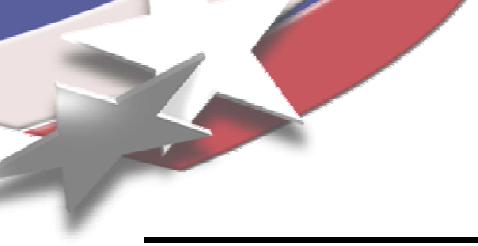


COOPR

SAND2013-4965C

Using Python and the Algebraic Modeling Language Pyomo to Specify, Solve, and Analyze Mathematical Programs

William E. Hart, John D. Siirola, and Jean-Paul Watson

Discrete Math and Complex Systems Department

Sandia National Laboratories

Albuquerque, NM USA

David Woodruff

Graduate School of Management

University of California Davis, USA

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.

NNSA National Nuclear Security Administration

Sandia National Laboratories

Rough Agenda

1. Introduction to Coopr / Pyomo
2. Introduction to Python
3. Fundamental Pyomo Components
4. Concrete Modeling
5. Abstract Modeling
6. Pyomo (and Python) Idioms and Efficiency
7. Scripting the optimization process
8. Hierarchical (block-oriented) Models
9. Disjunctive Programming: Coopr.GDP
10. Differential-Algebraic Models: Coopr.DAE
11. Stochastic Programming: Coopr.PySP
12. Reusing blocks: model libraries
13. Critiquing Coopr

0. Do You Have Pyomo Installed and Working?

- An interlude
  - But an important pre-requisite for what follows!

- If you don't:
  - Download & Install Python 2.7 (2.6 is also acceptable)
    - http://python.org/
  - Download & install Coopr
    - http://software.sandia.gov/trac/coopr/downloader
    - Linux / MacOS: coopr_install / coopr_votd
    - Windows: Coopr_*_setup.exe
  - Download & install "a solver":
    - glpk, cbc, ipopt are good starting points

1. Introduction to Pyomo

- Three really good questions:
  - *Why another Algebraic Modeling Language (AML)?*
  - *Why Python?*
  - *Why open-source?*

- Coopr: Software library infrastructure

- Coopr: Team overview and collaborators / users

- Where to find more information…

Why another AML: *Improve our productivity*

- Our initial objective was *not* to write another AML
  - However it ended up being a necessary prerequisite

- What we needed from a modeling environment:
  - Support rapid algorithm prototyping, development, and extension
  - Extensible to new modeling constructs (e.g., SP, GDP, DAE)
  - Facilitate hybrid approaches:
    - Hybrid models
    - Hybrid algorithms
    - Heuristic meta-algorithms
  - Transparent and accessible internal data structures
  - Interoperability (models, solvers, platforms, data sources)
  - Easily transferrable to non-expert user communities

Why Python: *it meets our needs*

- Python provides a full-featured object-oriented environment
  - Classes, inheritance, namespaces, exceptions, …
  - Interactive interpreter
- Python facilitates rapid prototyping and doesn't require a CS degree
  - Important for modelers and general productivity
- Python ships with a huge number of very useful libraries, including
  - Serialization, distributed computation, db/Excel interfaces, …
  - SciPy and NumPy
- Python has excellent support for dynamic loading
  - Critical for integrating 3rd-party extensions, custom user code
- Python introspection facilitates the development of generic algorithms

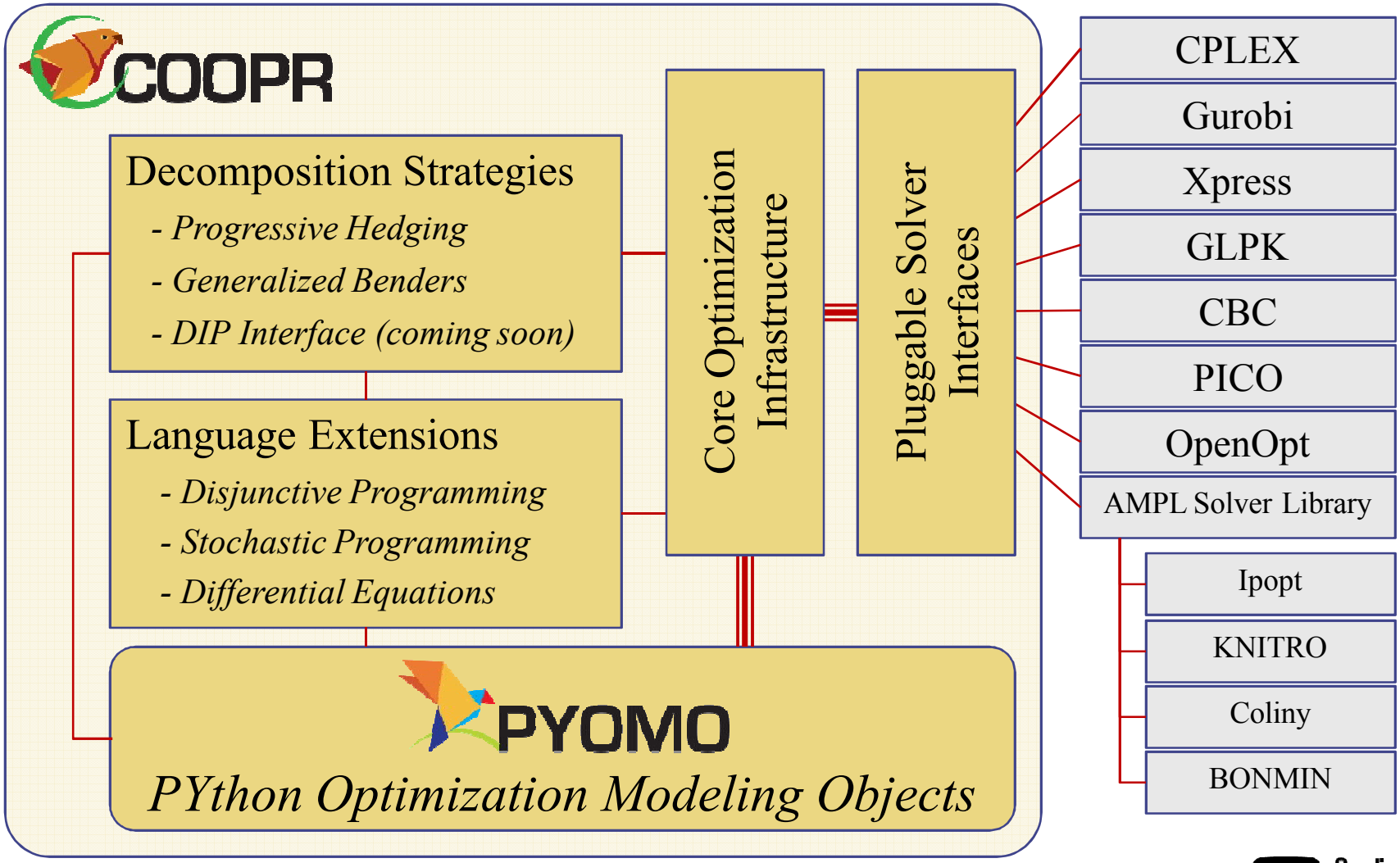
Why open source: *we want your involvement*

- Transparency and reliability

- Foster community involvement
  - Extend the modeling language
  - Develop new solvers / algorithms
  - Interface with additional external utilities
  - “Stone Soup” model

- Flexible licensing
  - Coopr/Pyomo released under 3-clause BSD license
    - No restrictions on deployment or commercial use

Coopr: *a COmmon Optimization Python Repository*



Team, Collaborators, (Known) Users

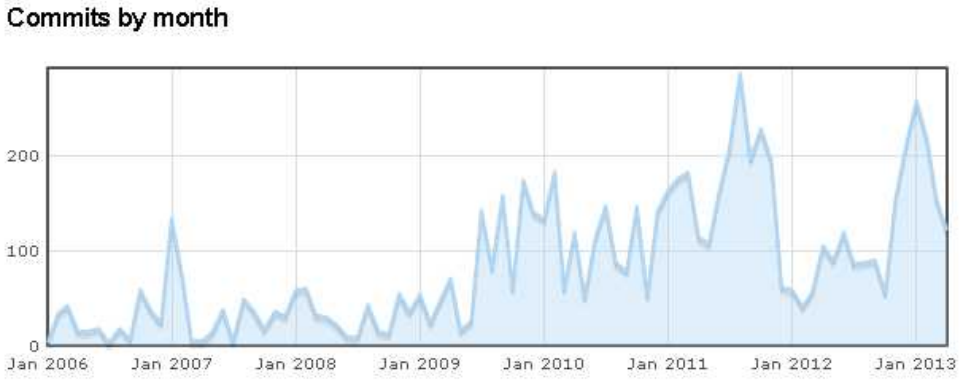
- Sandia National Laboratories
  - Bill Hart
  - Jean-Paul Watson
  - John Siirola
  - David Hart
  - Tom Brounstein
- University of California, Davis
  - Prof. David L. Woodruff
  - Prof. Roger Wets
- Texas A&M University
  - Prof. Carl D. Laird
  - Daniel Word
  - James Young
  - Gabe Hackebeil
- Carnegie Mellon University
  - Bethany Nicholson
- Texas Tech University
  - Zev Friedman
- Rose Hulman Institute
  - Tim Ekl
- William & Mary
  - Patrick Steele
- North Carolina State
  - Kevin Hunter

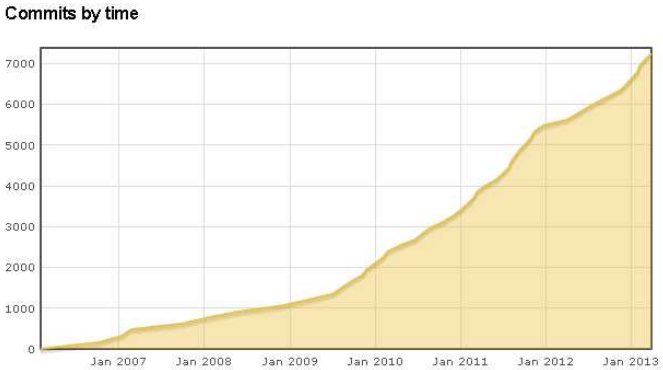
(Known) users:
- University of California, Davis
- Texas A&M University
- University of Texas
- Rose-Hulman Institute of Technology
- University of Southern California
- George Mason University
- Iowa State University
- N.C. State University
- University of Washington
- Naval Postgraduate School
- Universidad de Santiago de Chile
- University of Pisa
- Lawrence Livermore National Lab
- Los Alamos National Lab
- Federal Energy Regulatory Agency

Development, Community activity

- Coopr Forum membership doubled between 3/2012 and 5/2013
  - Active discussion list (~200 discussions since 11/2011)

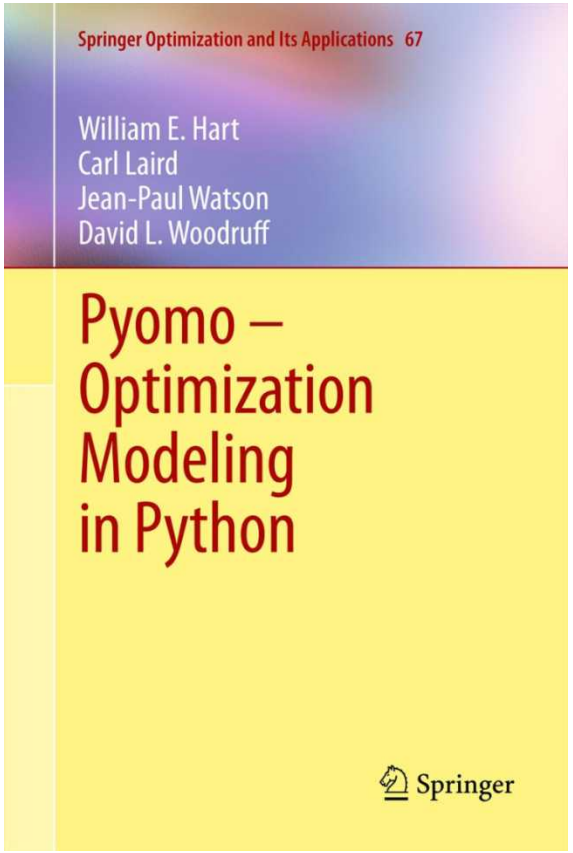
- Active developer community





For More Information…

- Project homepage
  - http://software.sandia.gov/coopr
- Mailing lists
  - "coopr-forum" Google Group
  - "coopr-dev" Google Group
- "The Book"



