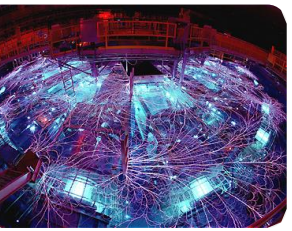


Sandia National Laboratories

Exceptional service in the national interest

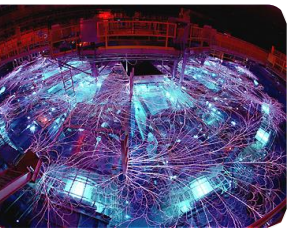
Pyomo Workshop – December 2023

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525



Sandia National Laboratories

Exceptional service in the national interest

Section 0
Preliminaries & Installing Pyomo

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration | CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

Download workshop slides, examples, and exercises

- https://github.com/Pyomo/pyomo-tutorials

To run the workshop exercises you will need:

- Anaconda Python distribution
  - Includes several packages for scientific computing already
  - Supports easy installation of Pyomo and solvers

- Python packages and solvers:
  - Pyomo
  - Glpk (solver for linear and mixed-integer linear problems)
  - Ipopt (solver for continuous nonlinear problems)
  - Idaes-pse (model debugging tools, Ipopt binaries with HSL for most platforms)

Installation Instructions

1. Install Anaconda:

- Download the installer from: https://www.anaconda.com/download
- Follow the installation instructions for your OS: https://docs.anaconda.com/free/anaconda/install/

Installation Instructions

2. Test the Python installation:

- On Windows, you will access Anaconda's Python from the "Anaconda Prompt" application. On the mac, you will use the terminal application. Both of these are command-line interfaces that allow you to run Python code. Open "Anaconda Prompt" or a terminal app.
- Type "python --version", and you should see something similar to:
  Python 3.11.5
- Type "python". You should see something similar to:
  Python 3.11.5 | packaged by Anaconda, Inc. | (main, Sep 11 2023, 13:26:23) [MSC v.1916 64 bit (AMD64)] on win32
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
- Try writing some Python code to test, e.g.,
  >>> print('Hello world!')
- Type "exit()" if everything is working correctly.

Installation Instructions

3. Install Pyomo:

- From the prompt (terminal application on Mac or Anaconda Prompt on Windows) type: “conda install -c conda-forge pyomo”.
- Test your pyomo installation with “pyomo --version”. You should see something similar to the following:
  Pyomo 6.6.2 (CPython 3.11.5 on Windows 10)

Installation Instructions

4. Install glpk:

- From the prompt, type: “conda install -c conda-forge glpk”
- Test the glpk solver. Type “glpsol --version”. You should see something like:
  GLPSOL--GLPK LP/MIP Solver 5.0
  Copyright (C) 2000-2020 Free Software Foundation, Inc. …

5. Install Ipopt:

- From the prompt type: “conda install -c conda-forge ipopt”
- Test the Ipopt solver. Type “ipopt --version”. You should see something like:
  Ipopt 3.13.2 (x86_64-w64-mingw32), ASL(20190605)

Installation Instructions

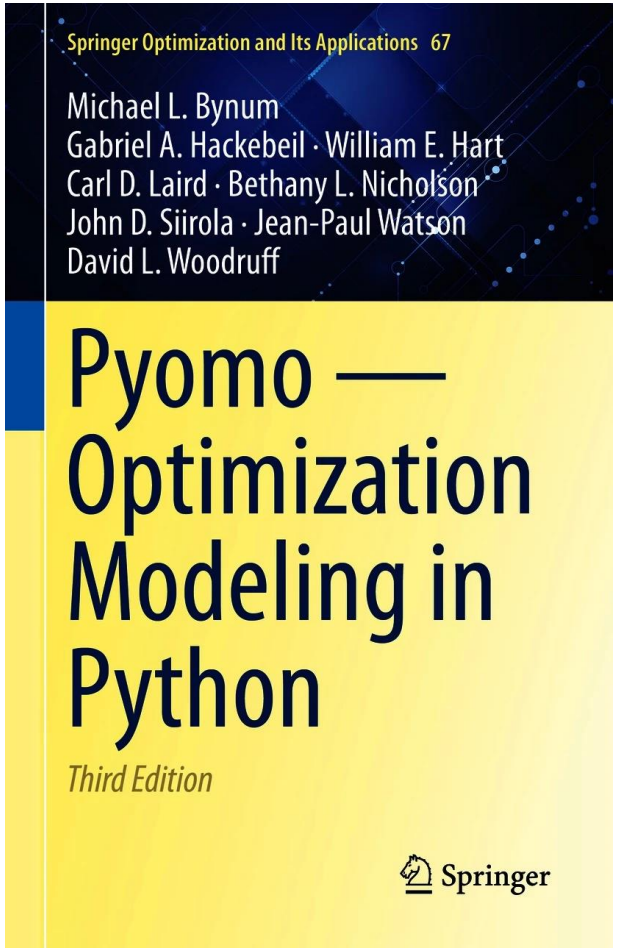
6. Install IDAES-PSE:

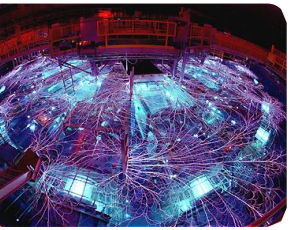
a. From the prompt type: “pip install idaes-pse”

b. Test the idaes install. Type “idaes --version”. You should see something like: idaes, version 2.2.0

Resources

- GitHub:
  - https://github.com/Pyomo
- Documentation:
  - https://pyomo.readthedocs.io
- Getting help:
  - StackOverflow: “pyomo” tag
  - Pyomo Forum: pyomo-forum@googlegroups.com





Optional Material

Sandia National Laboratories

Exceptional service in the national interest

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

Installation instructions using pip instead of conda

- Install Python
  - Linux & Mac OS/X typically have Python pre-installed
  - Scientific Python distributions have many utilities pre-installed
    - http://www.scipy.org/install.html
- Install Pyomo
  - Install in your system
    - `pip install pyomo`
  - (or) Install in a user directory
    - `pip install --user pyomo`
- Installing solvers
  - When using pip, solvers will need to be installed manually (or use the NEOS server)

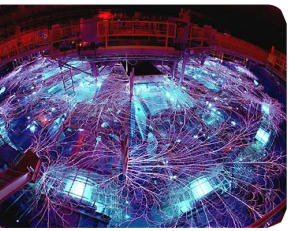
Other open source solver resources

- IDAES-PSE Binary Distribution (e.g., Ipopt+HSL, COIN-OR solvers)
  - https://idaes-pse.readthedocs.io/en/latest/tutorials/getting_started/binaries.html#binary-packages
  - Note: Some of the macOS binaries do not include HSL
- COIN-OR Binary Distributions (e.g., CBC, Ipopt)
  - http://www.coin-or.org/download/binary/
  - Note: Coin-Binary requires the Coin-HSL Archive (Personal License)
    - https://licences.stfc.ac.uk/product/coin-hsl
    - The Personal License permits commercial use ***but not redistribution***
- GLPK
  - http://ftp.gnu.org/gnu/glpk/
- SCIP
  - https://www.scipopt.org/index.php#download

Note: You may need to add the solver installation to the PATH environment variable

License

- Pyomo is released under the 3-clause BSD license
  - No restrictions on deployment or commercial use
  - https://github.com/Pyomo/pyomo/blob/main/LICENSE.md



Sandia National Laboratories

Exceptional service in the national interest

Section 1
Overview of Pyomo

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration | CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

What is “Pyomo”?

- “An open source object-oriented algebraic modeling language in Python for structured optimization problems.”
  - A Python package
  - A set of objects, classes, methods, and utilities for expressing optimization problems
  - A growing collection of utilities for manipulating optimization problems
  - A set of interfaces to common optimization solvers / search routines
  - The base of many other domain-specific optimization-centric modeling packages

- … and a really cool origami bird

Thanks, Doug Prout!

Optimization Modeling Environments / Algebraic Modeling Languages

- Goals:
  - Provide a natural syntax to describe mathematical models
    - Use structures and expressions that match our understanding of the system
    - Formulate large models with a concise syntax
  - Interface with numerous solvers
    - Provide standardized interface machinery (e.g., compiled forms, automatic differentiation)
    - Manage the translation from *what the user said* to *what the solver understands*
  - Separate modeling and data declarations
    - Enable data import and export in commonly used formats

- Examples:
  - Domain-specific languages
    - AMPL, GAMS, AIMMS
  - Language extensions / object libraries
    - FlopCPP (C++), OptimJ (Java), JuMP (Julia), PuLP (Python), Pyomo (Python)

Three really good questions:

- Why another Algebraic Modeling Language (AML)?
  - Existing AMLs made it challenging to express high-level model structure:
    - Composition, logic, dynamics, multi-level optimization
  - We wanted to explore new algorithms and approaches:
    - Decomposition, relaxations, model reformulations, iterative analysis algorithms
  - We wanted to build domain-specific optimization libraries
    - Electric grid model libraries, process model libraries, specialized tools for asset scheduling
- Why Python?
  - At the time? It was an interesting full-featured emerging language
  - Now? Python is a standard environment for analysis and data processing
- Why open-source?
  - Ease of deployment … distributed under the 3-clause BDS license
  - "Stone Soup" model
    - Make it easier for researchers to make their innovations available to the community (and us)

What does Pyomo let you do?

- Declare *models*, containing *variables, parameters, and sets*
  - Variables and parameters may be *scalars* or *indexed*
- Express *algebraic constraints* and *objectives*
  - We refer to this as "*equation-oriented (EO) modeling*"
    - ***All*** model equations must be explicitly provided (e.g., glass-box)
    - *Not*[*] for optimizing an existing simulation (e.g., not black-box)
  - The constraints must be satisfied at any valid solution

```
import pyomo.environ as pyo

m = pyo.ConcreteModel()
m.x = pyo.Var(initialize=-1.5, bounds=(-2,2))
m.y = pyo.Var(initialize=1.5, bounds=(-2,2))
m.obj = pyo.Objective(expr=
    (m.x - 1)**2 + 100*(m.y - m.x**2)**2 )
m.const = pyo.Constraint(expr=
    m.x**2 + m.y**2 <= 1 )

pyo.SolverFactory("ipopt").solve(m)
print(f"x={pyo.value(m.x)}, y={pyo.value(m.y)}")
```

- Send the model to a wide variety of *commercial* and *open-source* solvers
  - A *solver* will attempt to find valid solutions that minimize (or maximize) the objective
  - Which solver depends on the details of your variables, constraints, and objective(s)
    - Pyomo supports most of the optimization "alphabet soup": LP, MIP, QP, QCQP, NLP, MINLP
    - …and has extensions to support some non-algebraic model types: (P)DAE, GDP, CP, SP

More than *just* mathematical modeling: *Why we use Pyomo*

Sandia National Laboratories

Scripting

- Construct models using native Python data
- Iterative analysis of models leveraging Python functionality
- Data analysis and visualization of optimization results

Composable model libraries

- EGRET (power grid modeling), IDAES-PSE (chemical process modeling), WaterTAP

Model transformations (a.k.a. reformulations)

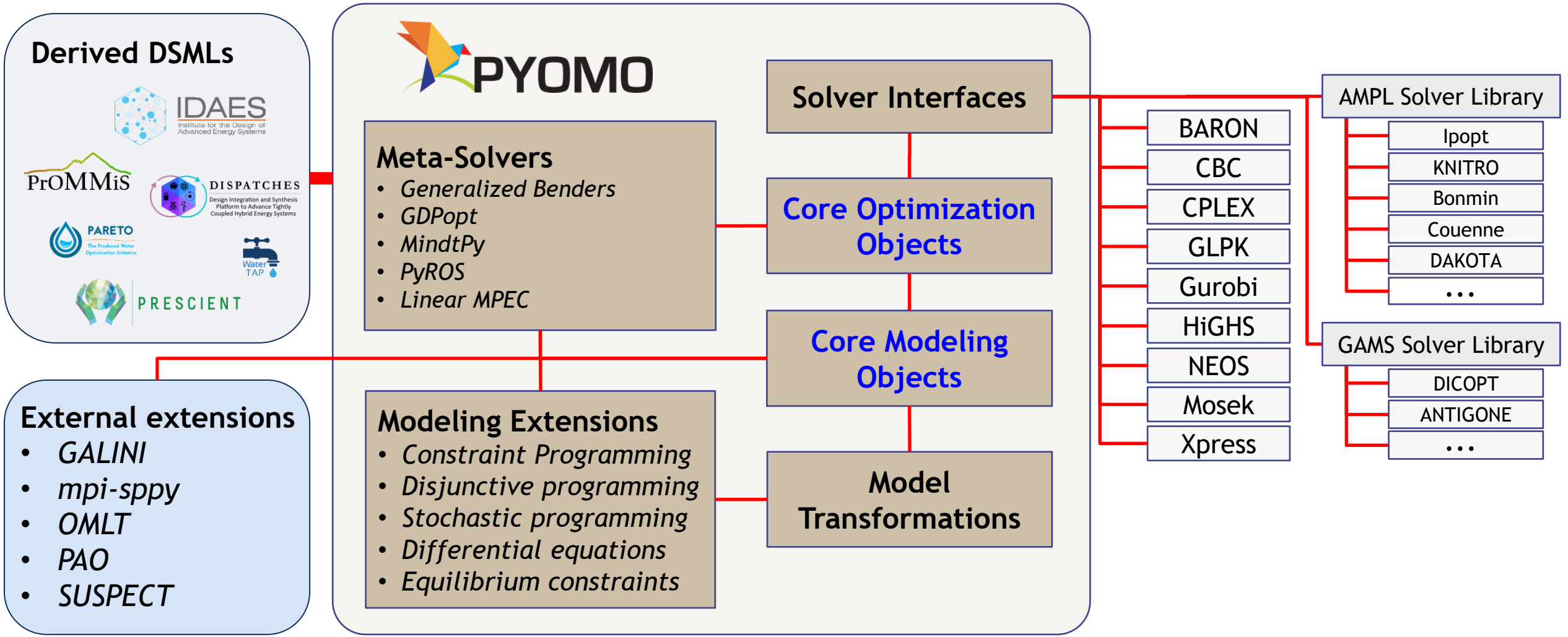
- Automate generation of one model from another
  - DAE → discretized model, GDP → {Big-M, Hull}
- Leverage Pyomo's object model to apply transformations sequentially