- Mathematical Programming Computation papers
  - Pyomo: Modeling and Solving Mathematical Programs in Python (3(3), 2011)
  - PySP: Modeling and Solving Stochastic Programs in Python (4(2), 2012)

2. An Introduction to Python

- Where credit is deserved…
  - Professor David Woodruff, University of California Davis

- Just enough Python to get you through the rest of the Pyomo tutorial

- For more information
  - www.python.org
  - Any of the great O'Reily books (www.ora.com)
  - Any of a zillion on-line tutorials

Python is a Calculator

- Python is often executed in an interactive mode
- But we don’t do this very often
  - Except when we do

Python is a Programming Language

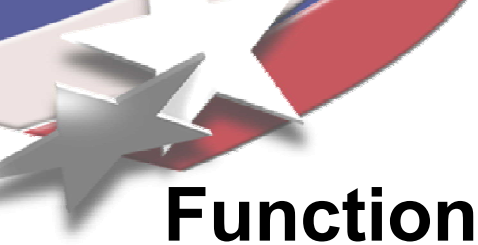
- Lists
- Dictionaries
- Sets and tuples, too
- First-class objects and functions
- (BTW: Python is an interpreted environment)

Lists (And Slicing)

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
```

Dictionaries

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```



Functions

```
>>> def foo(x):
…         for i in range(1,x+1):
…             print i
…
>>> foo(2)
1
2
>>> def foo2(x):
…         return x*x
…
>>> foo2(4)
16
```

Python Provides a Proper Application Framework

- Module definitions
- Class constructs
- And lots, lots more…

Python is Pretty Cool (Probably Cooler Than Ruby)

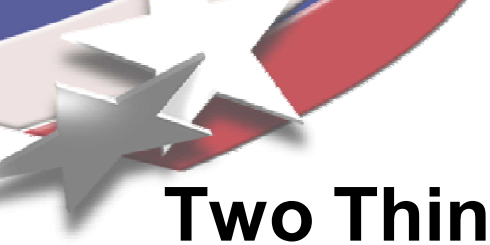
- Many in the audience can probably say much more, but here are some items of note:
  - List comprehensions
  - Magic methods (lambda functions)
  - Reentrant functions (generators)
  - Very extensive libraries
    - Exemplars: numpy, scipy, matplotlib
    - Caveat: not everything is bullet proof; it is a wild world
  - Pickling (take the state of things and encode it as an ASCII string; almost (but still, it is pretty cool))
  - Introspection
    - getattr and setattr

List Comprehensions

```
>>> vec = [2, 4, 6]

>>> [3*i for i in vec]
[6, 12, 18]

>>> [3*i for i in vec if i != 4]
[6, 18]
```

Things You Might Find Odd and/or Useful

- Indentation is important
- Shallow vs. deep copy
- Variables are passed “by reference”
- Functions often return multiple values
- First class functions (and objects)
- Mutable vs. immutable; defaults:
  - Strings: immutable
  - Integers: immutable
  - Objects: mutable



Two Things You Might Find Very Odd

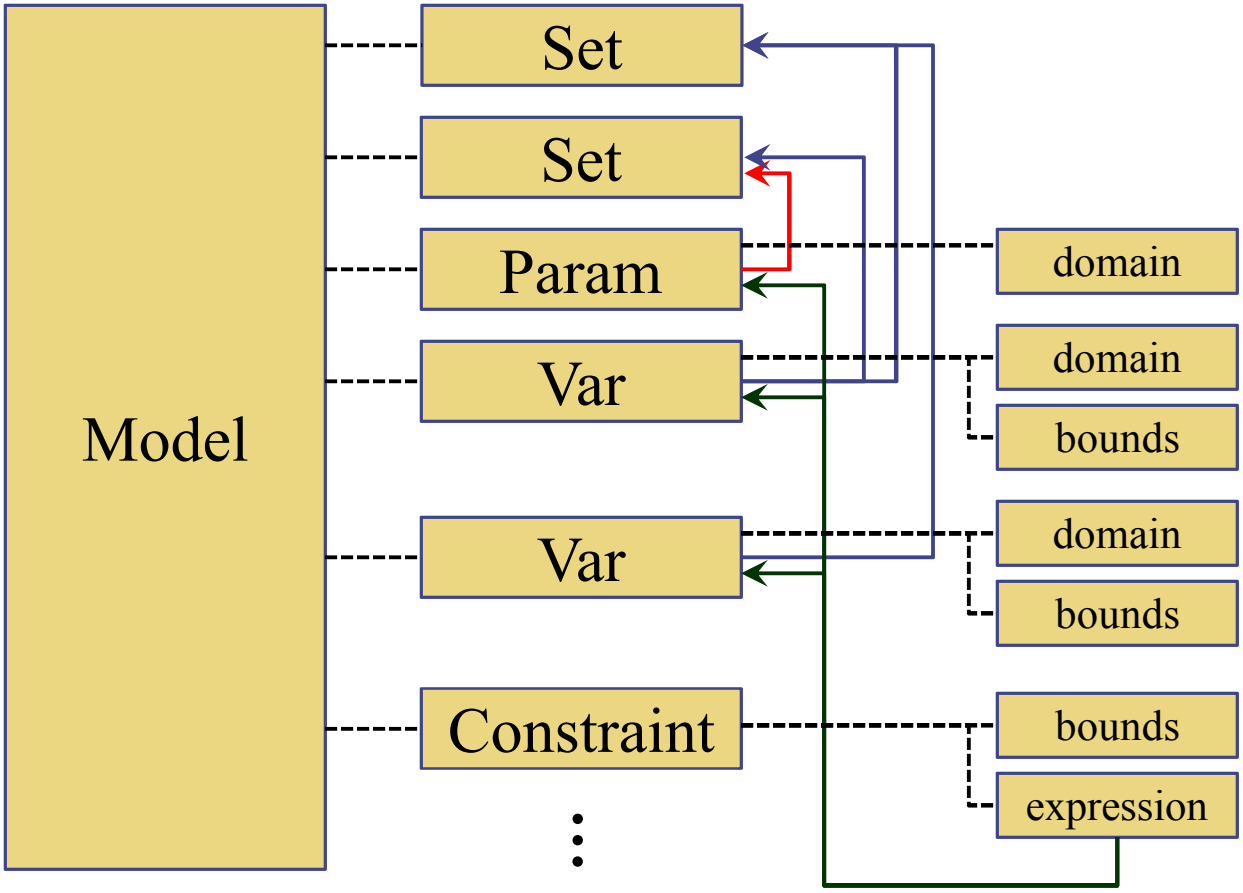
- Python is (very) weakly typed
- People get extremely excited about it

We Believe the Turing-Church Conjecture!!!!

- Everything algorithmically computable is also computable by a Turing machine
  - But you probably don't want to program a Turing machine
- We really value our time (over a computer's time)
  - Hence, our primary interest in Python
- Questions?

3. Fundamental Pyomo Components

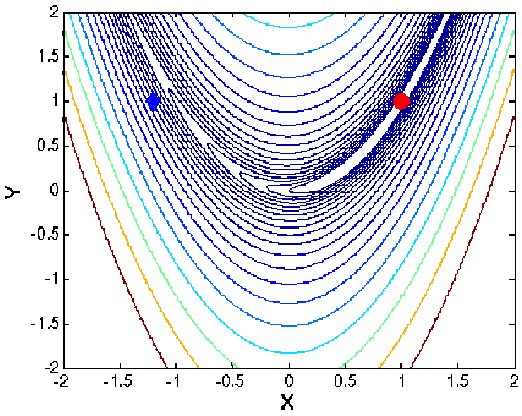
- Pyomo is an *object model* for describing optimization problems



Cutting to the chase: a simple Pyomo model

- rosenbrock.py:

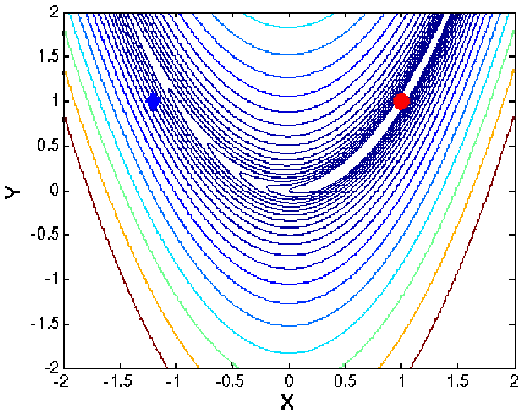
```
from coopr.pyomo import *

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 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 "coopr.pyomo" namespace:

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

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

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

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

Getting Started: the *Model*

```
from coopr.pyomo 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*

- Scalar 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

- 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

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

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

Python *list comprehensions* are very common for working over indexed variables

```
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.

Lists of Constraints

```
model.IDX = range(10)
model.b = Var(model.IDX)
model.c3 = ConstraintList()
for i in model.IDX:
    model.c3.add( (model.b[i] - i) ** 2 <= 1 )
```

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

4. Concrete Modeling

Putting It All Together: *Concrete p-Median*

$$\min \quad \sum_{n \in N, m \in M} \mathrm{d}_{\mathrm{n,m}} x_{n,m}$$

$$s.t. \quad \sum_{n \in N} x_{n,m} = 1 \qquad \forall m \in M$$

$$x_{n,m} \le y_n \qquad \forall n \in N, m \in M$$

$$\sum_{n \in N} y_n = P$$

$$0 \le x \le 1 \qquad y \in \{0,1\}$$

Concrete p-Median (1)

```
from coopr.pyomo import *

N = 5
M = 6
P = 3

import random
random.seed(1000)
d = { (n,m): random.uniform(1.0,2.0)
      for n in range(N) for m in range(M) }

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

  - Installed to:
    - `[PYTHONHOME]\Scripts` [Windows; typically `C:\Python27\Scripts`]
    - `[PYTHONHOME]/bin` [Linux; typically `~/coopr/bin` or `/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 Coopr will envoke
`--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

Syntax reminders:

```
ConcreteModel()
Var( [index, …], [within=domain], [bounds=(lower,upper)] )
Constraint( [index, …], [expr=expression|rule=function] )
ConstraintList();    c.add( expression )
Objective( sense={maximize|minimize},
           expr=expression|rule=function )
```

Concrete Knapsack: *Solution*

```
from coopr.pyomo 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 )
```

5: Abstract Modeling

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.IDX2 = Set( initialize = model.IDX * model.IDX,
                  filter = lower_tri_filter )
```

The filter returns *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.

Note: if you omit the "`rule`" keyword, Pyomo automatically looks for a function "<constraint name>`_rule`"

```
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 )
```

"`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 is imported from "`.dat`" file
  - Format similar to AMPL style
  - Explicit data from "`param`" declarations
  - External data through "`import`" declarations:
    - Excel

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

    - Databases

```
import "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 (1)

```
from coopr.pyomo import *
import random

random.seed(1000)

model = AbstractModel()

model.N = Param( within=PositiveIntegers )
model.Locations = RangeSet( 1,model.N )
model.P = Param( within=RangeSet( 1,model.N ) )
model.M = Param( within=PositiveIntegers )
model.Customers = RangeSet( 1,model.M )

model.d = Param( model.Locations, model.Customers, rule=
    lambda n, m, model : random.uniform(1.0,2.0), within=Reals)

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

Abstract p-Median (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=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 (3)

```
param N := 10;
param M := 6;
param P := 3;
```

In Class Exercise: Abstract Knapsack

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

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

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

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

Syntax reminders:

```
AbstractModel()
Var( [index, …], [within=domain], [bounds=(lower,upper)] )
Constraint( [index, …], [expr=expression|rule=function] )
ConstraintList();    c.add( expression )
Objective( sense={maximize|minimize},
           expr=expression|rule=function )
```

Abstract Knapsack: *Solution*

```
from coopr.pyomo 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;
```

6: Pyomo and Python: Idioms and Efficiency

- Being embedded in a high-level (and interpreted) programming language can present challenges
  - Inability to constrain syntax => users have many guns
  - Some of approaches may be *very* slow

- Some of the blame can be placed on Python
  - But a lot can be blamed on Pyomo

What are *reasonable* performance expectations?

- Python is a *byte-compiled scripting language*
  - and Pyomo is pure Python
  - …so expectations were not high
  - *…and raw speed has never been a goal!*

- Early experiences bore this out… in November, 2010:
  - *p*-median facility location
    - AMPL model construction time: ~4 seconds
    - Pyomo model construction time: >2000 seconds
  - Logistics disruption modeling
    - GAMS solution time: ~20 seconds
    - Pyomo solution time: >200 seconds

- ...but the gap is closing… in Coopr 3.2:
  - *p*-median facility location: ~36 seconds
  - Logistics disruption modeling: ~25 seconds

Managing edge cases

- Linking time; consider:

$$T \in [0..T_{\max}]$$

$$x_t = c x_{t-1} \qquad \{t \mid t \in T, t \neq 0\}$$

Managing edge cases

- Linking time; consider:

$$T \in [0..T_{\max}]$$

$$x_t = cx_{t-1} \qquad \{t \mid t \in T, t \neq 0\}$$

```
model.T = RangeSet(0, model.Tmax)

model.Tnot0 = model.T - [ 0 ]
def rule1(model, t):
    return model.x[t] = model.c * model.x[t-1]
model.link = Constraint( model.Tnot0, rule=rule1)
```

Managing edge cases

- Linking time; consider:

$$T \in [0..T_{\max}]$$

$$x_t = cx_{t-1} \qquad \{t \mid t \in T, t \neq 0\}$$

```
model.T = RangeSet(0, model.Tmax)
```

```
def rule2(model, t):
    if t == 0:
        return Constraint.Skip
    return model.x[t] = model.c * model.x[t-1]
model.link = Constraint( model.T, rule=rule2)
```

Managing edge cases

- Linking time; consider:

$$T \in [0..T_{\max}]$$

$$x_t = cx_{t-1} \qquad \{t \mid t \in T, t \neq 0\}$$

```
model.T = RangeSet(0, model.Tmax)

model.Tnot0 = model.T - [ 0 ]
def rule1(model, t):
     return model.x[t] = model.c * model.x[t-1]
model.link = Constraint( model.Tnot0, rule=rule1)


def rule2(model, t):
    if t == 0:
        return Constraint.Skip
     return model.x[t] = model.c * model.x[t-1]
model.link = Constraint( model.T, rule=rule2)
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_{n} \sum_{m} d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule1(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans = ans + model.d[n,m]*model.x[n,m]
    return ans
```

```
model.obj = Objective( rule=ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule2(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans += model.d[n,m]*model.x[n,m]
    return ans
```

```
model.obj = Objective( rule=ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule3(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=ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule4(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=ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule1(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans = ans + model.d[n,m]*model.x[n,m]
    return ans

def rule2(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans += model.d[n,m]*model.x[n,m]
    return ans

def rule3(model):
    return sum( [ model.d[n,m]*model.x[n,m]
        for n in model.Locations for m in model.Customers ] )

def rule4(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=ruleN )
```

[n = m = 1..640]

rule1:	>>10000 sec
rule2:	9.0 sec
rule3:	14.6 sec
rule4:	8.9 sec

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

```
model.a = Param( model.N, model.M, default=0, mutable=True )

def rule1(model,n):
     return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n] )
```

For n = 1..10, m = 1..1e5, 4% nonzero,

25.5 seconds to generate the constraint
1e5 terms in the constraint (dense!!)

```
model.C = Constraint( model.N, rule=ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

```
model.a = Param( model.N, model.M, default=0, mutable=True )
```

```
def rule2(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m] != 0 )
```

For n = 1..10, m = 1..1e5, 4% nonzero,

5 seconds *slower*, and still dense!

```
model.C = Constraint( model.N, rule=ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

```
model.a = Param( model.N, model.M, default=0, mutable=True )
```

```
def rule3(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if value(model.a[n,m]) != 0 )

def rule4(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m].value != 0 )

model.C = Constraint( model.N, rule=ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

[n = 1..10, m = 1..1e5, 4% fill]

rule1: 25.5 sec
rule2: 30.5 sec
rule3: 7.6 sec
rule4: 5.7 sec

```
model.a = Param( model.N, model.M, default=0, mutable=

def rule1(model,n):
    return sum( model.a[n,m] * model.x[m] for m in mo

def rule2(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m] != 0 )

def rule3(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if value(model.a[n,m]) != 0 )

def rule4(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m].value != 0 )

model.C = Constraint( model.N, rule=ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

```
def rule1(model,t,d,p):
    if t < model.dStart[d] or t > model.dEnd[d]:
        return Constraint.Skip
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]
model.C1 = Constraint( model.TIME, model.DISRUPTIONS, model.PRODUCTS, rule=rule1 )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

```
def rule2(model,t,d,p):
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]

def _filter2(model,t,d,p):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS * model.PRODUCTS,
                                filter=_filter2 )
model.C2 = Constraint( model.ACTIVE_DISRUPTIONS, rule=rule2 )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

```
def rule2(model,t,d,p):
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]
```

```
def _filter3(model,t,d):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS, filter=_filter3 )
model.C3 = Constraint( model.ACTIVE_DISRUPTIONS, model.PRODUCTS, rule=rule2 )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products$$

[t = d = 1..250,
p =1..10,
t*d = 2% fill]

C1: 19.8 sec
C2: 25.5 sec
C3: 3.2 sec

```
def rule1(model,t,d,p):
    if t < model.dStart[d] or t > model.dEnd[d]:
        return Constraint.Skip
    return model.storage[t,p,d] == model.storage[t-1,p
model.C1 = Constraint( model.TIME, model.DISRUPTIONS, m

def rule2(model,t,d,p):
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]

def _filter2(model,t,d,p):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS * model.PRODUCTS,
                                filter=_filter2 )
model.C2 = Constraint( model.ACTIVE_DISRUPTIONS, rule=rule2 )

def _filter3(model,t,d):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS, filter=_filter3 )
model.C3 = Constraint( model.ACTIVE_DISRUPTIONS, model.PRODUCTS, rule=rule2 )
```

The Performance "Elephant": Memory

- Known issue …
  - Python uses a fairly heavy-weight object model
  - "Significant" recent improvements in low-level core components
    - Coopr 3.3 uses <50% of the memory of Coopr 3.0
  - But…
    - 640 x 640 *p*-median problem still consumes ~1 GB.

- Focus of current efforts, with several more enhancements on the horizon.

7. Scripting the optimization process

- The `pyomo` command is useful for "simple" workflows
  - Model → Solve → Analyze Results
- Many problems require more complex workflows
  - Multi-start local search
  - Sequential models
  - Decomposition techniques
  - Model manipulation, transformations
  - Post-processing (e.g. generating plots)
- Solution: Create a custom "driver script"

The "simplest" driver script

```
from coopr.opt import SolverFactory
from Example import model
```

This assumes you have written your model in a file called "`Example.py`" located in the current working directory.

```
instance = model.create('example.dat')
```

Assuming you wrote an *Abstract* model, you must create a concrete instance by applying data. *Concrete* models may omit this step.

```
solver = SolverFactory('ipopt')
```

Create an instance of the "ipopt" solver using the global registry of available solvers

```
results = solver.solve(instance)
results.write()
```

Solve the model and print the results

Manipulating optimization results: plotting

```
from coopr.opt import SolverFactory
from Example import model

instance = model.create('example.dat')
solver = SolverFactory('ipopt')
results = solver.solve(instance)

instance.load(results)

measured = []
estimate = []
for t in instance.TIME:
    measured.append( value(instance.x[t]) )
    estimate.append( value(instance.x_hat[t]) )

import matplotlib.pyploy as plt
plt.plot(measured, 'o')
plt.plot(estimate)
plt.show()
```

It is frequently convenient to manipulate the results in the context of the model. To do that you must load the results back into the model.

Assuming your model has

```
Set TIME
Param x(TIME)
Var x_hat(TIME)
```

Note: use the `value()` function to extract the current numeric value of the `Var` / `Param`

Use matplotlib to generate a plot showing the data fit

Iterative solves: multi-start local search

```
from coopr.opt import SolverFactory
from Example import model

instance = model.create('example.dat')
solver = SolverFactory('ipopt')

from random import random

bestObj = float('inf')

numTries = 10

for trial in range(numTries):
    for t in instance.TIME:
        instance.x_hat[t] = random()

    results = solver.solve(instance)

    instance.load(results)

    trialObj = value(instance.obj)

    if trialObj is not None and trialObj < bestObj:
        best = results

best.write()
```

Assign each variable in `x_hat` a random starting value in [0, 1)

Iterative solves: adding cuts

```
from coopr.pyomo import *
from coopr.opt import SolverFactory
from Sudoku import model, printSolution

model.Cuts = ConstraintList()
solver = SolverFactory('glpk')
while True:
    model.preprocess()
    results = solver.solve(model)

    if str(results.Solution.Status) != 'optimal':
        break

    model.load(results)
    printSolution(model)
    model.Cuts.add(
        sum( (1-val) for val in model.y.itervalues()
             if value(val) >= 0.5 ) +
        sum( val for val in model.y.itervalues()
             if value(val) < 0.5 ) >= 1 )
```

In this example, `model` is Concrete

Note: if you edit a model instance:
- add/remove constraints
- fix/free variables
- change Param values

you *must* call `preprocess()`

8. Hierarchical (block-oriented) modeling

- "Real world" models have significant macroscopic structure
  - Arises from interactions among physical real-world entities
  - Desire to preserve that structure in optimization model
- Solution: entities map to model *blocks*
  - Collections of model components
    - Var, Param, Set, Constraint, etc.
  - Blocks may be arbitrarily nested
- Why blocks?
  - Support reusable modeling components
  - Express distinctly modeled concepts as distinct objects
  - Manipulate modeled components as distinct entities
  - Explicitly expose model structure (e.g., for decomposition)
- Prior art
  - Ubiquitous in the simulation community
  - Rare in Math Programming environments
    - Notable exceptions: ASCEND, JModelica.org

9. Disjunctive Programming

- Generalized Disjunctive Programming (GDP)
  - Switching entire blocks on/off through binary variables

- Introduce new modeling components:
  - "Disjunct"
    - a new form of model block
  - "Disjunction"
    - a new constraint for enforcing logical XOR over disjunctive sets

- Leverage model transformation extensions to convert GDP problems to MI(N)LP

$$
\begin{aligned}
\min \quad & \sum_k c_k + f(x) \\
s.t. \quad & g(x) \le 0 \\
& \bigvee_{i \in D_k} \begin{bmatrix} Y_{ik} \\ h_{ik}(x) \le o \\ c_k = \gamma_{ik} \end{bmatrix} \\
& \Omega(Y) = true \\
& Y_{ik} \in \{true, false\}
\end{aligned}
$$

GDP Example: Jobshop Scheduling

- [Raman & Grossmann, 1994]

$$\begin{aligned}
\min \quad & T \\
s.t. \quad & T \geq t_i + \sum_{j \in J(i)} \tau_{ij}, \quad \forall i \in I, \\
& \left[\begin{array}{c} Y_{ik} \\ t_i + \sum_{\substack{m \in J(i) \\ m \leq j}} \tau_{im} \leq t_k + \sum_{\substack{m \in J(k) \\ m < j}} \tau_{km} \end{array} \right] \vee \left[\begin{array}{c} Y_{ki} \\ t_k + \sum_{\substack{m \in J(k) \\ m \leq j}} \tau_{km} \leq t_i + \sum_{\substack{m \in J(i) \\ m < j}} \tau_{im} \end{array} \right], \\
& \quad \forall j \in C_{ij}, \forall i,k \in I, i < k, \\
& T \geq 0, \quad t_i \geq 0, \\
& Y_{ik} \in \{true, false\}
\end{aligned}$$

GDP Example: Jobshop Scheduling