Meta-solvers

- Integrate scripting and/or transformations into optimization solver
- Leverage power of Python to build generic capabilities
  - GDPopt, MindtPy, PyROS, SP → {progressive hedging, extensive form} → MIP

CCR Center for Computing Research

Pyomo at a Glance

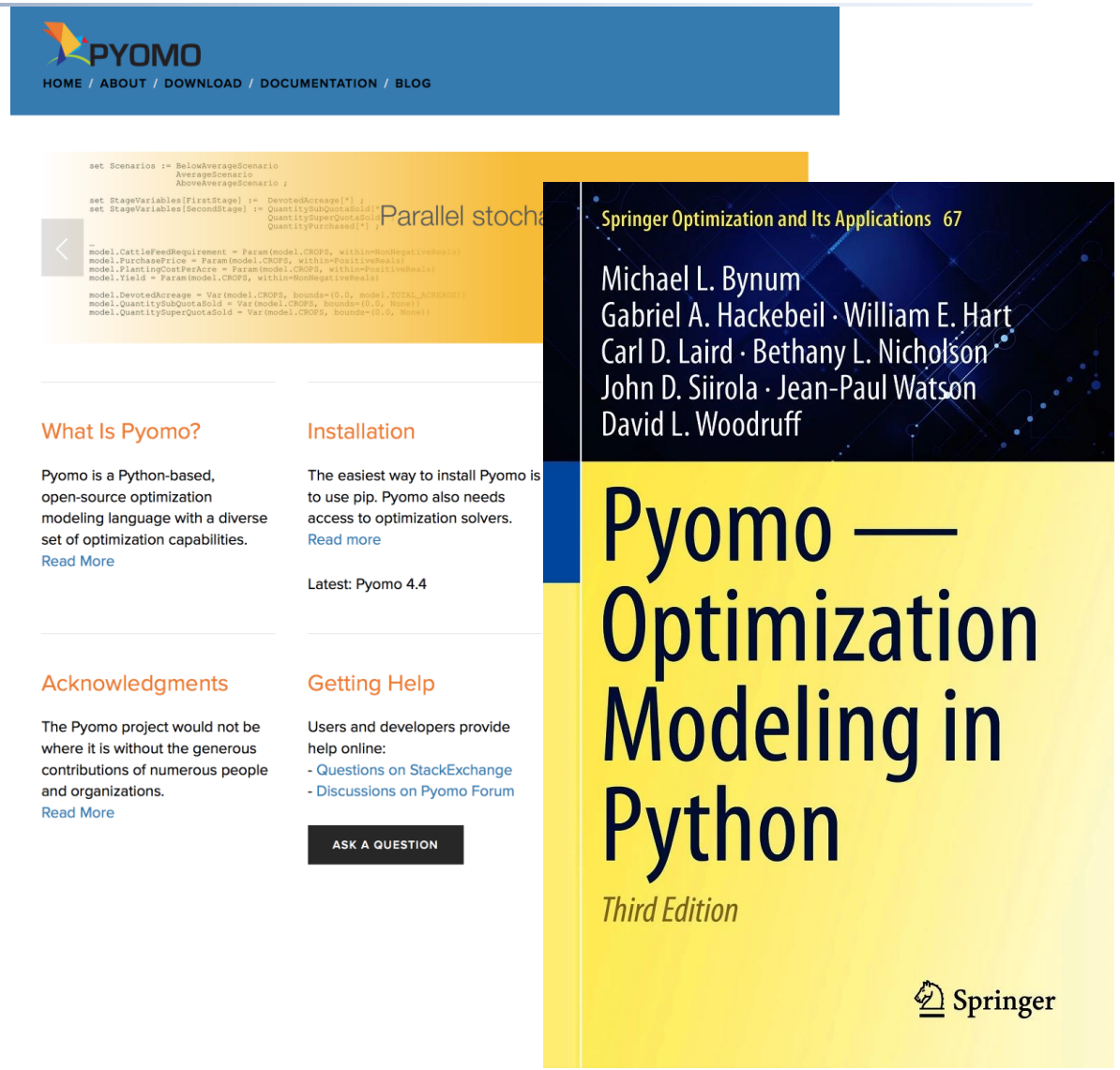


Where do I look for ____?

- Pyomo is a large codebase that has grown organically
  - Finding capabilities, examples, and implementations can be challenging
  - …even developers make heavy use of “recursive grep”
- How to balance between “production tool” and “research environment”?
- Conceptually, three “tiers” in the codebase
  - Core functionality
    - Documented in The Book , ReadTheDocs
    - Stable public API
    - `pyomo.core` - Core modeling objects
    - `pyomo.repn` - Model compilers / writers
    - `pyomo.opt`, `pyomo.solvers` - Solver interfaces
  - Core extensions
    - Extensions the core team has adopted
    - Intent to maintain for the foreseeable future
    - DAE, GDP, Network, MPEC
  - Developmental capabilities / contributed tools, interfaces, examples
    - Support (more) rapid development
    - Relaxed requirements for {documentation, testing, backwards compatibility}
    - pyomo.contrib

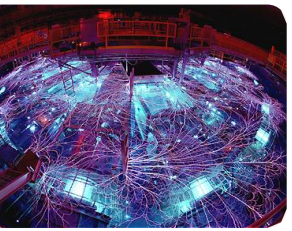
For More Information

- See the Pyomo homepage
  - www.pyomo.org
- Key resources
  - The Book
    - "Pyomo – Optimization Modeling in Python"
    - Documentation of the core API
    - …all examples from The Book, 3rd Ed should run in any 6.x version of Pyomo
  - ReadTheDocs
    - https://pyomo.readthedocs.io
    - Most current online documentation, including contrib / development features and changes since The Book
  - GitHub
    - https://github.com/Pyomo/pyomo
    - Code, issues, developer resources



Acknowledgements

- Core team (past & present)
  - William Hart (SNL)
  - Michael Bynum (SNL)
  - Gabe Hackebeil
  - Emma Johnson (SNL)
  - Carl Laird (CMU)
  - Miranda Mundt (SNL)
  - Bethany Nicholson (SNL)
  - John Siirola (SNL)
  - Jean-Paul Watson (LLNL)
  - Prof. David L. Woodruff (UC Davis)
- Additional contributors
  - Qi Chen
  - Natalie Isenberg
  - Cody Karcher
  - Kate Klise
  - Ben Knueven
  - Zedong Peng
  - Jose Santiago Rodriguez
  - Grant Seastream
  - Jason Sherman
  - Jialu Wang
  - Prof. Roger Wets
- And more than 75 others who have contributed to Pyomo over the years!



Sandia National Laboratories

Exceptional service in the national interest

Section 3
Pyomo Fundamentals

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration | CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

Learning Objectives

- Goals, benefits, drawbacks of Pyomo
- Basic understanding of Python

- Create and solve optimization problems within Pyomo
  - Vars, Constraints, Objectives
  - Sets, Parameters

- Basic scripting, workflow, and programming capabilities
  - Changing data, multiple solutions, importing data, plotting, reporting
  - Pieces to build your own workflows

Mathematical Programming

$$\begin{aligned} \min_{x} \quad & f(x) && \longleftarrow \text{Objective Function} \\ \text{s.t.} \quad & c(x) = 0 && \longleftarrow \text{Equality Constraints} \\ & d^L \leq d(x) \leq d^U && \longleftarrow \text{Inequality Constraints} \\ & x^L \leq x \leq x^U && \longleftarrow \text{Variable Bounds} \end{aligned}$$

- Mathematical programming (i.e. Optimization)
  - Problem types classified according to:
    - Linearity/Nonlinearity of objective and constraints
    - Continuous/Discrete/Mixed variables
  - LP, QP, NLP, MILP, MINLP

Useful for much more than… "optimization"

Simple Modeling Example: Classic Knapsack Problem

- Given a set of items A, with weight and value (benefit)
- Goal: select a subset of these items
  - Maximize the benefit
  - Under weight limit

Item (A)	Weight (w)	Benefit (b)
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

Max weight: 14

Symbol	Meaning
A	set of items available for purchase
b_i	benefit of item i
w_i	weight of item i
$W_{\max}$	max weight that can be carried

Simple Modeling Example: Classic Knapsack Problem

- Variables
- Objectives
- Constraints

- Sets
- Parameters

Symbol	Meaning
A	set of items available for purchase
b_i	benefit of item i
w_i	weight of item i
$W_{\max}$	max weight that can be carried

Item (A)	Weight (w)	Benefit (b)
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

Max weight: 14

$$\max_{x} \quad \sum_{i \in A} b_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x_i \in \{0,1\} \qquad \forall\, i \in A$$

Anatomy of a Concrete Pyomo Model

```
import pyomo.environ as pyo

A = ['hammer', 'wrench', 'screwdriver', 'towel']
b = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}

W_max = 14

model = pyo.ConcreteModel()
model.x = pyo.Var( A, within=pyo.Binary )

model.value = pyo.Objective(
    expr = sum( b[i]*model.x[i] for i in A ),
    sense = pyo.maximize )

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

opt = pyo.SolverFactory('glpk')
result_obj = opt.solve(model, tee=True)

model.pprint()
```

Import packages

Specify/ Import Data

Can be imported or even applied to model later

Item (A)	Weight (w)	Benefit (b)
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

Define Problem

$$\max_x \quad \sum_{i \in A} b_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x_i \in \{0,1\} \qquad \forall i \in A$$

Solve Problem

Report Results

Pyomo imports and *namespaces*

- Pyomo gathers commonly-used objects in the `pyomo.environ` namespace:

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

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

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

- However, good Python practice dictates one should use an explicit namespace:

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

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

Getting started: the *Model*

```
import pyomo.environ as pyo
```

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

```
model = pyo.ConcreteModel()
```

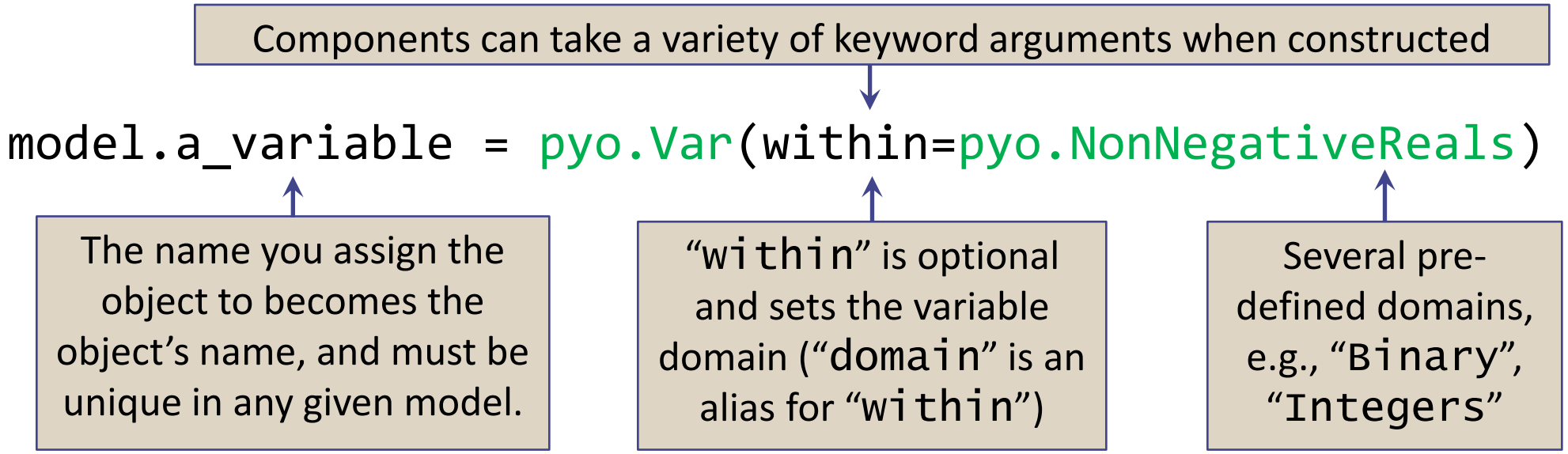
Create an instance of a *Concrete* model

- 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`" or "`m`"

Populating the Model: *Variables*




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

model.a_variable = pyo.Var(initialize=42.0)

model.a_variable = pyo.Var(initialize=42.0, bounds=(0, None))
```

Defining the *Objective*

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

model.obj = pyo.Objective(
    expr=( 1 - model.x )**2 + 100*( model.y - model.x**2 )**2,
    sense=pyo.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 = pyo.Var()
model.b = pyo.Var()
model.c = pyo.Var()
model.c1 = pyo.Constraint(expr = model.b + 5 * model.c <= model.a )
```