```
from coopr.pyomo import *
from coopr.gdp import *

model = AbstractModel()
model.JOBS        = Set()
model.STAGES      = Set()
model.I_BEFORE_K = RangeSet(0,1)

model.tau = Param( model.JOBS, model.STAGES, default=0 )

def t_bounds(model, I):
    return (0, sum(model.tau[idx] for idx in model.tau))
model.t = Var( model.JOBS, within=NonNegativeReals, bounds=t_bounds )

model.T = Var()
```

GDP Example: Jobshop Scheduling (2)

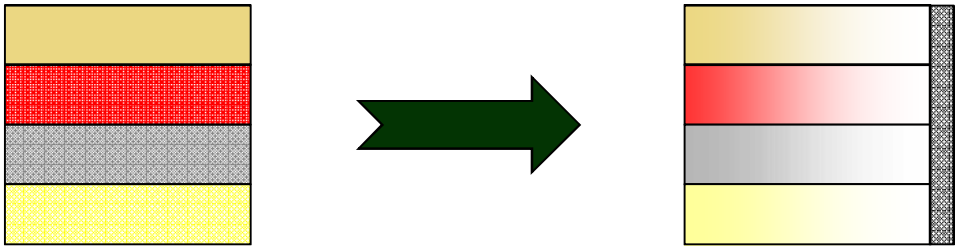
```
model.makespan = Objective(expr=model.T)

def _L_filter(model, i, k, j):
    return i < k and model.tau[i,j] and model.tau[k,j]
model.L = Set( initialize=model.JOBS * model.JOBS * model.STAGES,
               filter=_L_filter)

def _feas(model, i):
    return model.T >= model.t[i] + sum([model.tau[i,m] for m in model.STAGES])
model.Feas = Constraint(model.JOBS, rule=_feas)
```

GDP Example: Jobshop Scheduling (3)

```
def _NoClash(disjunct, i, k, j, IthenK):
    model = disjunct.model()
    lhs = model.t[i] + sum([m<j and model.tau[i,m] or 0 for m in model.STAGES])
    rhs = model.t[k] + sum([m<j and model.tau[k,m] or 0 for m in model.STAGES])
    if IthenK:
        disjunct.c = Constraint( expr= lhs + model.tau[I,J] <= rhs )
    else:
        disjunct.c = Constraint( expr= rhs + model.tau[K,J] <= lhs )
model.NoClash = Disjunct( model.L, model.I_BEFORE_K, rule=_NoClash )

def _disj(model, i, k, j):
    return [ model.NoClash[i,k,j,IthenK] for IthenK in model.I_BEFORE_K ]
model.disj = Disjunction(model.L, rule=_disj)
```

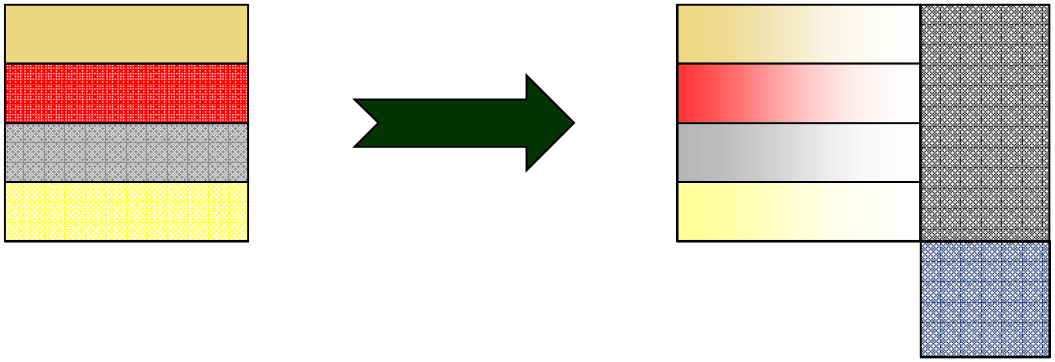
Solving GDP models

- Automated transformations generate “flat” MI(N)LPs
  - Big-M relaxation



```
pyomo --preprocess=coopr.gdp.bigm jobshop.py jobshop.dat
```

  - Convex hull relaxation

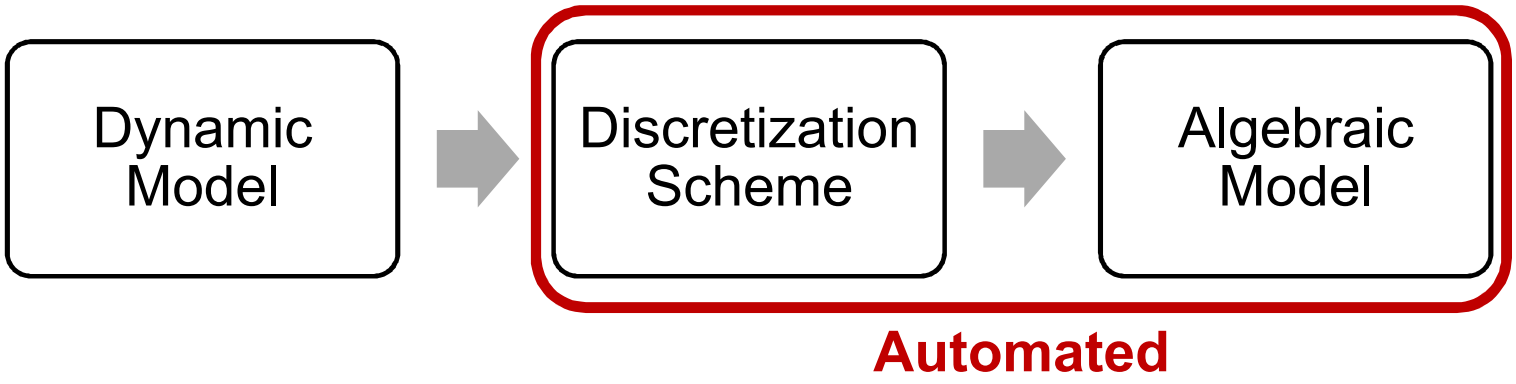


```
pyomo --preprocess=coopr.gdp.chull jobshop.py jobshop.dat
```

10. Differential-Algebraic Models

- Where credit is deserved…
  - Bethany Nicholson, Carnegie Mellon University

- Provide support for optimizing dynamic models
  - Modeling components for differential equations
  - Automated transformations to discretize model to (N)LP



General Syntax

- DifferentialSet

```
m.t = DifferentialSet(bounds=(0,10))
m.t = DifferentialSet(initialize=[1,2,3,4])
m.t = DifferentialSet()
```

- Differential

```
m.xdot = Differential(
        dvar=m.x,
        rule=_xdot,
        dset=m.t,
        bounds=(0,10),
        initialize=_init_xdot )
```

Optimal Control Example (1)

$$\min \; x_3(t_f)$$
$$s.t. \quad \dot{x_1} = x_2$$
$$\dot{x_2} = -x_2 + u$$
$$\dot{x_3} = x_1^2 + x_2^2 + 0.005 * u^2$$
$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$
$$x_1(0) = 0$$
$$x_2(0) = -1$$
$$x_3(0) = 0$$
$$t_f = 1$$

```
from coopr.pyomo import *
from coopr.dae import *

m = ConcreteModel()
m.t = DifferentialSet(bounds=(0,1))
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)
m.u = Var(m.t)

def _obj(m):
    return m.x3[1]
m.obj = Objective(rule=_obj)

def _x1dot(m,t):
    return m.x2[t]
m.x1dot = Differential(dv=m.x1, rule=_x1dot)

def _x2dot(m,t):
    return -m.x2[t]+m.u[t]
m.x2dot = Differential(dv=m.x2, rule=_x2dot)

def _x3dot(m,t):
    return m.x1[t]**2+m.x2[t]**2+0.005*m.u[t]**2
m.x3dot = Differential(dv=m.x3, rule=_x3dot)

def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

Optimal Control Example (2)

$$\min \; x_3(t_f)$$
$$s.t. \quad \dot{x_1} = x_2$$
$$\dot{x_2} = -x_2 + u$$
$$\dot{x_3} = x_1^2 + x_2^2 + 0.005 * u^2$$
$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$
$$x_1(0) = 0$$
$$x_2(0) = -1$$
$$x_3(0) = 0$$
$$t_f = 1$$

```
from coopr.pyomo import *
from coopr.dae import *

m = ConcreteModel()
```


```
from coopr.pyomo import *
from coopr.dae import *
```


```
m.obj = Objective(rule=_obj)

def _x1dot(m,t):
    return m.x2[t]
m.x1dot = Differential(dv=m.x1, rule=_x1dot)

def _x2dot(m,t):
    return -m.x2[t]+m.u[t]
m.x2dot = Differential(dv=m.x2, rule=_x2dot)

def _x3dot(m,t):
    return m.x1[t]**2+m.x2[t]**2+0.005*m.u[t]**2
m.x3dot = Differential(dv=m.x3, rule=_x3dot)

def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

Optimal Control Example (3)

$$\min \; x_3(t_f)$$
$$s.t. \quad \dot{x}_1 = x_2$$
$$\dot{x}_2 = -x_2 + u$$
$$\dot{x}_3 = x_1^2 + x_2^2 + 0.00$$
$$x_2 - 8 * (t - 0.5)^2 +$$
$$x_1(0) = 0$$
$$x_2(0) = -1$$
$$x_3(0) = 0$$
$$t_f = 1$$

```
from coopr.pyomo import *
from coopr.dae import *

m = ConcreteModel()
m.t = DifferentialSet(bounds=(0,1))
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)
m.u = Var(m.t)

def _obj(m):
```

```
m = ConcreteModel()
m.t = DifferentialSet(bounds=(0,1))
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)
m.u = Var(m.t)
```