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

$$b + 5c \leq a$$

```
model.c2 = pyo.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.

$$a + b \leq 1$$

Higher-dimensional components

- (Almost) All Pyomo *components* can be *indexed*

```
A = [1, 2, 5]
model.x = pyo.Var(A)
model.x[1]  # or model.x[5] but not model.x[3]
```

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

- E.g., Indexed variables

```
A = [1, 2, 5]
B = ['wrench', 'hammer']
model.y = pyo.Var(A, B)

model.y[1, 'wrench'] # or model.y[1,'hammer']
```

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

Note: While indexed variables look like matrices, they are **not**

- In particular, we (mostly) do not support matrix algebra

Generator expressions and list comprehensions

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

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

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

Python *list comprehensions* look and behave very similarly, but are slower as Python must create (and throw away) a `list` object

```
model.c1_slower = pyo.Constraint(
    expr = sum([ model.b[i] for i in model.IDX ]) <= model.a )
```

Putting it all together: Concrete Knapsack

```
# knapsack.py
import pyomo.environ as pyo

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

model = pyo.ConcreteModel()

model.x = pyo.Var( A, within=pyo.Binary )

model.value = pyo.Objective(
    expr = sum( b[i]*model.x[i] for i in A ),
    sense = pyo.maximize )

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

opt = pyo.SolverFactory('glpk')

result_obj = opt.solve(model, tee=True)

model.pprint()
```

- Import packages
- Specify/import data
- Create model object
- Define variable x
- Define objective
- Define constraint
- Create solver
- Solve the problem
- Print results

Putting it all together: Concrete Knapsack

```
# knapsack.py
import pyomo.environ as pyo
. . .
result_obj = opt.solve(model, tee=True)
model.pprint()
```

```
> python knapsack.py
```

```
1 Set Declarations
    x_index : Size=1, Index=None, Ordered=Insertion
        Key  : Dimen : Domain : Size : Members
        None :     1 :    Any :    4 : {'hammer', 'wrench', 'screwdriver', 'towel'}

1 Var Declarations
    x : Size=4, Index=x_index
        Key         : Lower : Value : Upper : Fixed : Stale : Domain
             hammer :     0 :   1.0 :     1 : False : False : Binary
        screwdriver :     0 :   1.0 :     1 : False : False : Binary
              towel :     0 :   1.0 :     1 : False : False : Binary
             wrench :     0 :   0.0 :     1 : False : False : Binary

1 Objective Declarations
    obj : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : maximize : 8*x[hammer] + 3*x[wrench] + 6*x[screwdriver] + 11*x[towel]

1 Constraint Declarations
    weight_con : Size=1, Index=None, Active=True
        Key  : Lower : Body                                                    : Upper : Active
        None :  -Inf : 5*x[hammer] + 7*x[wrench] + 4*x[screwdriver] + 3*x[towel] :  14.0 :   True

4 Declarations: x_index x obj weight_con
```

Putting it all together: Concrete Knapsack

```
# knapsack.py
import pyomo.environ as pyo
. . .
result_obj = opt.solve(model, tee=True)
model.display()
```

```
> python knapsack.py
```

```
  Variables:
    x : Size=4, Index=x_index
        Key         : Lower : Value : Upper : Fixed : Stale : Domain
            hammer :     0 :   1.0 :     1 : False : False : Binary
        screwdriver :     0 :   1.0 :     1 : False : False : Binary
              towel :     0 :   1.0 :     1 : False : False : Binary
            wrench :     0 :   0.0 :     1 : False : False : Binary

  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True :  25.0

  Constraints:
    weight_con : Size=1
        Key  : Lower : Body : Upper
        None :  None : 12.0 :  14.0
```

Pyomo Fundamentals: Exercises #1

1.1. *Knapsack example*: Solve the knapsack problem shown in the tutorial using IDE (e.g., Spyder) or the command line: `>python knapsack.py`. Which items are acquired in the optimal solution? What is the value of the selected items?

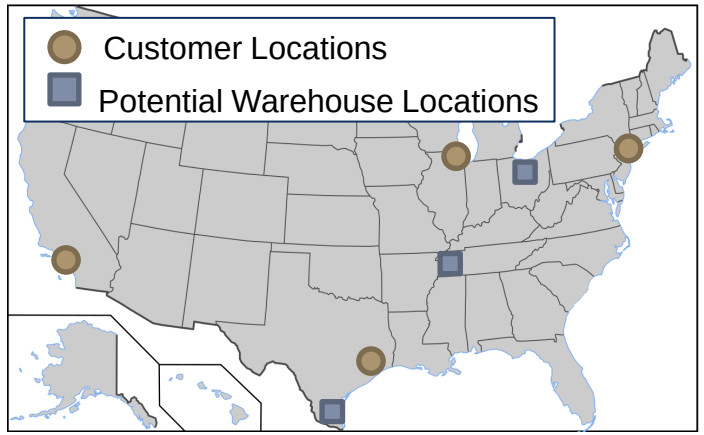
1.2. *Knapsack with improved printing*: The `knapsack.py` example shown in the tutorial uses `model.pprint()` to see the value of the solution variables. Starting with the code in `knapsack_print_incomplete.py`, complete the missing lines to produce formatted output. Note that the Pyomo `value` function should be used to get the floating point value of Pyomo modeling components (e.g., `print(value(model.x[i]))`). Also, print the value of the items selected (the objective), and the total weight. (A solution can be found in `knapsack_print_soln.py`).

1.3. *Changing data*: If we were to increase the value of the wrench, at what point it become selected as part of the optimal solution? (A solution can be found in `knapsack_wrench_soln.py`).

1.4. *Loading data from Excel*: In the knapsack example shown in the tutorial slides, the data is harcoded at the top of the file. Instead of hardcoding the data, use Python to load the data from a different source. You can start from the file `knapsack_pandas_excel_incomplete.py`. (A solution that uses pandas to load the data from excel is shown in `knapsack_excel_soln.py`).

1.5. *NLP vs MIP*: Solve the knapsack problem with **`IPOPT`** instead of `glpk`. (Hint: Switch `glpk` to `ipopt` in the call `SolverFactory`. Print the solution values for `model.x`. What happened? Why?

More Complex Example: Warehouse Location



- Determine the set of P warehouses chosen from the set W of candidates to minimize the cost of serving all customers C
- Here $d_{w,c}$ is the cost of serving customer c from warehouse at location w.

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \le y_w \qquad \forall\ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

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

$x_{w,c}$: fraction of demand of customer c served by warehouse w

y_w: discrete variable (location w selected or not)

Key difference?

Knapsack

$$\max_{x} \sum_{i \in A} v_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x \in \{0,1\}^{|A|}$$

$$w_{hammer} x_{hammer} + \cdots + w_{wrench} x_{wrench} \leq W_{\max}$$

$$x_{Har,NYC} + x_{Mem,NYC} + x_{Ash,NYC} = 1$$

$$x_{Har,LA} + x_{Mem,LA} + x_{Ash,LA} = 1$$

$$x_{Har,Chi} + x_{Mem,Chi} + x_{Ash,Chi} = 1$$

$$x_{Har,Hou} + x_{Mem,Hou} + x_{Ash,Hou} = 1$$

Warehouse Location

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

W	C	NYC	LA	Chicago	Houston
Harlingen		1956	1606	1410	330
Memphis		1096	1792	531	567
Ashland		485	2322	324	1236

Key difference?

Knapsack

$$\max_{x} \sum_{i \in A} v_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x \in \{0,1\}^{|A|}$$

Warehouse Location

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

- In Knapsack problem:
  - One single (scalar) constraint
- In Warehouse Location problem:
  - Multiple constraints defined over indices
  - Pyomo uses *construction rules* to generate groups of related constraints

Indexed Constraints: Construction Rules

$$\sum_{w \in W} x_{w,c} = 1 \quad \forall \; c \in C$$

```
W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
model.x = pyo.Var( W, C, bounds=(0,1) )
model.y = pyo.Var( W, within=pyo.Binary )

def one_per_cust_rule(m, c):
    return sum( m.x[w,c] for w in W ) == 1
model.one_per_cust = pyo.Constraint( C, rule=one_per_cust_rule )
```

The "current" index is passed as argument(s) to the rule, and the rule returns the expression for that index

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

Rule is called once for each entry in C!

Construction Rules: Multiple indices

$$x_{w,c} \leq y_w \quad \forall\ w \in W, c \in C$$

```
W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
model.x = pyo.Var( W, C, bounds=(0,1) )
model.y = pyo.Var( W, within=pyo.Binary )

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = pyo.Constraint( W, C, rule=warehouse_active_rule )
```

Each dimension of the indices is a separate argument to the rule

Rule is called once for each entry w in W crossed with every entry c in C!

Construction Rules: Complex logic

$$x_i = \begin{cases} \cos(y_i), & i \text{ even} \\ \sin(y_i), & i \text{ odd} \end{cases} \quad \forall \{i \in N : i <> 0\}$$

```
def complex_rule(model, i):
    if i == 0:
        return pyo.Constraint.Skip
    elif i % 2 == 0:
        return model.x[i] == pyo.cos( model.y[i] )
    return model.x[i] == pyo.sin( model.y[i] )
model.complex_constraint = pyo.Constraint( N, rule=complex_rule )
```

Constraint rules can contain complex logic on the indices and other parameters, but **NOT** on the value of model variables.

More on *Construction Rules*

- 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
- Naming conventions
  - the component name prepended with "_" (`c4` → `_c4`)
  - the component name with "`_rule`" appended (`c4` → `c4_rule`)
  - The component name is the *same* as the rule name
  - Each rule is called "`rule`" (Python implicitly overrides each declaration)
- Abstract models (discussed later) construct the model in two passes:
  - Python parses the model declaration
    - Creating "empty" Pyomo components in the model
  - Pyomo loads and parses external data

See the "decorator notation" approach on the next slide

Decorator Notation

- Reduce redundancy in *rule definitions* using ***Decorators***

```
@model.Constraint(W, C)
def warehouse_active(m, w, c):
    return m.x[w,c] <= m.y[w]
```

- General notation

```
@<block>.<Component>(indices, …, keywords=…)
def my_thing(m, index, …):
    return # …
```

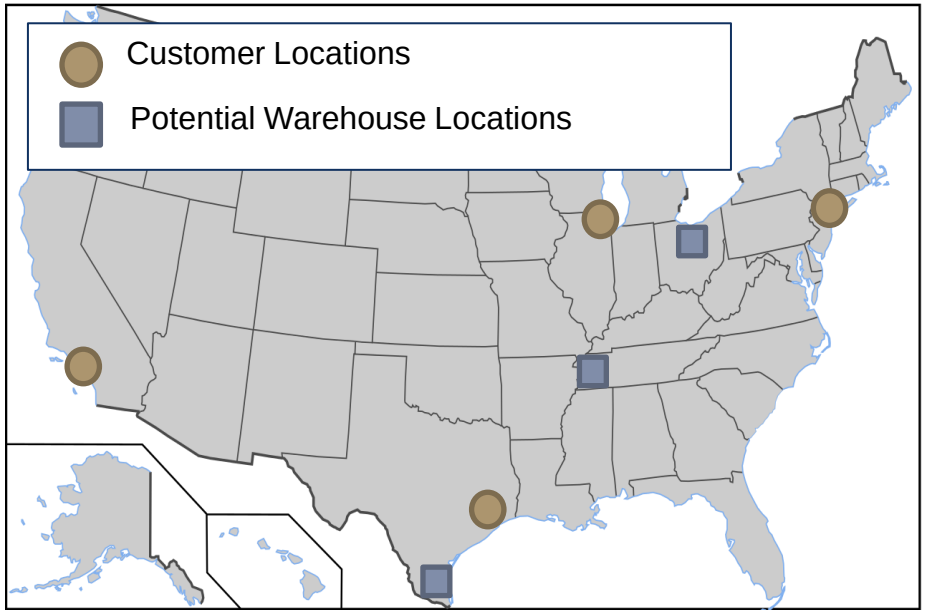
Is equivalent to

```
def my_thing(m, index, …):
    return # …
<block>.my_thing = <Component>(indices, …, rule=my_thing, keywords=…)
```

Note: This syntax *only* works for defining the "`rule`" keyword.

More Complex Example: Warehouse Location

- Determine the set of P warehouses chosen from the set W of candidates to minimize the cost of serving all customers C
- Here $d_{w,c}$ is the cost of serving customer c from warehouse at location w.



W	C	NYC	LA	Chicago	Houston
Harlingen		1956	1606	1410	330
Memphis		1096	1792	531	567
Ashland		485	2322	324	1236

$$P = 2$$

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

Putting It All Together: Warehouse Location

```
import pyomo.environ as pyo

model = pyo.ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = pyo.Var(W, C, bounds=(0,1))
model.y = pyo.Var(W, within=pyo.Binary)

@model.Objective()
def obj(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)

@model.Constraint(C)
def one_per_cust(m, c):
    return sum(m.x[w,c] for w in W) == 1

@model.Constraint(W, C)
def warehouse_active (m, w, c):
    return m.x[w,c] <= m.y[w]

@model.Constraint()
def num_warehouses (m):
    return sum(m.y[w] for w in W) <= P

pyo.SolverFactory('glpk').solve(model)

model.display()
```

$$x_{w,c} \quad \forall\, w \in W, c \in C$$

$$y_w \quad \forall\, w \in W$$

$$\min_{x,y} \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$\sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C$$

$$x_{w,c} \leq y_w \;\forall\, w \in W, c \in C$$

$$\sum_{w \in W} y_w \leq P$$

Pyomo Fundamentals: Exercises #2

2.1. *Knapsack problem with rules*: Rules are important for defining indexed constraints, however, they can also be used for single (i.e. scalar) constraints. Starting with `knapsack.py`, reimplement the model using rules for the objective and the constraints. (A solution can be found in `knapsack_rules_soln.py`.)

2.2. *Integer formulation of the knapsack problem*: Consider again the knapsack problem. Assume now that we can acquire multiple items of the same type. In this new formulation, x_i is now an integer variable instead of a binary variable. One way to formulate this problem is as follows:

$$\max_{q,x} \quad \sum_{i \in A} v_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x_i = \sum_{j=0}^{N} j q_{i,j} \qquad \forall\, i \in A$$

$$0 \leq x \leq N$$

$$q_{i,j} \in \{0,1\}$$

Starting with `knapsack_rules_soln.py`, implement this new formulation and solve. Is the solution surprising? (A solution can be found in `knapsack_integer_soln.py`.)

Parameters

- Pyomo models can be built using standard python numeric types (e.g., int, float) for all constants/data
  - This is how we have done examples so far.

- Pyomo also supports a parameter component (`Param`)
  - Keeps data documented on the model
  - Allows for validation of data, default values, and changes in data without the need to rebuild entire model
  - Allows Abstract model definitions (declare model, apply data later)

- Scalar numeric values

Provide an (initial) value of 42 for the parameter

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

Parameters

- Indexed numeric values

```
model.a_param_vec = pyo.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

- Mutable Parameters

```
model.a_parameter = pyo.Param( initialize = 42,
                               mutable = True )
```

Indicates to Pyomo that you may want to change this parameter later.

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 = pyo.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 = pyo.Set( [1,2,5] )`

Note: This does not mean what you think it does.
This creates a 3-member *indexed set*, where each set is *empty.*
    ```

Sequential Indices: *RangeSet*

- Sets of sequential integers are common
  - `model.IDX = pyo.Set( initialize=range(5) )`
  - `model.IDX = pyo.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 = pyo.RangeSet( 0, 4 )`

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

Manipulating Sets

- Creating sparse sets

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

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

- Sets support efficient higher-dimensional indices

```
model.IDX = pyo.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 (^)

Higher-dimensional Sets and Flattening

- Higher-dimensional sets contain multiple indices per element

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

tuples = list()
for i in model.IDX:
    for j in model.IDX:
        tuples.append( (i,j) )
model.IDXT = pyo.Set( initialize=tuples )
```

- IDX2 ≡ IDXT ≡ [(1,1), (1,2), (1,5),
  (2,1), (2,2), (2,5),
  (5,1), (5,2), (5,5)]

- Higher-dimensional sets and rules

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

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

Set ordering

- By default, Sets are *ordered* (by insertion order, just like Python dictionaries)

```
model.IDX = pyo.Set( initialize=[1,2,5] )
for i in model.IDX:
    print(i)
```

No specific order guaranteed for this loop (e.g., 1,5,2 … 5,1,2 …)

```
model.x = pyo.Var(model.IDX)
for k in model.x:
    print(value(model.x[i]))
```

This is also true for variables indexed by unordered sets.

Use the keyword argument `ordered=False` to relax ordering fixed order.

```
model.IDXO = pyo.Set( initialize=[1,2,5], ordered=True )
```

Other Modeling Components

- Pyomo supports “list”-like indexed components (useful for meta-algorithms and addition of cuts)

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

Advanced Modeling Concepts: non-finite Sets

- Pyomo Sets can represent both finite and non-finite ranges
  - (useful for meta-algorithms and addition of cuts)
  - Components defined over non-finite sets will not have the rule called at construction time

```
model.a = pyo.Var()
model.b = pyo.Var()
model.c = pyo.Var()

model.limits = pyo.Constraint(pyo.PositiveIntegers)
model.limits[1] = 30*model.a + 15*model.b + 10*model.c <= 100
model.limits[2] = 10*model.a + 25*model.b +  5*model.c <= 75
model.limits[3] = 6*model.a  + 11*model.b +  3*model.c <= 30

model.limits[0] = 1*model.a  +  2*model.b +  4*model.c >= 10
[...]
KeyError: "Index '0' is not valid for indexed component 'c'"
```

Advanced, advanced topic: `pyo.Any` is a valid non-finite Set for indexing components

Expression performance tips

- In general, Pyomo is reasonably agnostic to how you express your constraints
  - (although this has not always been the case)

- If you feel that model construction is “too slow,” look for “problematic” constraints
  - You can ask Pyomo to report the construction of individual Pyomo components by running

```
from pyomo.common.timing import report_timing
report_timing()
```

  - (before building your model)

Pyomo Fundamentals: Exercises #3

3.1. *Changing Parameter values*: In the tutorial slides, we saw that a parameter could be specified to be `mutable`. This tells Pyomo that the value of the parameter may change in the future and allows the user to change the parameter value and resolve the problem without the need to rebuild the entire model each time. We will use this functionality to find a better solution to an earlier exercise. Considering again the knapsack problem, we would like to find when the wrench becomes valuable enough to be a part of the optimal solution. Create a Pyomo `Parameter` for the value of the items, make it mutable, and then write a loop that prints the solution for different wrench values. Start with the file `knapsack_mutable_parameter_incomplete.py`. (A solution for this problem can be found in `knapsack_mutable_parameter_soln.py`.)

3.2. *Integer cuts*: Often, it can be important to find not only the "best" solution, but a number of solutions that are equally optimal, or close to optimal. For discrete optimization problems, this can be done using something known as an integer cut. Consider again the knapsack problem where the choice of which items to select is a discrete variable $x_i \; \forall \; i \; \in A$. Let x_i^* be a particular set of x values we want to remove from the feasible solution space. We define an integer cut using two sets. The first set S_0 contains the indices for those variables whose current solution is 0, and the second set S_1 consists of indices for those variables whose current solution is 1. Given these two sets, an integer cut constraints that would prevent such a solution from appearing again is defined by,

$$\sum_{i \in S_0} x[i] + \sum_{i \in S_1} (1 - x[i]) \geq 1$$

Starting with `knapsack_rules_soln.py`, write a loop that solves the problem 5 times, adding an integer cut to remove the previous solution, and printing the value of the objective function and the solution at each iteration of the loop. (A solution for this problem can be found in `knapsack_integer_cut_soln.py`.)

Pyomo Fundamentals: Exercises #3

3.3. ***Putting it all together with lot sizing example***: We will now write a complete model from scratch using a well-known multi-period optimization problem for optimal lot-sizing adapted from Hagen et al. (2001) shown below.

$$\min \quad \sum_{t \in T} c_t\, y_t + h_t^+ I_t^+ + h_t^- I_t^- \qquad (1)$$

$$s.t. \quad I_t = I_{t-1} + X_t - d_t \qquad \forall t \in T \qquad (2)$$

$$I_t = I_t^+ - I_t^- \qquad \forall t \in T \qquad (3)$$

$$X_t \leq P y_t \qquad \forall t \in T \qquad (4)$$

$$X_t, I_t^+, I_t^- \geq 0 \qquad \forall t \in T \qquad (5)$$

$$y_t \in \{0,1\} \qquad \forall t \in T \qquad (6)$$

Parameter	Description	Value
c	Fixed cost of production	4.6
I_0^+	Initial value of positive inventory	5.0
I_0^-	Initial value of backlogged orders	0.0
h^+	Cost (per unit) of holding inventory	0.7
h^-	Shortage cost (per unit)	1.2
P	Maximum production amount (big-M value)	5
d	Demand	[5, 7, 6.2, 3.1, 1.7]

Our goal is to find the optimal production X_t given known demands d_t, fixed cost c_t associated with active production in a particular time period, an inventory holding cost h_t^+ and a shortage cost h_t^- (cost of keeping a backlog) of orders. The variable y_t (binary) determine if we produce in time t or not, and I_t^+ represents inventory that we are storing across time period t, while I_t^- represents the magnitude of the backlog. Note that equation (4) is a constraint that only allows production in time period t if the indicator variables $y_t = 1$.

Write a Pyomo model for this problem and solve it using `glpk` using the data provided below. You can start with the file `lot_sizing_incomplete.py`. (A solution for this problem can be found in `lot_sizing_soln.py`.)

Other Topics

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

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

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)

```
import pyomo.environ as pyo

model = pyo.AbstractModel()

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

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

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

model.x = pyo.Var( model.Locations, model.Customers, bounds=(0.0, 1.0) )
model.y = pyo.Var( model.Locations, within=pyo.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 = pyo.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 = pyo.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 = pyo.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 = pyo.Constraint( rule=num_facilities_rule )
```

Abstract p-Median (pmedian.py, 2)

```
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

$$\max_{x} \quad \sum_{i \in A} v_i x_i$$

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

$$x_i \in \{0,1\} \qquad \forall\, i \in A$$

Item (A)	Weight (w)	Benefit (b)
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

Max weight: 14

Syntax reminders:

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

Abstract Knapsack: *Solution*

```
import pyomo.environ as pyo

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

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

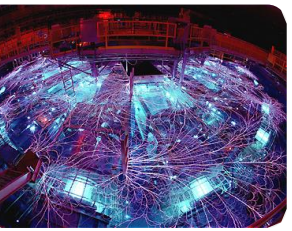
def weight_rule(model):
    return sum( model.w[i]*model.x[i] for i in model.ITEMS ) \
        <= model.W_max
model.weight = pyo.Constraint( rule=weight_rule )
```

Abstract Knapsack: *Solution*

```
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;
```



Sandia National Laboratories

Exceptional service in the national interest

Section 5
Nonlinear Problems

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration | CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

Nonlinear problems are easy…

Nonlinear problems are easy…

… to write in Pyomo (correct formulation and solution is another story)

Nonlinear: Supported expressions

Operation	Operator	Example
multiplication	`*`	`expr = model.x * model.y`
division	`/`	`expr = model.x / model.y`
exponentiation	`**`	`expr = (model.x+2.0)**model.y`
in-place multiplication[1]	`*=`	`expr *= model.x`
in-place division[2]	`/=`	`expr /= model.x`
in-place exponentiation[3]	`**=`	`expr **= model.x`

```
model = pyo.ConcreteModel()
model.r = pyo.Var()
model.h = pyo.Var()

def surf_area_obj_rule(m):
    return 2 * math.pi * m.r * (m.r + m.h)
model.surf_area_obj = pyo.Objective(rule=surf_area_obj_rule)

def vol_con_rule(m):
    return math.pi * m.h * m.r**2 == 355
model.vol_con = pyo.Constraint(rule=vol_con_rule)
```

Nonlinear: Supported nonlinear functions

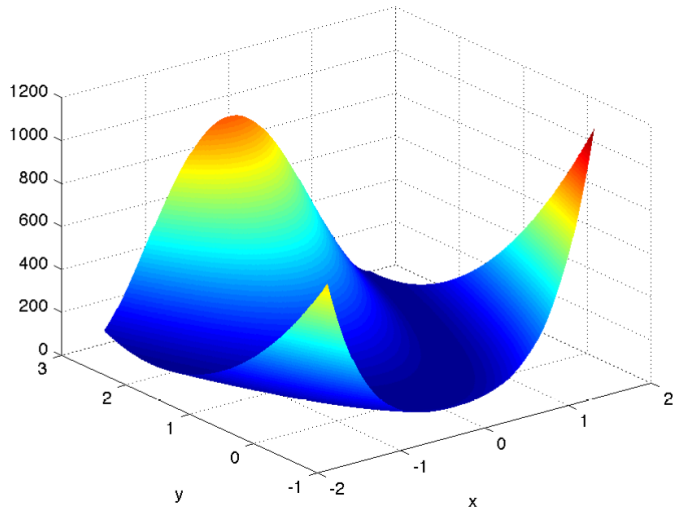
Operation	Function	Example
arccosine	`acos`	`expr = pyo.acos(model.x)`
hyperbolic arccosine	`acosh`	`expr = pyo.acosh(model.x)`
arcsine	`asin`	`expr = pyo.asin(model.x)`
hyperbolic arcsine	`asinh`	`expr = pyo.asinh(model.x)`
arctangent	`atan`	`expr = pyo.atan(model.x)`
hyperbolic arctangent	`atanh`	`expr = pyo.atanh(model.x)`
cosine	`cos`	`expr = pyo.cos(model.x)`
hyperbolic cosine	`cosh`	`expr = pyo.cosh(model.x)`
exponential	`exp`	`expr = pyo.exp(model.x)`
natural log	`log`	`expr = pyo.log(model.x)`
log base 10	`log10`	`expr = pyo.log10(model.x)`
sine	`sin`	`expr = pyo.sin(model.x)`
square root	`sqrt`	`expr = pyo.sqrt(model.x)`
hyperbolic sine	`sinh`	`expr = pyo.sinh(model.x)`
tangent	`tan`	`expr = pyo.tan(model.x)`
hyperbolic tangent	`tanh`	`expr = pyo.tanh(model.x)`

Caution: Always use the intrinsic functions that are part of the Pyomo package.

Example: Rosenbrock function

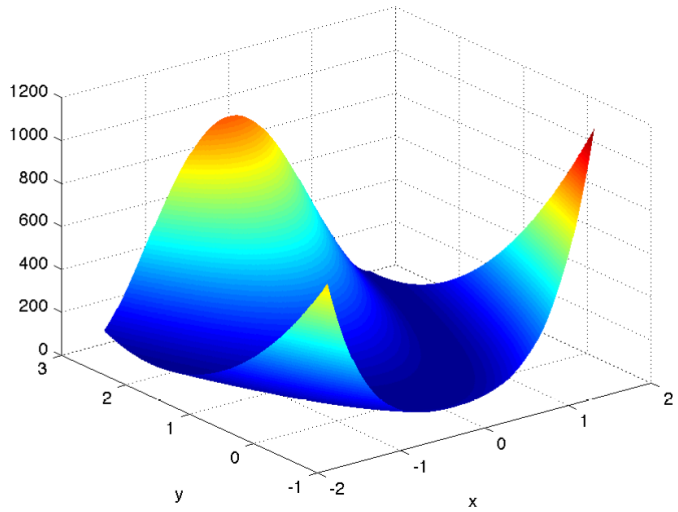
$$\min_{x,y} \; f(x,y) = (1-x)^2 + 100\left(y-x^2\right)^2$$

- Minimize the rosenbrock function using Pyomo and Ipopt
- Initialize at x=1.5, y=1.5



Example: Rosenbrock function

$$\min_{x,y} \; f(x,y) = (1-x)^2 + 100\left(y - x^2\right)^2$$



- Minimize the rosenbrock function using Pyomo and Ipopt
- Initialize at x=1.5, y=1.5