```
def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

Optimal Control Example (4)

$$\min \; x_3(t_f)$$

$$s.t. \qquad \dot{x_1} = x_2$$

$$\dot{x_2} = -x_2 + u$$

$$\dot{x_3} = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$

$$x_1(0) = 0$$

$$x_2(0) = -1$$

$$x_3(0) = 0$$

$$t_f = 1$$

```
from coopr.pyomo import *
from coopr.dae import *

m = ConcreteModel()
m.t = DifferentialSet(bounds=(0,1))
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)
m.u = Var(m.t)

def _obj(m):
    return m.x3[1]
m.obj = Objective(rule=_obj)

def _x1dot(m,t):
    return m.x2[t]
m.x1dot = Differential(dv=m.x1, rule=_x1dot)
```


```
def _obj(m):
    return m.x3[1]
m.obj = Objective(rule=_obj)
```


```
def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

Optimal Control Example (5)

$$\min \; x_3(t_f)$$

$$s.t. \quad \dot{x_1} = x_2$$

$$\dot{x_2} = -x_2 + u$$

$$\dot{x_3} = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$

```
from coopr.pyomo import *
from coopr.dae import *

m = ConcreteModel()
m.t = DifferentialSet(bounds=(0,1))
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)
m.u = Var(m.t)

def _obj(m):
    return m.x3[1]
m.obj = Objective(rule=_obj)

def _x1dot(m,t):
    return m.x2[t]
m.x1dot = Differential(dv=m.x1, rule=_x1dot)

def _x2dot(m,t):
```

```
def _x1dot(m,t):
    return m.x2[t]
m.x1dot = Differential(dv=m.x1, dset=m.t,
      rule=_x1dot)
```

```
def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

Optimal Control Example (6)

$$\min \quad \int_0^1 x_3 \, dt$$

$$s.t. \quad \dot{x}_1 = x_2$$

$$\dot{x}_2 = -x_2 + u$$

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$

$$x_1(0) = 0$$

$$x_2(0) = -1$$

$$x_3(0) = 0$$

$$t_f = 1$$

```
def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

```
def _obj(m):
    return m.x3[1]
m.obj = Objective(rule=_obj)

def _x1dot(m,t):
    return m.x2[t]
m.x1dot = Differential(dv=m.x1, rule=_x1dot)

def _x2dot(m,t):
    return -m.x2[t]+m.u[t]
m.x2dot = Differential(dv=m.x2, rule=_x2dot)

def _x3dot(m,t):
    return m.x1[t]**2+m.x2[t]**2+0.005*m.u[t]**2
m.x3dot = Differential(dv=m.x3, rule=_x3dot)

def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

Optimal Control Example (7)

$$\min \; x_3(t_f$$
$$s.t. \quad \dot{x}_1 =$$
$$\dot{x}_2 = -x_2 +$$
$$\dot{x}_3 = x_1^2 + x_2^2 + 0.($$
$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$
$$x_1(0) = 0$$
$$x_2(0) = -1$$
$$x_3(0) = 0$$
$$t_f = 1$$

```
from coopr.pyomo import *
from coopr.dae import *
```

```
def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

```
m.x1dot = Differential(dv=m.x1, rule=_x1dot)

def _x2dot(m,t):
    return -m.x2[t]+m.u[t]
m.x2dot = Differential(dv=m.x2, rule=_x2dot)

def _x3dot(m,t):
    return m.x1[t]**2+m.x2[t]**2+0.005*m.u[t]**2
m.x3dot = Differential(dv=m.x3, rule=_x3dot)

def _con(m,t):
    return m.x2[t]-8*(t-0.5)**2+0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

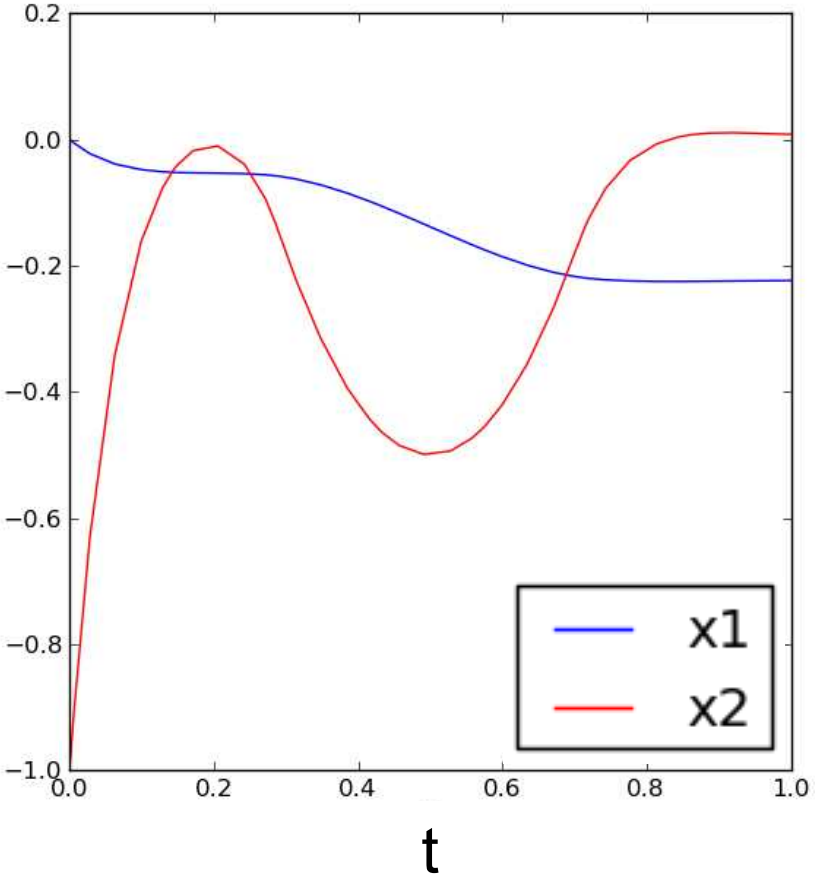

```
def _init_conditions(m,i):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = ConstraintList(
                        rule=_init_conditions)
```

Optimal Control Example: driver script

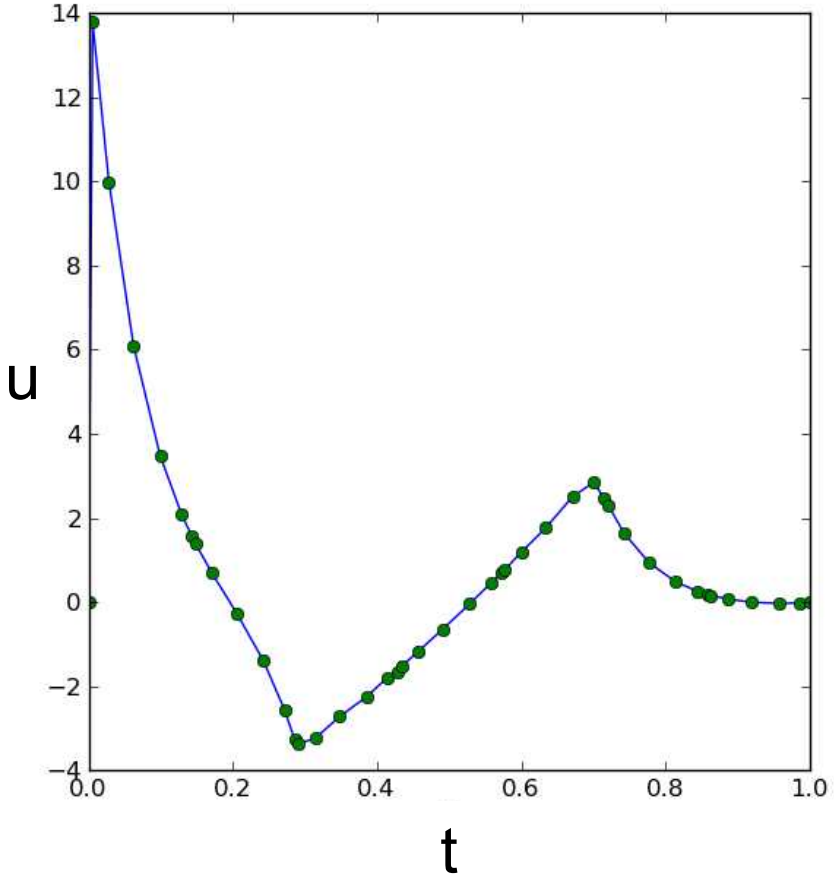
```
from coopr.pyomo import *
from coopr.dae import *
from coopr.opt import SolverFactory
from coopr.dae.colloc import Collocation_Discretization

# Import model
from example import m as model

# Create model instance
instance = model.create()

# Discretize instance using radau collocation
discretizer = Collocation_Discretization(instance)
discretized_instance = discretizer.apply(nfe=7, ncp=6)

# Solve discrete model
opt = SolverFactory('ipopt')
results = opt.solve(discretized_instance)

# Load Results
discretized_instance.load(results)
```

Optimal Control Example: results

Differential Variables

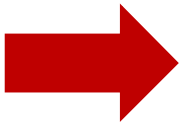


Control Variable



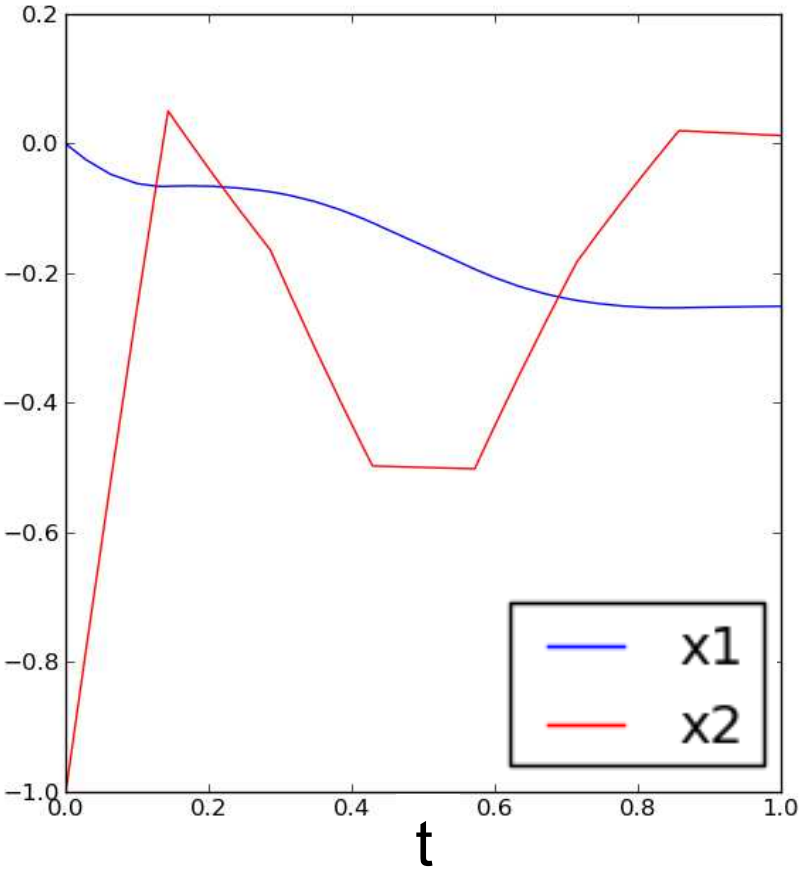
Optimal Control Example: alternate driver

```
from coopr.pyomo import *
from coopr.dae import *
from coopr.opt import SolverFactory
from coopr.dae.colloc import Collocation_Discretization
from smallExample import model

instance = model.create()

# Discretize instance using radau collocation
discretizer = Collocation_Discretization(instance)
instance = discretizer.apply(nfe=7, ncp=6)

# Control variable u made constant over each finite element
instance = discretizer.reduce_collocation_points(
    var=instance.u, ncp=1, diffset=instance.t )

opt = SolverFactory('ipopt')
results = opt.solve(instance)

instance.load(results)
```