```
# rosenbrock.py: A Pyomo model for the Rosenbrock problem
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.x = pyo.Var(initialize=1.5)
model.y = pyo.Var(initialize=1.5)

def rosenbrock(m):
    return (1.0-m.x)**2 + 100.0*(m.y - m.x**2)**2
model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

pyo.SolverFactory('ipopt').solve(model, tee=True)
model.pprint()
```

Nonlinear: Exercises

1.1 Alternative Initialization: Effective initialization can be critical for solving nonlinear problems, since they can have several local solutions and numerical difficulties. Solve the Rosenbrock example using different initial values for the x variables. Write a loop that varies the initial value from 2.0 to 6.0, solves the problem, and prints the solution for each iteration of the loop. (A solution for this problem can be found in `rosenbrock_init_soln.py`.)

1.2 Evaluation errors: Consider the following problem with initial values x=5, y=5.

$$\min_{x,y} \; f(x,y) = (x-1.01)^2 + y^2$$
$$\text{s.t.} \quad y = \sqrt{x-1.0}$$

(a) Starting with `evaluation_error_incomplete.py`, formulate this Pyomo model and solve using IPOPT. You should get a list of errors from the solver. Add the IPOPT solver option `solver.options['halt_on_ampl_error']='yes'` to find the problem. (Hint: error output might be ordered strangely, look up in the console output.) What did you discover? How might you fix this? (A solution for this can be found in `evaluation_error_soln.py`.)

(b) Add bounds $x \geq 1$ to fix this problem. Resolve the problem. Comment on the number of iterations and the quality of solution. (Note: The problem still occurs because $x \geq 1$ is not enforced exactly, and small numerical values still cause the error.) (A solution for this can be found in `evaluation_error_bounds_soln.py`.)

(c) Think about other solutions for this problem. (e.g., $x \geq 1.001$). (A solution for this can be found in `evaluation_error_bounds2_soln.py`.)

Nonlinear: Exercises

1.3 Alternative Formulations: Consider the following problem with initial values x=5, y=5.

$$\min_{x,y} \; f(x,y) = (x-1.01)^2 + y^2$$
$$\text{s.t.} \quad \frac{x-1}{y} = 1$$

Note that the solution to this problem is x=1.005 and y=0.005. There are several ways that the problem above can be reformulated. Some examples are shown below. Which ones do you expect to be better? Why? Starting with `formulation_incomplete.py` finish the Pyomo model for each of the formulations and solve with IPOPT. Note the number of iterations and quality of solutions. What can you learn about problem formulation from these examples?

(a) (A solution can be found in `formulation_1_soln.py`.)

$$\min_{x,y} \; f(x,y) = (x-1.01)^2 + y^2$$
$$\text{s.t.} \quad \frac{x-1}{y} = 1$$

(b) (A solution for this can be found in `formulation_2_soln.py`.)

$$\min_{x,y} \; f(x,y) = (x-1.01)^2 + y^2$$
$$\text{s.t.} \quad \frac{x}{y+1} = 1$$

(c) (A solution for this can be found in `formulation_3_soln.py`.)

$$\min_{x,y} \; f(x,y) = (x-1.01)^2 + y^2$$
$$\text{s.t.} \quad y = x - 1$$

(d) Bounds and initialization can be very helpful when solving nonlinear optimization problems. Starting with `formulation_incomplete.py` resolve the original problem, but add bounds, $y \geq 0$. Note the number of iterations and quality of solution, and compare with what you found in 1.2 (a). (A solution for this can be found in `formulation_4_soln.py`.)

Nonlinear: Exercises

1.4 Reactor design problem (Hart et al., 2017; Bequette, 2003): In this example, we will consider a chemical reactor designed to produce product B from reactant A using a reaction scheme known as the Van de Vusse reaction:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C$$
$$2A \xrightarrow{k_3} D$$

Under appropriate assumptions, F is the volumetric flowrate through the tank. The concentation of component A in the feed is c_{Af}, and the concentrations in the reactor are equivalent to the concentrations of each component flowing out of the reactor, given by c_A, c_B, c_C, and c_D.

If the reactor is too small, we will not produce sufficient quantity of B, and if the reactor is too large, much of B will be further reacted to form the undesired product C. Therefore, our goal is to solve for the reactor volume that maximizes the outlet concentration for product B.

The steady-state mole balances for each of the four components are given by,

$$\begin{aligned}
0 &= \frac{F}{V}c_{Af} - \frac{F}{V}c_A - k_1 c_A - 2k_3 c_A^2 \\
0 &= -\frac{F}{V}c_B + k_1 c_A - k_2 c_B \\
0 &= -\frac{F}{V}c_C + k_2 c_B \\
0 &= -\frac{F}{V}c_D + k_3 c_A^2
\end{aligned}$$

The known parameters for the system are,

$$c_{Af} = 10\ \frac{\text{gmol}}{\text{m}^3} \quad k_1 = \frac{5}{6}\ \text{min}^{-1} \quad k_2 = \frac{5}{3}\ \text{min}^{-1} \quad k_3 = \frac{1}{6000}\ \frac{\text{m}^3}{\text{mol min}}.$$

Formulate and solve this optimization problem using Pyomo. Since the volumetric flowrate F always appears as the numerator over the reactor volume V, it is common to consider this ratio as a single variable, called the space-velocity SV. (A solution to this problem can be found in `reactor_design_soln.py`.)

OTHER MATERIAL

Introduction to Ipopt

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & d^L \leq d(x) \leq d^U \\ & x^L \leq x \leq x^U \end{aligned}$$

- $f(x)$ ← Objective Function
- $c(x) = 0$ ← Equality Constraints
- $d^L \leq d(x) \leq d^U$ ← Inequality Constraints
- $x^L \leq x \leq x^U$ ← Variable Bounds

$$x \in \Re^n$$
$$f(x) : \Re^n \mapsto \Re$$
$$c(x) : \Re^n \mapsto \Re^m$$
$$d(x) : \Re^n \mapsto \Re^p$$

- f(x), c(x), d(x)
  - general nonlinear functions (non-convex?)
  - Smooth (C^2)
- The x variables are continuous
  - x(x-1)=0 for discrete conditions really doesn't work

Introduction to Ipopt

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & d^L \leq d(x) \leq d^U \\ & x^L \leq x \leq x^U \end{aligned}$$

- $f(x)$ ← Cost/Profit, Measure of fit
- $c(x) = 0$ ← Physics of the system
- $d^L \leq d(x) \leq d^U$ ← Physical, Performance, Safety Constraints
- $x^L \leq x \leq x^U$ ← Physical, Performance, Safety Constraints

$$x \in \Re^n$$

$$f(x) : \Re^n \mapsto \Re$$

$$c(x) : \Re^n \mapsto \Re^m$$

$$d(x) : \Re^n \mapsto \Re^p$$

- f(x), c(x), d(x)
  - general nonlinear functions (non-convex?)
  - Smooth (C^2)
- The x variables are continuous
  - x(x-1)=0 for discrete conditions really doesn't work

Large Scale Optimization

- Gradient Based Solution Techniques

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & x \geq 0 \end{aligned}$$

$$\begin{aligned} \nabla f(x) + \nabla c(x)^T \cdot \lambda - z &= 0 \\ c(x) &= 0 \\ X \cdot z &= 0 \\ (x \geq 0, z &\geq 0) \end{aligned}$$

Newton Step

$$\begin{bmatrix} W_k & \nabla c(x_k) \\ \nabla c(x_k)^T & 0 \end{bmatrix} \begin{pmatrix} \Delta x \\ \Delta \lambda \end{pmatrix} = - \begin{bmatrix} \nabla f(x_k) + \nabla c(x_k)^T \lambda_k \\ c(x_k) \end{bmatrix}$$

$$\left(W_k = \nabla^2_{xx} \mathcal{L} = \nabla^2_{xx} f(x_k) + \nabla^2_{xx} c(x_k) \lambda \right)$$

Active-set Strategy

Interior Point Methods

Original NLP

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & x \geq 0 \end{aligned}$$

Barrier NLP

$$\begin{aligned} \min_x \quad & f(x) - \mu \cdot \sum_i ln(x_i) \\ \text{s.t.} \quad & c(x) = 0 \end{aligned}$$

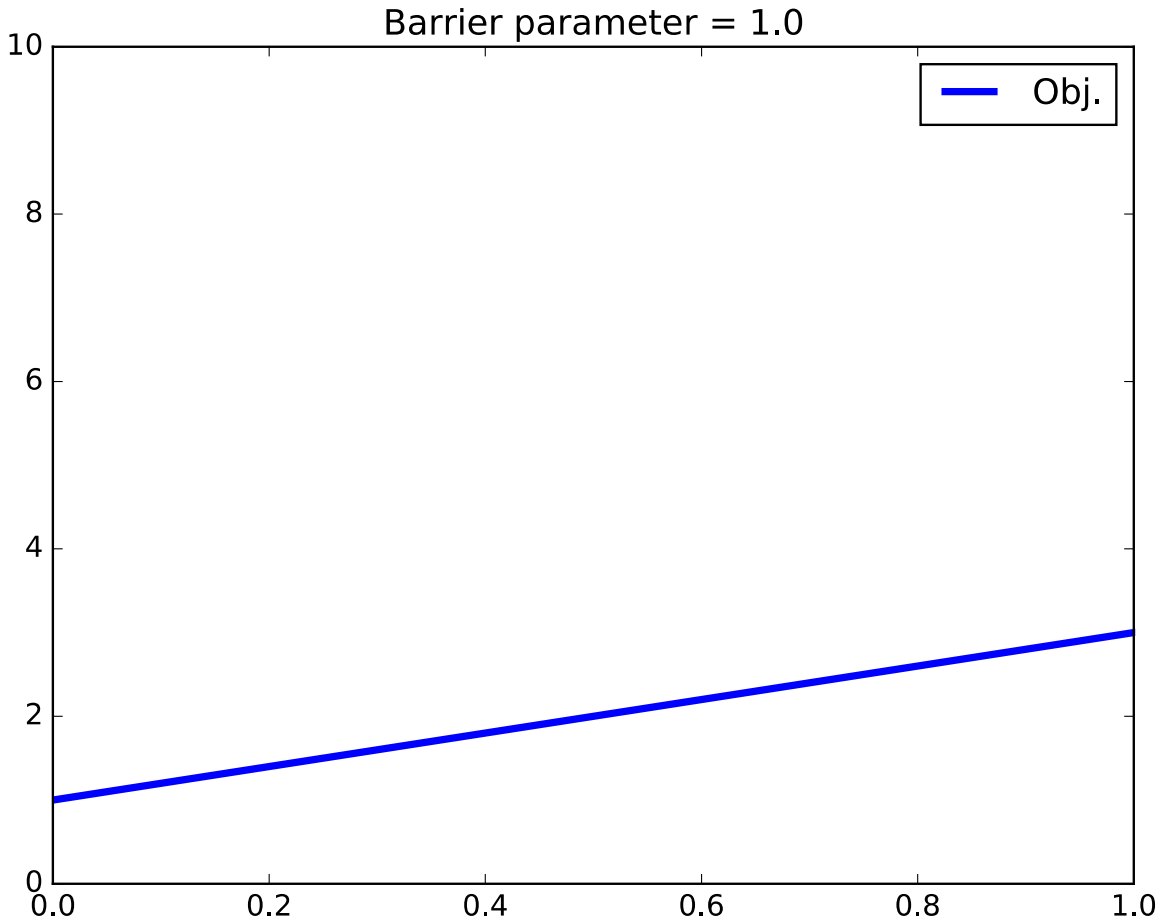
as $x \to 0, \quad ln(x) \to \infty$

as $\mu \to 0, \quad x^\star(\mu) \to x^\star$

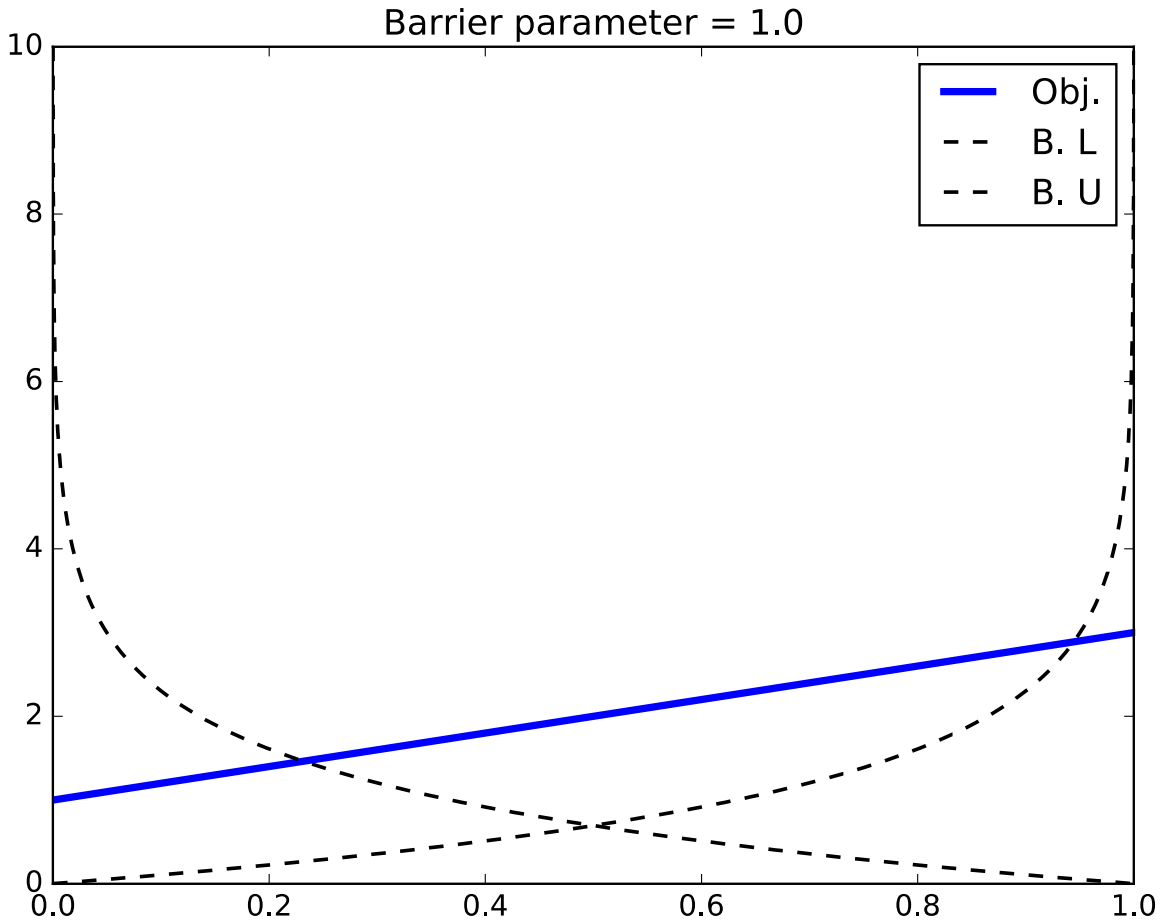
Fiacco & McCormick (1968)

- Initialize
  $x_0 > 0, \quad \mu_0 > 0, \quad \text{Set } l \leftarrow 0$
- Solve Barrier NLP for μ_l
- Decrease the barrier parameter
  $\mu_{l+1} < \mu_l$
- Increase $l \leftarrow l + 1$

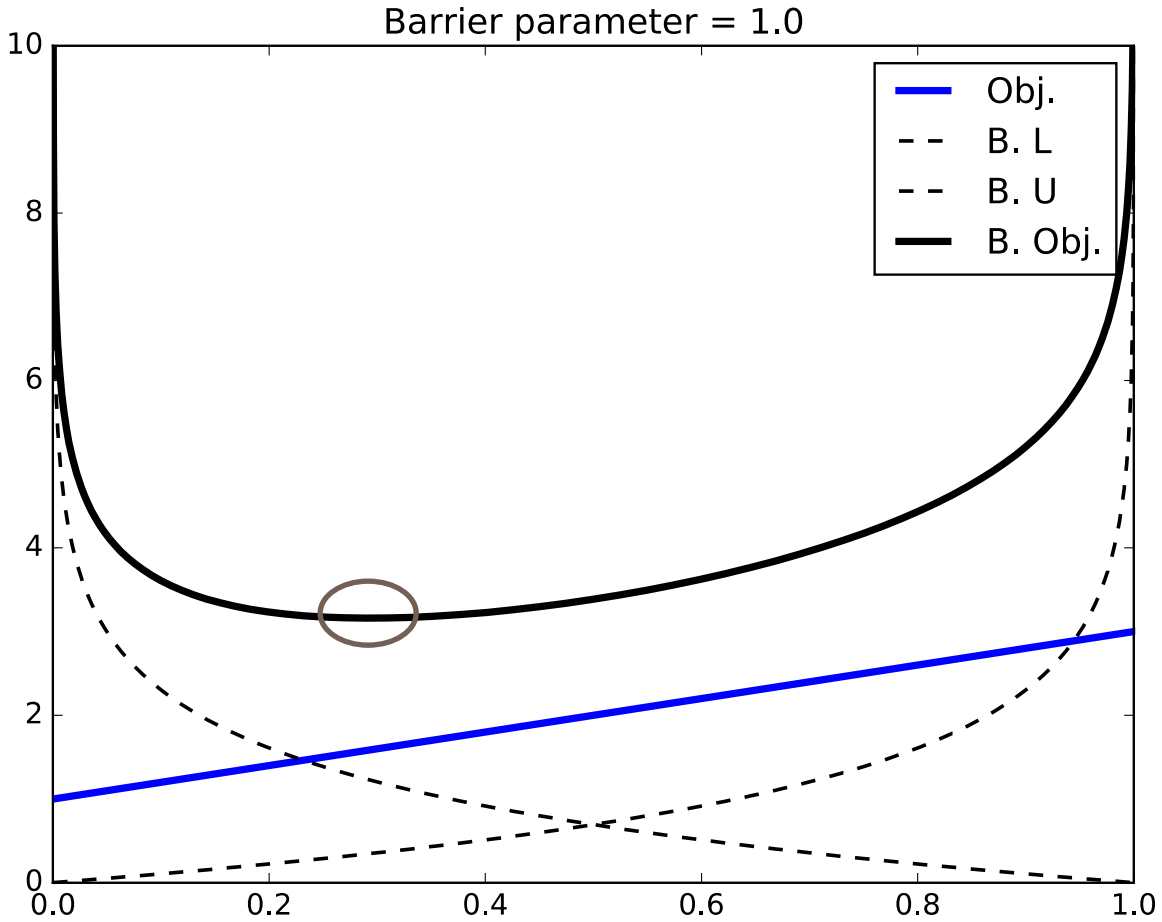
Effect of Barrier Term



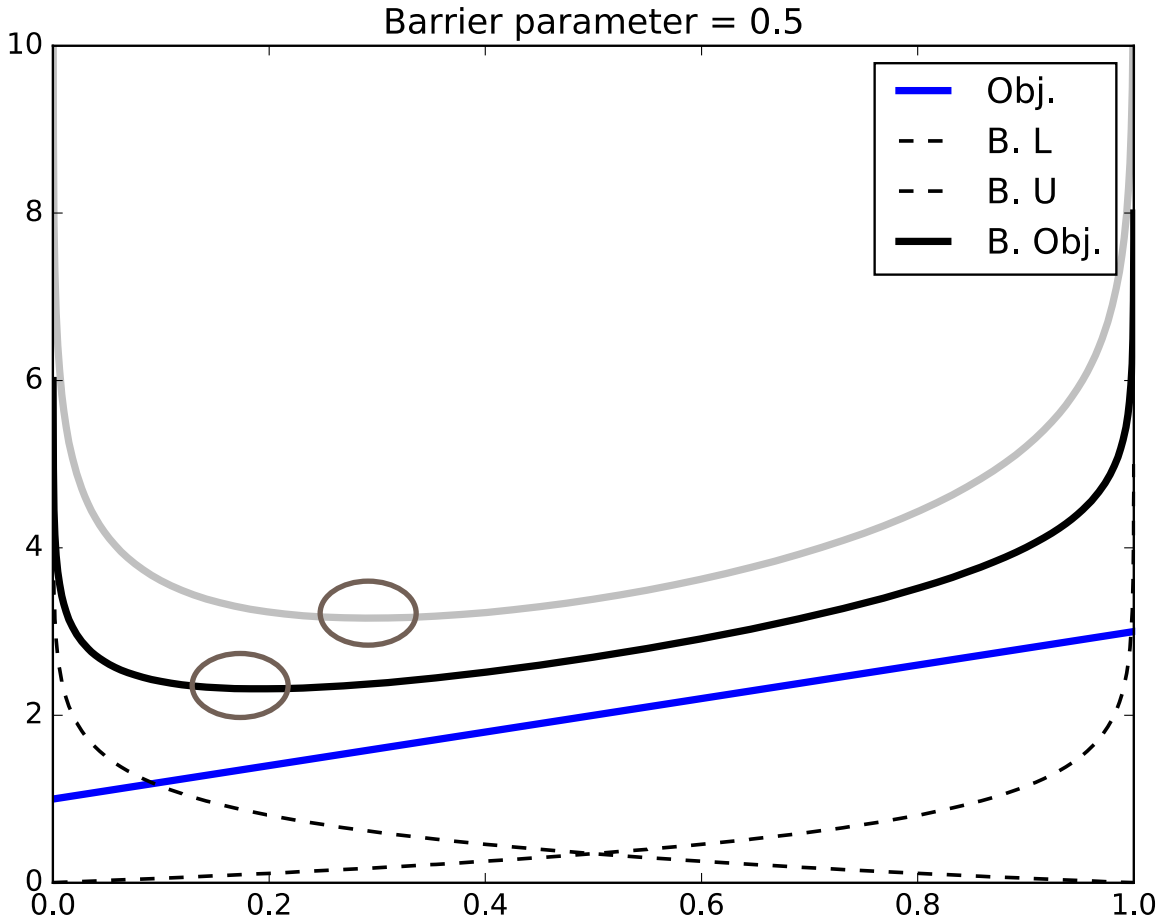
Effect of Barrier Term



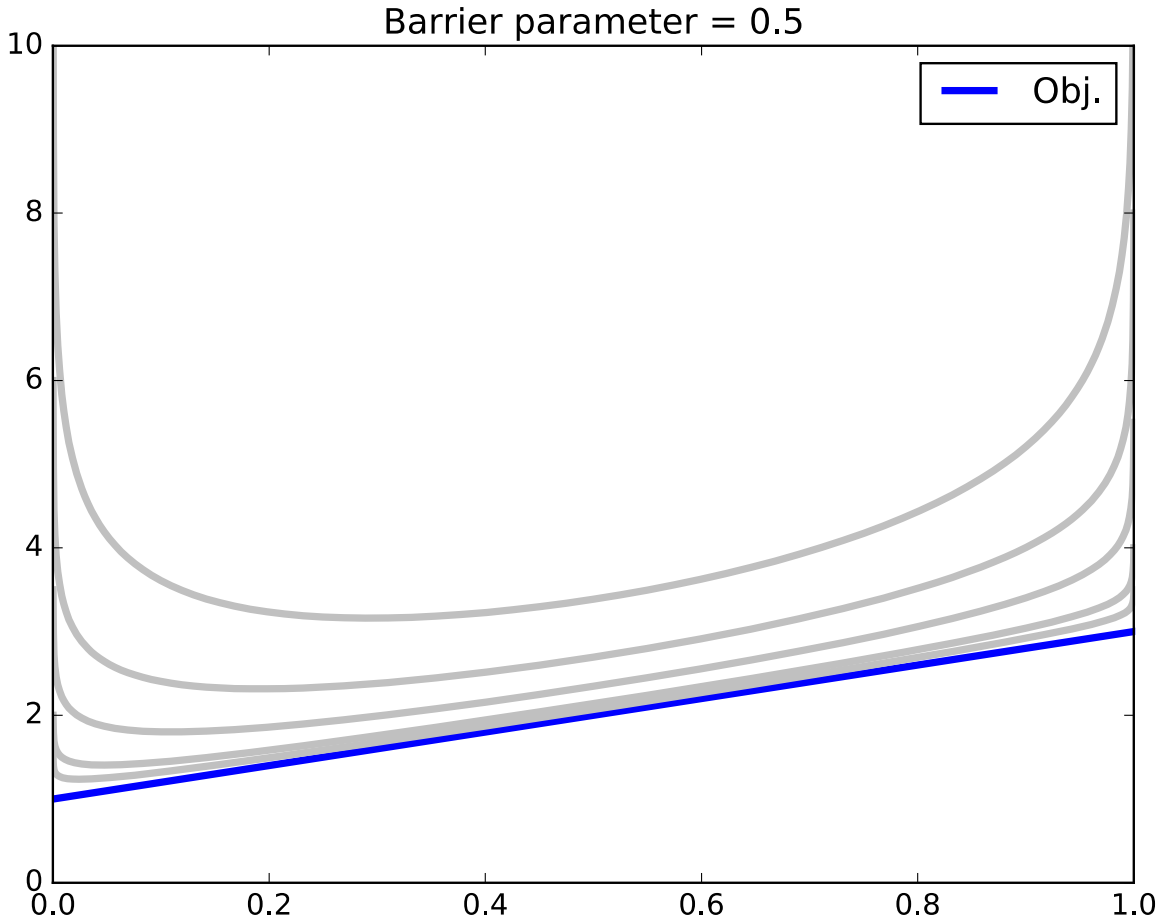
Effect of Barrier Term



Effect of Barrier Term



Effect of Barrier Term



Interior Point Methods

Original NLP

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & x \geq 0 \end{aligned}$$

Barrier NLP

$$\begin{aligned} \min_x \quad \varphi_\mu(x) = \quad & f(x) - \mu \cdot \sum_i ln(x_i) \\ \text{s.t.} \quad & c(x) = 0 \end{aligned}$$

$$\text{as} \quad x \to 0, \quad ln(x) \to \infty$$

$$\text{as} \quad \mu \to 0, \quad x^\star(\mu) \to x^\star$$

Fiacco & McCormick (1968)

- Initialize
  $x_0 > 0, \quad \mu_0 > 0, \quad \text{Set } l \leftarrow 0$
- Solve Barrier NLP for μ_l
- Decrease the barrier parameter
  $\mu_{l+1} < \mu_l$
- Increase $l \leftarrow l + 1$

Solve Barrier NLP?
Barrier parameter update?
Globalization?

- KNITRO (Byrd, Nocedal, Hribar, Waltz)
- LOQO (Benson, Vanderbei, Shanno
- Ipopt (Waechter, Biegler)

Interior Point Methods

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t} \quad & c(x)=0 \\ & x \geq 0 \end{aligned}$$

$$\begin{aligned} \min_x \quad & f(x) - \mu \sum_i \ln(x_i) \\ \text{s.t} \quad & c(x)=0 \end{aligned}$$

$$\begin{aligned} \nabla f(x) + \nabla c(x)^T \lambda - z &= 0 \\ c(x) &= 0 \\ Xz &= \mu e \\ (x > 0, z > 0) & \end{aligned}$$

$z=\mu X^{-1} e$

$$\begin{aligned} \nabla f(x) + \nabla c(x)^T \lambda - \mu X^{-1} e &= 0 \\ c(x) &= 0 \\ (x > 0) & \end{aligned}$$

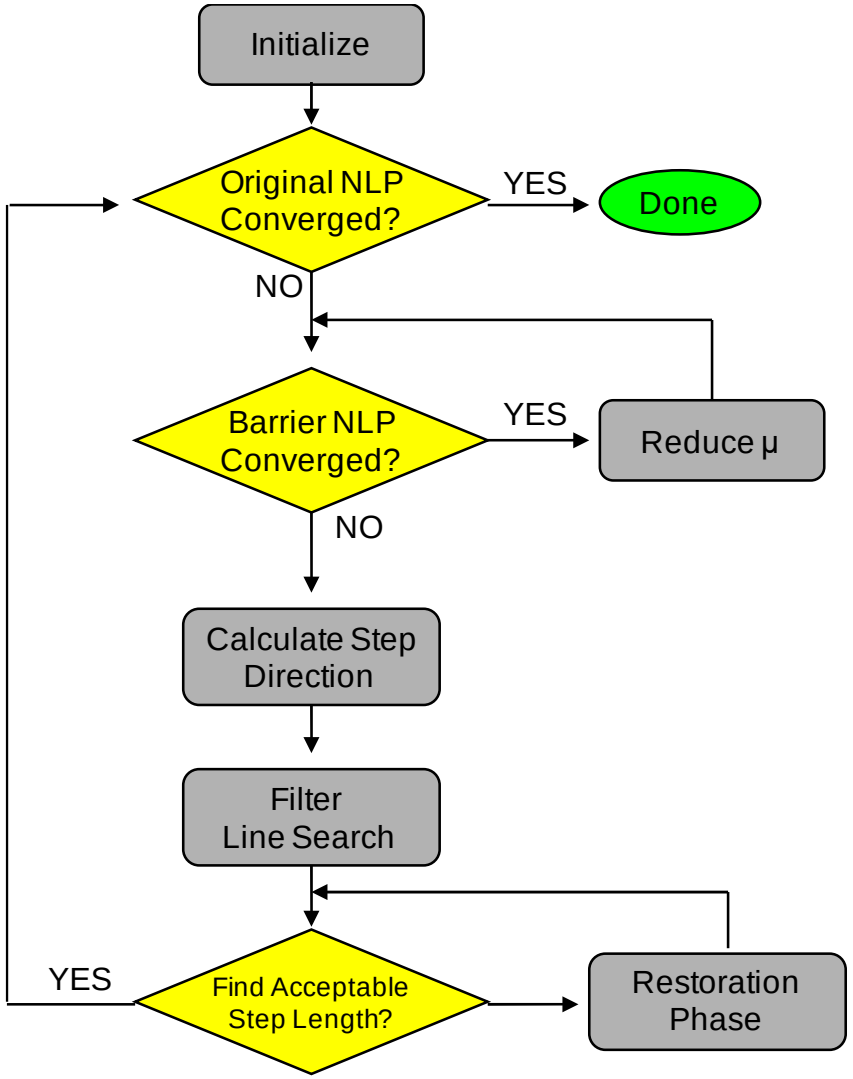
$$\begin{bmatrix} W_k + \Sigma_k + \delta_w I & \nabla c(x_k) \\ \nabla c(x_k)^T & -\delta_c I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = - \begin{bmatrix} \nabla \psi_\mu(x_k) + \nabla c(x_k)^T \lambda_k \\ c(x_k) \end{bmatrix}$$

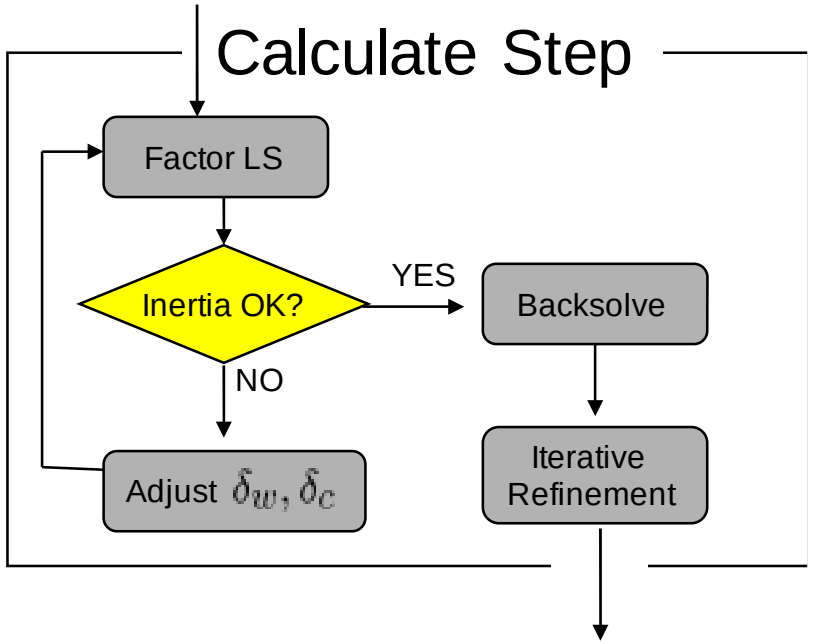
$$(W_k = \nabla^2_{xx} \mathcal{L},\ \Sigma_k = Z_k X_k^{-1})$$

Ipopt: Other Considerations

- Regularization:
  - If certain convexity criteria are not satisfied at a current point, Ipopt may need to regularize. (This can be seen in the output.)
  - We do NOT want to see regularization at the final iteration (solution).
  - Can be an indicator of poor conditioning.
- Globalization:
  - Ipopt uses a filter-based line-search approach
  - Accepts the step if sufficient reduction is seen in objective or constraint violation
- Restoration Phase:
  - Minimize constraint violation
  - Regularized with distance from current point
  - Similar structure to original problem (reuse symbolic factorization)

Ipopt Algorithm Flowsheet





Ipopt Output

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

This is Ipopt version 3.11.7, running with linear solver ma27.

Number of nonzeros in equality constraint Jacobian...:        2
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian.............:        1

Total number of variables............................:        2
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        1
                     variables with only upper bounds:        0
Total number of equality constraints.................:        1
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  5.0000000e-01 2.50e-01 5.00e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
```