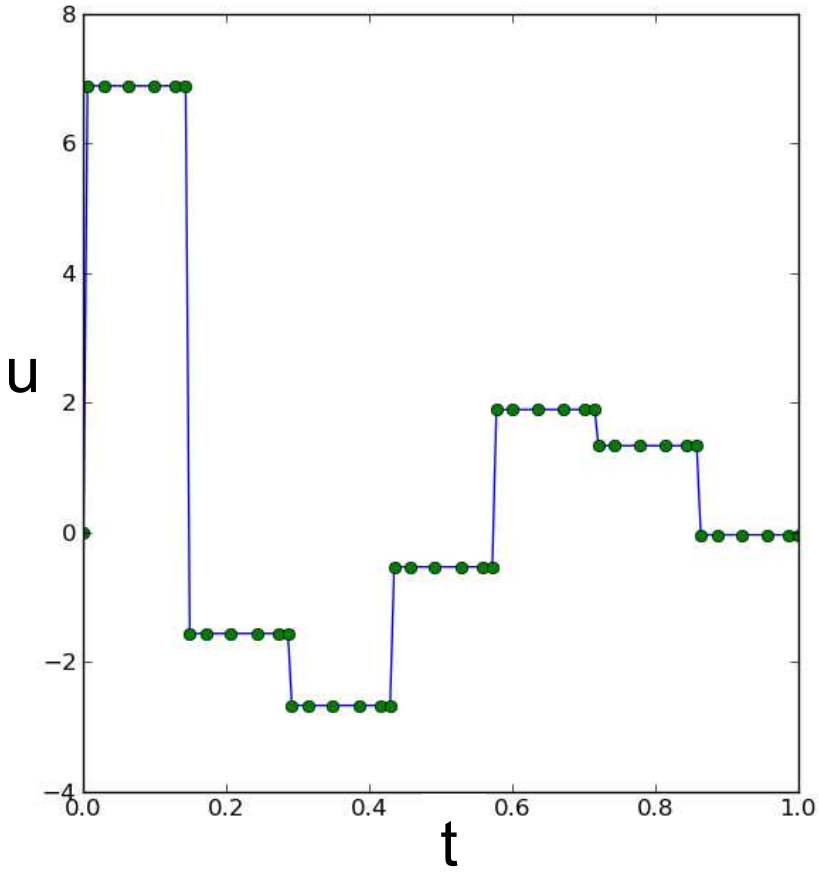


Optimal Control Example: results

Differential Variables



Control Variable



Hart, Siirola, Watson, Woodruff, p. 100

11. Stochastic Programming: Coopr.PySP

- Constructing the deterministic scenario model
- Specifying the scenario tree
- Specifying scenario instance data
- Creating and solving the extensive form
- Decomposition

Constructing and Solving Block-Diagonal Models

- PySP: Stochastic Programming in Python
  - Begin with a deterministic (Pyomo) system model
  - Annotate model with data that defines the scenario(*) tree
  - Leverage automatic transformation, model decomposition

- Automated solution strategies
  - Solve the Extensive Form
    - Replicate deterministic model
    - Form nonanticipitivity constraints
    - Send to solver
  - Decompose (Progressive Hedging)
    - Replicate deterministic model
    - Duplicate complicating variables
    - Solve scenarios independently
    - Blend complicating variables
    - Weight scenarios to encourage convergence
    - Iterate

PySP: Formulating the System Model (1)

```
from coopr.pyomo import *
model = AbstractModel()

# Parameters
model.CROPS = Set()
model.TOTAL_ACREAGE = Param( within=PositiveReals )
model.PriceQuota = Param( model.CROPS, within=PositiveReals )
model.SubQuotaSellingPrice = Param( model.CROPS, within=PositiveReals )
model.SuperQuotaSellingPrice = Param( model.CROPS )
model.CattleFeedRequirement = Param( model.CROPS, within=NonNegativeReals )
model.PurchasePrice = Param( model.CROPS, within=PositiveReals )
model.PlantingCostPerAcre = Param( model.CROPS, within=PositiveReals )
model.Yield = Param( model.CROPS, within=NonNegativeReals )

# Variables
model.DevotedAcreage = Var( model.CROPS, bounds=(0.0, model.TOTAL_ACREAGE) )
model.QuantitySubQuotaSold = Var( model.CROPS, bounds=(0.0, None) )
model.QuantitySuperQuotaSold = Var( model.CROPS, bounds=(0.0, None) )
model.QuantityPurchased = Var( model.CROPS, bounds=(0.0, None) )
model.FirstStageCost = Var()
model.SecondStageCost = Var()
```


Birge and Louveaux's (1997) Farmer Example

PySP: Formulating the System Model (2)

```
# Constraints
def ConstrainTotalAcreage_rule(model):
    return summation(model.DevotedAcreage) <= model.TOTAL_ACREAGE

model.ConstrainTotalAcreage = Constraint()

def EnforceCattleFeedRequirement_rule(model, i):
    return model.CattleFeedRequirement[i] <= ( model.Yield[i] \
        * model.DevotedAcreage[i] ) + model.QuantityPurchased[i] \
        - model.QuantitySubQuotaSold[i] - model.QuantitySuperQuotaSold[i]

model.EnforceCattleFeedRequirement = Constraint( model.CROPS )

def LimitAmountSold_rule(model, i):
    return model.QuantitySubQuotaSold[i] + model.QuantitySuperQuotaSold[i] \
        - (model.Yield[i] * model.DevotedAcreage[i]) <= 0.0

model.LimitAmountSold = Constraint( model.CROPS )

def EnforceQuotas_rule(model, i):
    return(0.0, model.QuantitySubQuotaSold[i], model.PriceQuota[i])

model.EnforceQuotas = Constraint( model.CROPS )
```


Birge and Louveaux's (1997) Farmer Example

PySP: Formulating the System Model (3)

```
# Stage-specific cost computations
def ComputeFirstStageCost_rule(model):
    return 0.0 == model.FirstStageCost \
        - summation( model.PlantingCostPerAcre, model.DevotedAcreage )

model.ComputeFirstStageCost = Constraint()

def ComputeSecondStageCost_rule(model):
    expr = summation( model.PurchasePrice, model.QuantityPurchased )
    expr -= summation( model.SubQuotaSellingPrice, model.QuantitySubQuotaSold )
    expr -= summation( model.SuperQuotaSellingPrice, model.QuantitySuperQuotaSold )
    return (model.SecondStageCost - expr) == 0.0

model.ComputeSecondStageCost = Constraint()

# Objective
def Total_Cost_Objective_rule(model):
    return model.FirstStageCost + model.SecondStageCost

model.Total_Cost_Objective = Objective( sense=minimize )
```


Birge and Louveaux's (1997) Farmer Example

PySP: Specifying the Scenario Tree

```
set Stages := FirstStage SecondStage ;

set Nodes := RootNode
             BelowAverageNode
             AverageNode
             AboveAverageNode ;

param NodeStage := RootNode         FirstStage
                   BelowAverageNode SecondStage
                   AverageNode      SecondStage
                   AboveAverageNode SecondStage ;

set Children[RootNode] := BelowAverageNode
                          AverageNode
                          AboveAverageNode ;

param ConditionalProbability := RootNode          1.0
                                BelowAverageNode  0.33333333
                                AverageNode       0.33333334
                                AboveAverageNode  0.33333333 ;

set Scenarios := BelowAverageScenario
                 AverageScenario
                 AboveAverageScenario ;

param ScenarioLeafNode :=
                    BelowAverageScenario BelowAverageNode
                    AverageScenario      AverageNode
                    AboveAverageScenario AboveAverageNode ;

set StageVariables[FirstStage] :=  DevotedAcreage[*] ;

set StageVariables[SecondStage] := QuantitySubQuotaSold[*]
                                   QuantitySuperQuotaSold[*]
                                   QuantityPurchased[*] ;

param StageCostVariable := FirstStage  FirstStageCost
                           SecondStage SecondStageCost ;
```

PySP: Specifying the Scenario Instance Data

- Two methods are available to specify scenario data
  - Scenario-based
  - Node-based

- In the scenario-based approach, a single and complete .dat file is specified for each individual scenario
  - Redundant, but straightforward if computer-generated

- In the node-based approach, a single .dat file is specified for each node in the scenario tree
  - Maximally compact, but requires some book-keeping

Writing and Solving the Extensive Form (1)

- Now that you have a stochastic programming model in PySP…
- Step #1: Write the extensive form and hope that your favorite solver can actually solve it
  - Fantastic if it works
  - But often it doesn't
- In PySP, the ***runef*** script is provided to both write and solve the extensive form of a stochastic programming model
- The basic command-line:

```
runef --model-directory=models \
      --instance-directory=scenariodata \
      --solve
```

Writing and Solving the Extensive Form (2)

- After solution, you get:

```
Tree Nodes:

        Name=AboveAverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[CORN]=48.0
                QuantitySubQuotaSold[SUGAR_BEETS]=6000.0
                QuantitySubQuotaSold[WHEAT]=310.0

        Name=AverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[SUGAR_BEETS]=5000.0
                QuantitySubQuotaSold[WHEAT]=225.0

        Name=BelowAverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[SUGAR_BEETS]=4000.0
                QuantitySubQuotaSold[WHEAT]=140.0
                QuantityPurchased[CORN]=48.0

        Name=RootNode
        Stage=FirstStage
        Parent=None
        Variables:
                DevotedAcreage[CORN]=80.0
                DevotedAcreage[SUGAR_BEETS]=250.0
                DevotedAcreage[WHEAT]=170.0
```

In-Class Exercise: Stochastic Knapsack

- Extend the (abstract) deterministic knapsack models to create and solve a stochastic knapsack problem

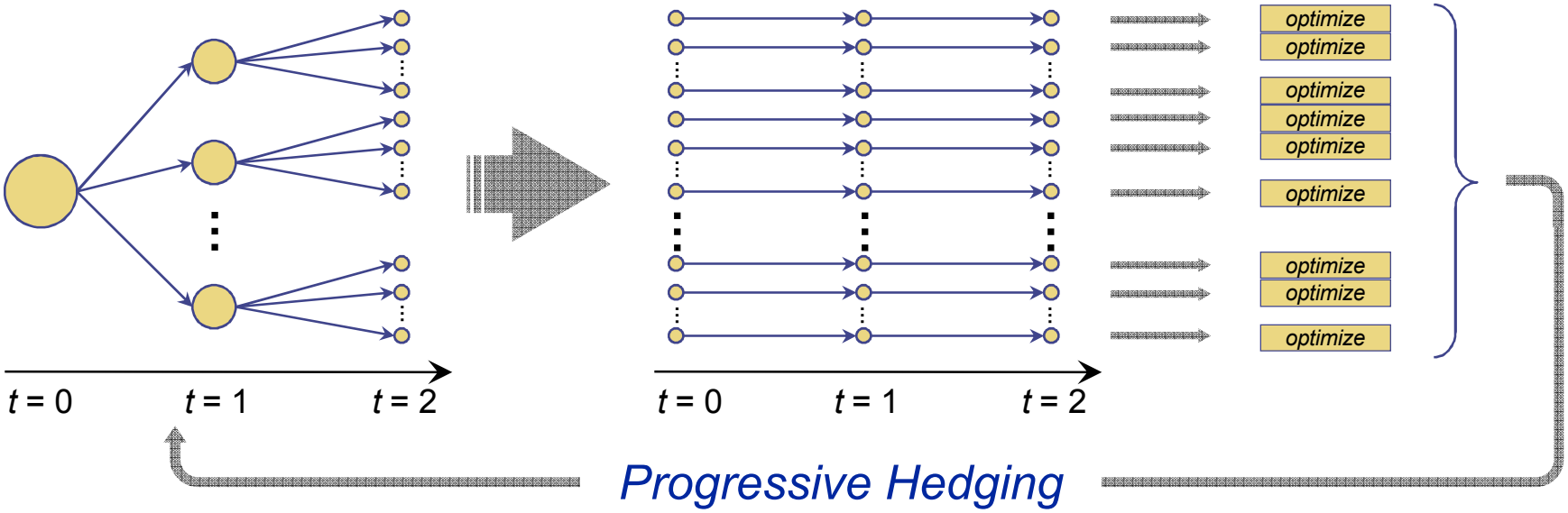
- Scenarios
  - Random values/profits – fixed weights!
  - 3 scenarios is sufficient for the exercise

- Scenario Tree

- Solve with **runef** script
  - `runef -m your-dir -i your-dir --solve –solver=glpk`

What if the EF is too difficult?

- Use problem decomposition!
  - Stage-wise (e.g., Benders)
  - Scenario-based (e.g., Progressive Hedging)



Progressive Hedging: *a Review and/or Introduction*

1. $k := 0$
2. For all $s \in \mathcal{S}$, $x_s^{(k)} := \text{argmin}_x(c \cdot x + f_s \cdot y_s) : (x, y_s) \in \mathcal{Q}_s$
3. $\bar{x}^k := (\sum_{s\in\mathcal{S}} p_s\, d_s x_s^{(k)}) / \sum_{s\in\mathcal{S}} p_s\, d_s$
4. For all $s \in \mathcal{S}$, $w_s^{(k)} := \rho(x_s^{(k)} - \bar{x}^{(k)})$
5. $k := k + 1$
6. For all $s \in \mathcal{S}$, $x_s^{(k)} := \text{argmin}_x(c \cdot x + w_s^{(k-1)} x + \rho/2 \left\|x - \bar{x}^{(k-1)}\right\|^2 + f_s \cdot y_s)$ $: (x, y_s) \in \mathcal{Q}_s$
7. $\bar{x}^{(k)} := (\sum_{s\in\mathcal{S}} p_s\, d_s x_s^{(k)}) / \sum_{s\in\mathcal{S}} p_s\, d_s$
8. For all $s \in \mathcal{S}$, $w_s^{(k)} := w_s^{(k-1)} + \rho\left(x_s^{(k)} - \bar{x}^{(k)}\right)$
9. $g^{(k)} := \frac{(1-\alpha)|\mathcal{S}|}{\sum_{s\in\mathcal{S}} p_s\, d_s} \sum_{s\in\mathcal{S}} \left\|x^{(k)} - \bar{x}^{(k)}\right\|$
10. If $g^{(k)} < \epsilon$, then go to step 5. Otherwise, terminate.

Hart, Siirola, Watson, Woodruff, p. 112

Rockafellar and Wets (1991)

Sandia National Laboratories

PySP: Generic Progressive Hedging

- If you don't care about the value of the penalty parameter (ρ) or you are willing to take chances, and/or you have time to kill:

```
runph --model-directory=models --instance-directory=scenariodata
```

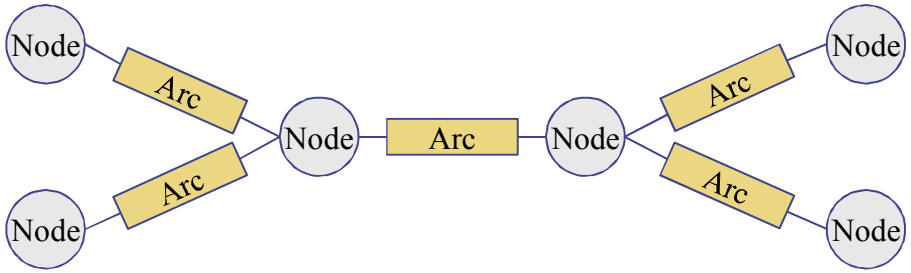
- If you think a global value of the penalty parameter will work:
  - Add the argument "`--default-rho=your-favorite-value`"

- PySP provides automatic, generic linearization mechanisms
  - Requires specification of variable lower and upper bounds
  - Specify number of breakpoints, distribution strategy

- Lots of other features, including
  - Cycle detectors, straightforward parallelization, acceleration mechanisms, bundling, asychronous solves, …

12: Reusing blocks: model libraries

- Capture connected block structure, e.g., *network flow*



  - Embed physical component models within separate blocks
  - Connect blocks using conceptual interfaces:
    - Connectors: groups of named numeric values
      - Constant, Parameter, Variable, Expression
    - "Connect" connectors with simple constraints
  - Block implementation independent of network definition

Domain	Node	Arc	Connector Vars
Fluid flow	Mass balance	Pressure Drop	Pressure; Volumetric flow
AC Power flow	KCL	Active power transfer; Reactive power transfer	Phase angle; Active power flow; Reactive power flow

A word about the example problem…

- Operating an electric power grid:

$$\sum demand = \sum generation - \sum losses \quad \forall\, t \in T$$

- (In the US) implemented through markets (run by ISO/RTO):
  - "Unit commitment" (UC) / "Day-ahead Market" (DAM)
    - MIP run ~10 hours before the start of a day
    - Sets on/off state for all generator units hourly for 24 hours
  - "Reliability Unit Commitment" (RUC)
    - MIP run ~8 hours before the start of the day
    - Commits additional generators to meet spinning reserve and reliability (N-1 robustness) requirements
  - "Economic Dispatch" (ED) / "Security-constrained ED" (SCED)
    - "Real-time" markets: LP run hourly / every 5 minutes
    - Set generation levels, prices to meet realized demand
- Problem scale
  - 100's – 1000's of buses; 2-3x lines

Hart, Siirola, Watson, Woodruff, p. 115

Simple input-output blocks

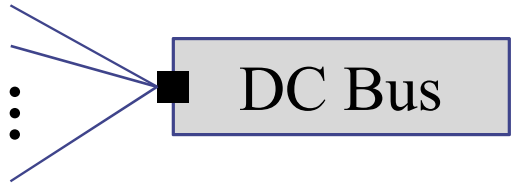
```
def dc_line_rule(line, id):

    line.B          = Param()
    line.Limit      = Param()
    line.Angle_in   = Var()
    line.Angle_out  = Var()
    line.Power      = Var( bounds= ( -line.Limit, line.Limit ) )

    line.power_flow = Constraint( expr=
        line.Power == line.B*(line.Angle_in - line.Angle_out) )

    line.IN  = Connector( initialize=
        { "Power": -line.Power, "Angle": line.Angle_in } )

    line.OUT = Connector( initialize=
        { "Power":  line.Power, "Angle": line.Angle_out } )
```

Using "model" as the first argument to a rule is actually a misnomer: the first argument is the *owning block*.

Connectors are a map of unique names to expressions.

Arbitrary inputs: conservation blocks

```
def dc_bus_rule(bus, id):
    bus.D      = Param()

    bus.Angle  = Var()
    bus.Power  = VarList()

    def _power_balance(bus, P):
        return summation(P) == bus.D

    bus.BUS = Connector( initialize={ “Angle”: bus.Angle })
    bus.BUS.add( bus.Power, "Power", aggregate=_power_balance )
```



A VarList defines a unique local variable for every connection

Connectors may be assembled programmatically as well as through the `initialize` keyword.

The aggregation rule is called *after* expanding all connections, and is passed the expanded VarList

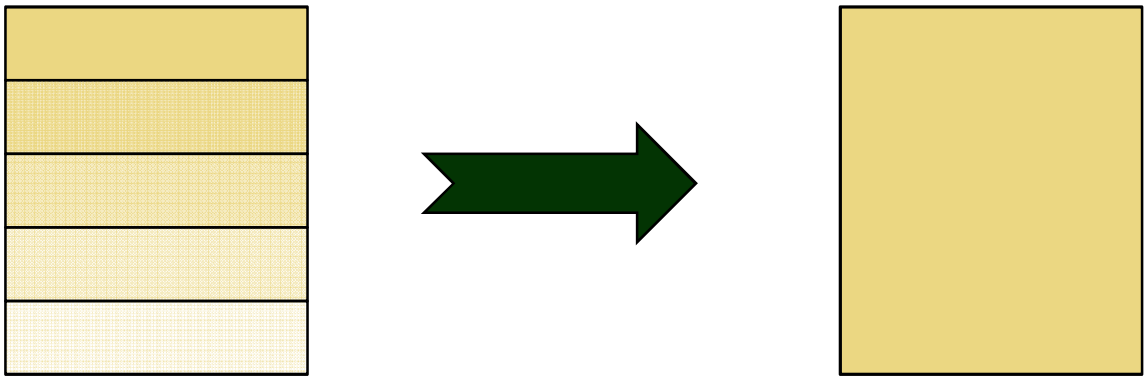
General power flow model

```
from power_flow import \
        dc_line_rule as line_rule, \
        dc_bus_rule as bus_rule, \
        dc_generator_rule as generator_rule

model.BUSES      = Set()
model.LINES      = Set()
model.GENERATORS = Set()

model.links      = Param( model.LINES, ['IN', 'OUT'] )
model.bus        = Block( model.BUSES,      rule=bus_rule )
model.line       = Block( model.LINES,      rule=line_rule )
model.generator  = Block( model.GENERATORS, rule=generator_rule )

def _network(model, l):
   yield model.line[l].IN  == model.bus[ value(model.links[l, 'IN' ]) ].BUS
   yield model.line[l].OUT == model.bus[ value(model.links[l, 'OUT']) ].BUS
   yield ConstraintList.End
model.network = ConstraintList( model.LINES, rule=_network )

def _generator_placement(model, g):
   return model.generator[g].OUT == model.bus[ value(model.generator[g].bus) ].BUS
model.generator_placement = Constraint( model.GENERATORS, rule=_generator_placement )
```

Only domain-specific component
(Note: we have only shown the line and bus rules and not the generator rule)

So, what's really happening?

1) Construct hierarchical model
   - Generate blocks (Variables + Internal constraints)
   - "Connect" blocks by forming constraints over block connectors
2) An automatic *model transformation* "flattens" the model
   - Replicates connector constraints for each variable in connector
   - Generates aggregating constraints
   - (Eliminates redundant variables)



Leveraging components: AC power flow

```
from power_flow import \
        ac_line_rule as line_rule, \
        ac_bus_rule as bus_rule, \
        ac_generator_rule as generator_rule
```

```
model.BUSES      = Set()
model.LINES      = Set()
model.GENERATORS = Set()

model.links      = Param( model.LINES, ['IN', 'OUT'] )
model.bus        = Block( model.BUSES,      rule=bus_rule )
model.line       = Block( model.LINES,      rule=line_rule )
model.generator  = Block( model.GENERATORS, rule=generator_rule )

def _network(model, l):
   yield model.line[l].IN  == model.bus[ value(model.links[l, 'IN' ]) ].BUS
   yield model.line[l].OUT == model.bus[ value(model.links[l, 'OUT']) ].BUS
   yield ConstraintList.End
model.network = ConstraintList( model.LINES, rule=_network )

def _generator_placement(model, g):
   return model.generator[g].OUT == model.bus[ value(model.generator[g].bus) ].BUS
model.generator_placement = Constraint( model.GENERATORS, rule=_generator_placement )
```

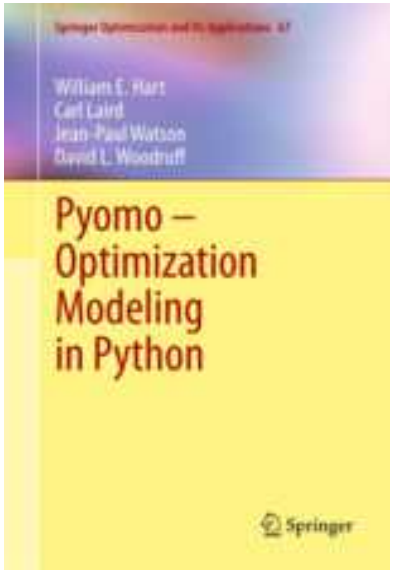
13. Critiquing Coopr

Q: Has Coopr effectively leveraged Python to support a robust modeling and analysis environment?