Ipopt Output

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  5.0000000e-01 2.50e-01 5.00e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  2.4298076e-01 2.33e-01 7.67e-01  -1.0 5.20e-01    -  7.73e-01 9.52e-01h  1
   2  2.6898113e-02 7.23e-05 4.09e-04  -1.7 2.16e-01    -  1.00e+00 1.00e+00h  1
   3  1.8655807e-04 1.83e-04 7.86e-05  -3.8 2.68e-02    -  1.00e+00 9.97e-01f  1
   4  1.8250072e-06 1.23e-12 2.22e-16  -5.7 1.85e-04    -  1.00e+00 1.00e+00h  1
   5 -1.7494097e-08 8.48e-13 0.00e+00  -8.6 1.84e-06    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 5

                                   (scaled)                 (unscaled)
Objective...............:  -1.7494096510394117e-08   -1.7494096510394117e-08
Dual infeasibility......:   0.0000000000000000e+00    0.0000000000000000e+00
Constraint violation....:   8.4843243541854463e-13    8.4843243541854463e-13
Complementarity.........:   2.5050549017950606e-09    2.5050549017950606e-09
Overall NLP error.......:   2.5050549017950606e-09    2.5050549017950606e-09
```

- iter: iterations (codes)
- objective: objective
- Inf_pr: primal infeasibility (constraints satisfied? current constraint violation)
- Inf_du: dual infeasibility (am I optimal?)
- lg(mu): log of the barrier parameter, mu
- ||d||: length of the current stepsize
- lg(rg): log of the regularization parameter
- alpha_du: stepsize for dual variables
- alpha_pr: stepsize for primal variables
- ls: number of line-search steps

Exit Conditions

- Successful Exit
- Successful Exit with regularization at solution
- Infeasible
- Unbounded

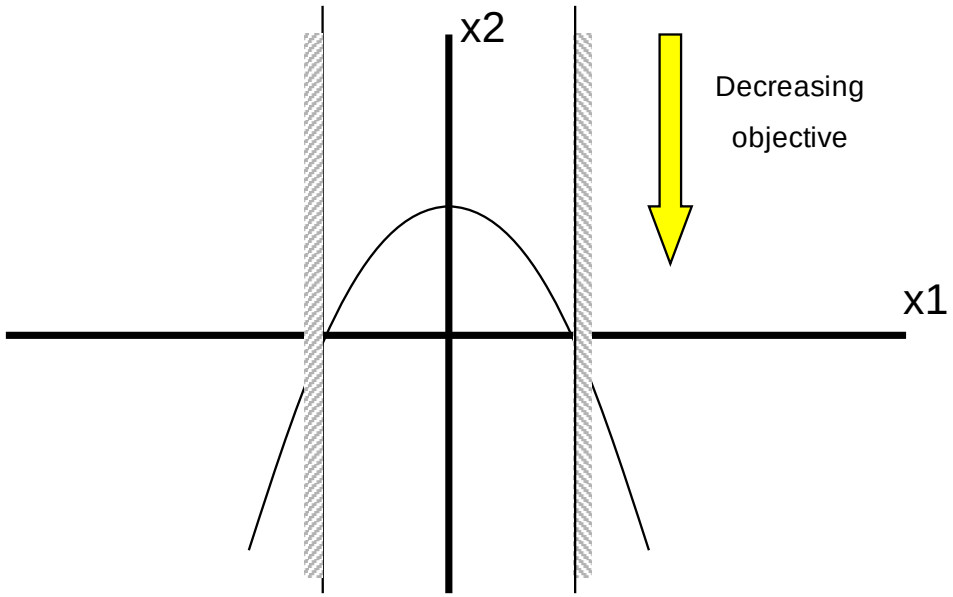
Exit Conditions: Successful

$$\min \quad x_2$$

$$-x_2 = x_1^2 - 1$$

$$-1 \leq x_1 \leq 1$$

Initialize at (x1=0.5, x2=0.5)



Exit Conditions: Successful

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  5.0000000e-01 2.50e-01 5.00e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  2.4298076e-01 2.33e-01 7.67e-01  -1.0 5.20e-01    -  7.73e-01 9.52e-01h  1
   2  2.6898113e-02 7.23e-05 4.09e-04  -1.7 2.16e-01    -  1.00e+00 1.00e+00h  1
   3  1.8655807e-04 1.83e-04 7.86e-05  -3.8 2.68e-02    -  1.00e+00 9.97e-01f  1
   4  1.8250072e-06 1.23e-12 2.22e-16  -5.7 1.85e-04    -  1.00e+00 1.00e+00h  1
   5 -1.7494097e-08 8.48e-13 0.00e+00  -8.6 1.84e-06    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 5

                                   (scaled)                 (unscaled)
Objective...............:  -1.7494096510367012e-08   -1.7494096510367012e-08
Dual infeasibility......:   0.0000000000000000e+00    0.0000000000000000e+00
Constraint violation....:   8.4843243541854463e-13    8.4843243541854463e-13
Complementarity.........:   2.5050549017950606e-09    2.5050549017950606e-09
Overall NLP error.......:   2.5050549017950606e-09    2.5050549017950606e-09

. . .

EXIT: Optimal Solution Found.

Ipopt 3.11.1: Optimal Solution Found

*** soln
x1 = 1.0
x2 = -1.7494096510367012e-08
```