Feature/Capability	Rating
Documentation	
Solver customization	
Custom solvers	
Model interrogation	
Custom data setup	
Robust library	
Representation management	
Language robustness and stability	
Portability	

Documentation

Coopr/Pyomo:



- Hart, Laird, Watson and Woodruff. *Pyomo – Optimization Modeling in Python*. Springer, 2012.
- *Getting Started with Coopr (html/pdf/eBook)*
- *Coopr Installation Guide (html/pdf/eBook)*
- Hart, Watson and Woodruff. *Pyomo: Modeling and solving mathematical programs in Python*. MPC 3(2), 2011.
- Watson, Woodruff and Hart. *PySP: Modeling and solving stochastic programs in Python*. MPC 4(2), 2012.

Python:

- There are many excellent Python books. New users can usually pick up the basics of Python quite easily.
- Supports robust testing environment for modeling examples

Solver Customization

Idea: use call-backs to customize solver behavior

Example: Dippy (PuLP)

- The DIP solver supports branch-and-cut
- Call-back functions can be registered with the problem object
  - Generate constraints, heuristic solver, etc.

Coopr:

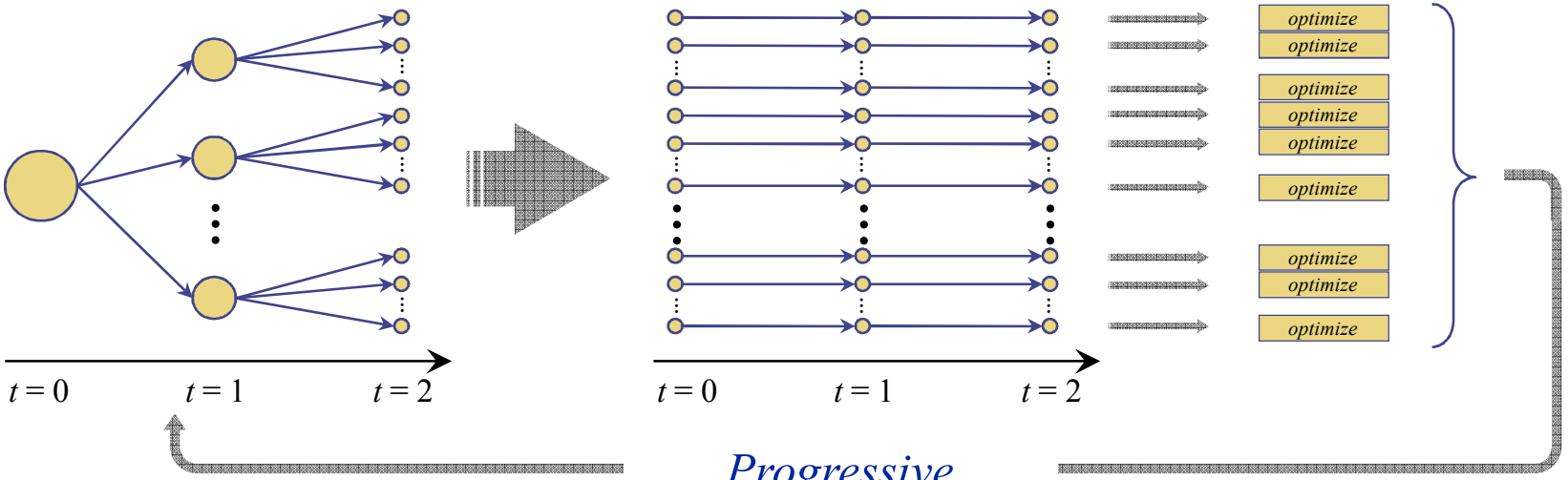
- Has limited support for direct solver interfaces
  - Does not currently interface with DIP
  - Requires library-level linking of Python and C/C++
- Direct solver interfaces significantly complicate the installation of Coopr

Custom Solvers

Python: makes it “easy” to develop new solvers!

Progressive Hedging:

- Heuristic for multi-stage stochastic programming
- Scenario-level decomposition is used
- Sub-problems are easily setup and solved within Coopr



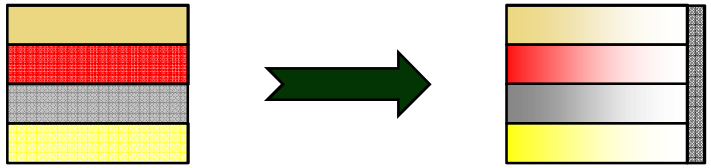
Progressive Hedging

Custom Solvers (cont)

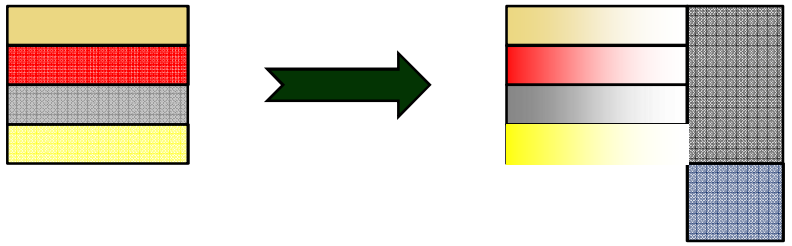
Disjunctive Programming:

- New Pyomo modeling components used to define disjunctive blocks
  - Disjunct: declares a model block
  - Disjunction: a constraint for enforcing disjunction

- Automated transformations generate MIP formulations
  - Big-M relaxation



  - Convex hull relaxation



Model Interrogation

Post-optimization analysis:

- May require many different values generated by the optimizer
  - Value, reduced cost, dual value, etc.
- User needs to interrogate the model to access these values

Suffixes:

- Associate values with variable and constraint components
- Very natural use of Python syntax

Coopr:

- Solver suffixes are supported
- User can specify the suffixes that are needed
- Inconsistent support of suffixes (by solvers) is an issue for users

Custom Data Setup

Python:

- Scripting is easy, and it facilitates the creation of complex models from native Python data types

Coopr:

- Currently supports scripting with ModelData objects
- New DataPortal objects will have much broader capabilities

DataPortal Objects:

- Support loading and storing data
- I/O for set, parameter, variables, suffix data, etc.
- Explicitly manage connections to data sources
- Initialize both abstract and concrete models
- Data sources: csv, tab, odbc, xls/xlsm/xlsb/xlsx, yaml, json, xml, and Pyomo data command files

Data Portal Example: Abstract Model

```
M = AbstractModel()
M.A = Set(dimen=2)
M.q = Param(M.A)
M.p = Param(M.A)

dp = DataPortal()
dp.load(filename='foo.csv', param=(M.p,M.q), index=M.A)

Instance = M.create(dp)
```

foo.csv

```
A1,A2,p,q
a,b,1,2
c,d,3,4
```

Data Portal Example: Concrete Models

```
dp = DataPortal()
dp.load(filename='foo.csv', param='q', index='A')
db.load(filename='bar.tab', param='p')

M = ConcreteModel()
M.A = Set(dimen=2, initialize=db.data('A'))
M.q = Param(M.A, initialize=db.data('q'))
M.p = Param(M.A, initialize=db.data('p'))
Instance = M.create()
```

foo.csv

```
A1,A2,q
a,b,2
c,d,4
```

bar.tab

```
A1,A2,p
a,b,1
c,d,3
```

Robust Library

Coopr:

- Initial development focused on end-user command: pyomo
- Coopr libraries are not particularly user-friendly

Functors:

- Defined using a Python function with a standardized definition
  - Facilitates plug-n-play
- Function docstrings declare the function API
- Global registration and factory

Water Security Toolkit:

- A proof-of-concept for embedding Coopr in another package
- Strongly leveraged modular software design in Coopr

Coopr Functors (2)

```
@coopr_api
def f1(data, x=0, y=1):
    """A simple example.

    Required:
        x: A required keyword argument

    Optional:
        y: An optional keyword argument

    Return:
        a: A return value
        b: Another return value
    """
    return CooprAPIData(a=2*data.z, b=x+y)
```

```
g = CooprAPIFactory('f1')

data = CooprAPIData(z=1)

val = g(data, x=2)
```

Hart, Siirola, Watson, Woodruff, p. 131

Coopr Functors (3)

Functor Examples

- apply_optimizer:

  Perform optimization with a concrete instance
- apply_postprocessing:

  Apply post-processing steps.
- apply_preprocessing:

  Execute preprocessing files
- create_model:

  Create instance of Pyomo model.
- process_results:

  Process optimization results.
- run_command:

  Execute a function that processes command-line arguments
- setup_environment:

  Setup Pyomo execution environment

Hart, Siirola, Watson, Woodruff, p. 132

Representation Management

Idea: customize format/structure of data representation for a model

- Can leverage user knowledge of application space
- Can explicitly manage memory footprint of the model

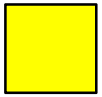
Examples:

- Dense or sparse matrix representation (MIPs)
- Expression tree (NLPs)

Coopr:

- The pyomo command provides limited control of the model representation
- A library of problem transformation functions is being developed
- Model I/O needs to be consolidated to improve code robustness

Language Robustness and Stability



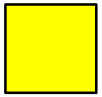
Python:

- The core language is stable, but there have been some major evolutions during Coopr development (2.x -> 3.x)
- New Python implementations are emerging, which complicates the support for Coopr
  - Cpython — Standard C implementation
  - IronPython — .Net
  - PyPy — Python written in Python
  - Jython — Java

Coopr:

- Works with Python 2.6 and 2.7
- Supporting newer versions of Python (3.x) may be difficult

Portability



Coopr:

- Python is available on virtually any platform
- Works with Python 2.6 and 2.7 with CPython
- Works in Mac OSX, MS Windows (Vista, Windows 7), Linux (various)

Installation Challenges:

- Python packages that depend on C libraries are not easily distributed
  - Scipy, pyyaml, pyipopt, etc.
- Need to support installers that are aware of subversion
  - For developers and bleeding-edge users
- Need to install third-party packages, which may not be robustly distributed

Installers, Installers, Installers

MS Windows

- Reworked to support both Python 2.6 and 2.7
- Does not modify the PATH environment

Linux/Unix

- **coopr_install** : off-line installation of latest release
- **coopr_votd** : off-line installation of VOTD snapshot
- **coopr_votd** : the –-trunk option can be used to install from subversion

PyPI

- The Python easy_install command can be used to install Coopr from the internet:
  - easy_install Coopr

Recap

Note: These issues similarly impact other open source modeling frameworks.

Feature/Capability	Rating
Documentation	
Solver customization	
Custom solvers	
Model interrogation	
Custom data setup	
Robust library	
Representation management	
Language robustness and stability	
Portability	

12. Advanced Topics

- Lots of things that we would like to talk about
  - But we assume we're already out of time

- Examples of advanced topics
  - External data sources
  - Parallelization with Pyro
  - Non-linear programming and the AMPL solver library interface
  - External functions
  - Scripting for complex workflows
  - …

Thanks! Please Contribute!

- Now that you actually know something about Pyomo
- Please go off and do great things
- We welcome contributions
  - Bug reports
  - Extensions
  - Links to your projects that use Coopr/Pyomo
  - Source code
  - Documentation
  - …