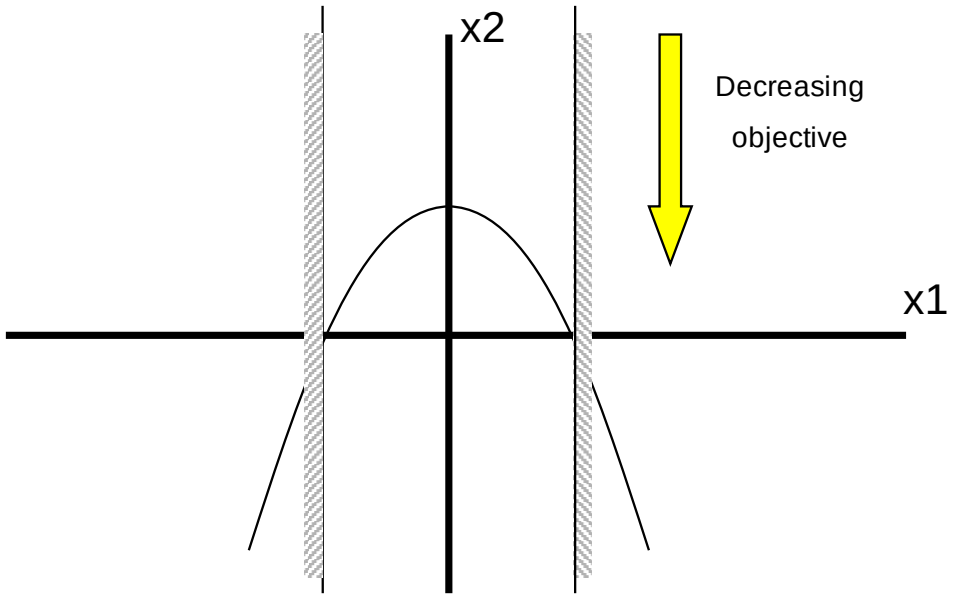
Exit Conditions: Successful w/ Regularization

$$\min \quad x_2$$

$$-x_2 = x_1^2 - 1$$

$$-1 \leq x_1 \leq 1$$

Initialize at (x1=0.0, x2=2.0)



Exit Conditions: Successful w/ Regularization

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  2.0000000e+00 1.00e+00 0.00e+00  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.0000000e+00 0.00e+00 1.00e-04  -1.7 1.00e+00  -4.0 1.00e+00 1.00e+00h  1
   2  1.0000000e+00 0.00e+00 0.00e+00  -3.8 0.00e+00   0.9 1.00e+00 1.00e+00   0
   3  1.0000000e+00 0.00e+00 0.00e+00  -5.7 0.00e+00   0.5 1.00e+00 1.00e+00T  0
   4  1.0000000e+00 0.00e+00 0.00e+00  -8.6 0.00e+00   0.9 1.00e+00 1.00e+00T  0

Number of Iterations....: 4

                                   (scaled)                 (unscaled)
Objective...............:   1.0000000000000000e+00    1.0000000000000000e+00
Dual infeasibility......:   0.0000000000000000e+00    0.0000000000000000e+00
Constraint violation....:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity.........:   2.5059035596800808e-09    2.5059035596800808e-09
Overall NLP error.......:   2.5059035596800808e-09    2.5059035596800808e-09

. . .

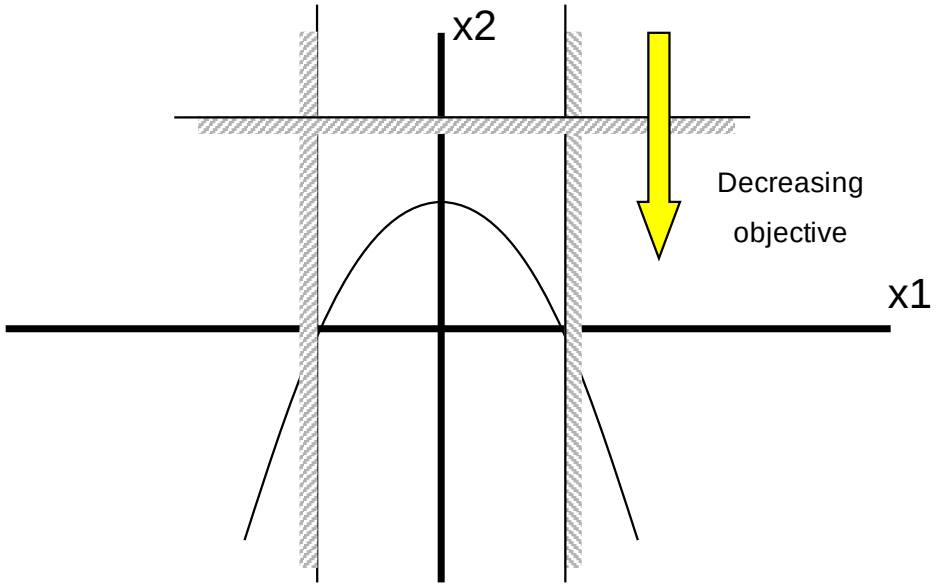
EXIT: Optimal Solution Found.

Ipopt 3.11.1: Optimal Solution Found

*** soln
x1 = 0.0
x2 = 1.0
```

Exit Conditions: Infeasible

$$\begin{aligned} \min \quad & -x_2 \\ \text{s.t.} \quad & -x_2 = x_1^2 - 1 \\ & -1 \leq x_1 \leq 1 \\ & x_2 \geq 2 \end{aligned}$$



Exit Conditions: Infeasible

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
. . .
   5  2.0000016e+00 1.00e+00 4.74e+03  -1.0 3.47e+01    -  2.79e-02 4.16e-04h  7
   6r 2.0000016e+00 1.00e+00 1.00e+03   0.0 0.00e+00    -  0.00e+00 4.44e-07R  3
   7r 2.0010000e+00 1.01e+00 1.74e+02   0.0 8.73e-02    -  1.00e+00 1.00e+00f  1
   8r 2.0010010e+00 1.00e+00 1.32e-03   0.0 8.73e-02    -  1.00e+00 1.00e+00f  1
   9r 2.0000080e+00 1.00e+00 5.18e-03  -2.1 6.12e-03    -  9.94e-01 9.99e-01h  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10r 2.0000000e+00 1.00e+00 3.73e-06  -4.7 7.99e-06    -  1.00e+00 1.00e+00f  1
  11r 2.0000000e+00 1.00e+00 1.79e-07  -7.1 2.85e-08    -  1.00e+00 1.00e+00f  1

Number of Iterations....: 11

                                   (scaled)                 (unscaled)
Objective...............:   1.9999999800009090e+00    1.9999999800009090e+00
Dual infeasibility......:   1.0000000002321485e+00    1.0000000002321485e+00
Constraint violation....:   9.9999998000090895e-01    9.9999998000090895e-01
Complementarity.........:   9.0909091652062654e-10    9.0909091652062654e-10
Overall NLP error.......:   9.9999998000090895e-01    1.0000000002321485e+00

. . .

EXIT: Converged to a point of local infeasibility. Problem may be infeasible.

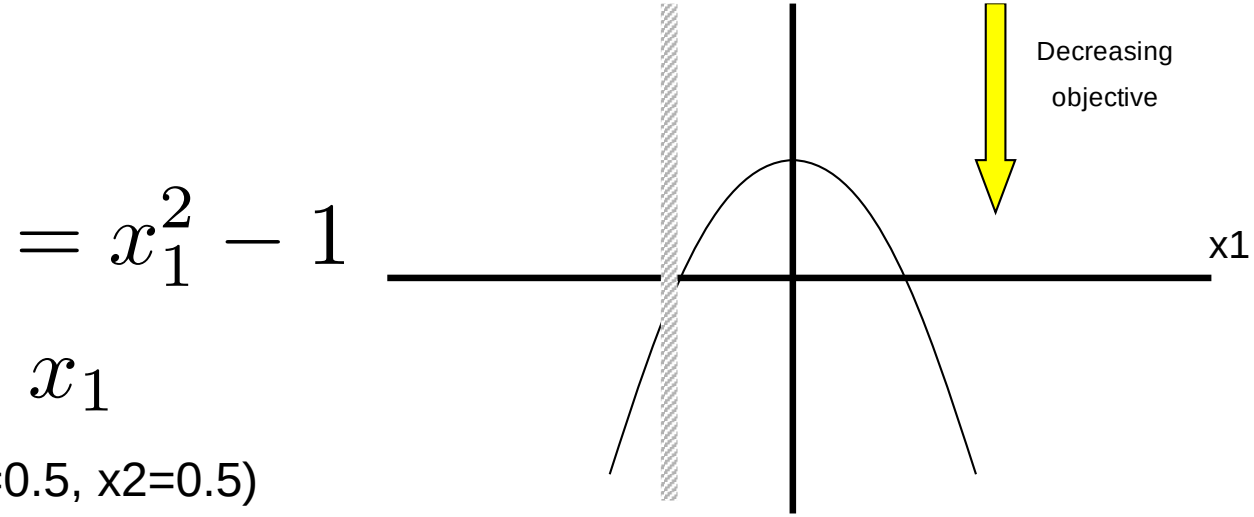
Ipopt 3.11.1: Converged to a locally infeasible point. Problem may be infeasible.
WARNING - Loading a SolverResults object with a warning status into model=unknown; message from solver=Ipopt
3.11.1\x3a Converged to a locally infeasible point. Problem may be infeasible.

*** soln
x1 = -6.353194883662875e-12
x2 = 2.0
```

Exit Conditions: Unbounded

$$\begin{aligned} \min \quad & -x_2 \\ \text{s.t.} \quad & -x_2 = x_1^2 - 1 \\ & -1 \leq x_1 \end{aligned}$$

Initialize at (x1=0.5, x2=0.5)



Exit Conditions: Unbounded

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
. . .
  45 -2.2420218e+11 1.00e+04 1.00e+00  -1.7 1.25e+19 -19.1 3.55e-08 7.11e-15f 48
  46 -2.2420225e+11 1.00e+04 1.00e+00  -1.7 3.75e+19 -19.6 1.20e-08 1.78e-15f 50
  47 -2.2420229e+11 1.00e+04 1.00e+00  -1.7 1.25e+19 -19.1 1.00e+00 3.55e-15f 49
  48 -3.7503956e+19 1.57e+27 8.36e+09  -1.7 3.75e+19 -19.6 1.18e-08 1.00e+00w  1
  49 -1.3750923e+20 3.92e+26 2.09e+09  -1.7 1.00e+20 -20.0 1.00e+00 1.00e+00w  1

Number of Iterations....: 49

                                   (scaled)                 (unscaled)
Objective...............:  -1.3750923074037683e+20   -1.3750923074037683e+20
Dual infeasibility......:   2.0888873315629249e+09    2.0888873315629249e+09
Constraint violation....:   3.9209747283936173e+26    3.9209747283936173e+26
Complementarity.........:   3.1115099971882619e+03    3.1115099971882619e+03
Overall NLP error.......:   3.9209747283936173e+26    3.9209747283936173e+26

EXIT: Iterates diverging; problem might be unbounded.

Ipopt 3.11.1: Iterates diverging; problem might be unbounded.
WARNING - Loading a SolverResults object with a warning status into model=unknown; message from
solver=Ipopt 3.11.1\x3a Iterates diverging; problem might be unbounded.

*** soln
x1 = 0.5
x2 = 0.5
```

Ipopt Options

- Solver options can be set through scripts (and the pyomo command line)
- print_options_documentation yes
  - Outputs the complete set of Ipopt options (with documentation and their defaults)
- mu_init
  - Sets the initial value of the barrier parameter
  - Can be helpful to make this smaller when initial guesses are known to be good
- bounds_push
  - By default, Ipopt pushes the bounds a little further out.
  - This can be set to remove this behavior
  - E.g., sqrt(x), x>= 0
- linear_solver
  - Set the linear solver that will be used for the KKT system
  - Significantly better performance with HSL (MA27) over default MUMPS
- print_user_options
  - Print options set and whether or not they were used
  - Helpful to detect mismatched options

Ipopt Options

```
# rosenbrock_options.py: A Pyomo model for the Rosenbrock problem
import pyomo.environ as pyo

model = pyo.ConcreteModel()
model.x = pyo.Var()
model.y = pyo.Var()

def rosenbrock(m):
    return (1.0-m.x)**2 + 100.0*(m.y - m.x**2)**2
model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)

solver = pyo.SolverFactory('ipopt')
solver.options['mu_init'] = 1e-4
solver.options['print_user_options'] = 'yes'
solver.options['ma27_pivtol'] = 1e-4
solver.solve(model, tee=True)

print()
print('*** Solution *** :')
print('x:', pyo.value(model.x))
print('y:', pyo.value(model.y))
```

Ipopt Options

```
ma27_pivtol=0.0001
print_user_options=yes
mu_init=0.0001
ma27_pivtol=0.0001
print_user_options=yes
mu_init=0.0001


List of user-set options:

                                    Name   Value                used
                           ma27_pivtol = 0.0001                 no
                               mu_init = 0.0001                 yes
                     print_user_options = yes                   yes

******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

NOTE: You are using Ipopt by default with the MUMPS linear solver.
      Other linear solvers might be more efficient (see Ipopt documentation).


This is Ipopt version 3.11.1, running with linear solver mumps.
```

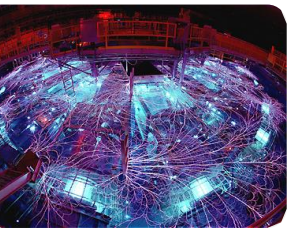
Other Considerations

- Linear Solvers
  - Performance of Ipopt is HIGHLY dependent on the linear solver selected
  - The version of Ipopt installed in this workshop uses a freely distributable linear solver, MUMPS.
  - Performance of Ipopt with the Harwell Subroutine Library suite can be significantly better
  - MA27 (HSL) is **free** for personal and commercial use, but not **distributable**
  - IDAES-PSE distributes compiled Ipopt binaries with HSL for most (but not all) platforms
  - Alternatively you can manually download and install the Coin-HSL Archive (MA27) from HSL
    - Web search: "HSL for Ipopt“

- Variable Initialization
  - Proper initialization of nonlinear problems can be critical for effective solution.
  - Strategies include:
    - Using understood physics or past successful solutions
    - Solving simpler problem(s) first, progressing to more difficult

Other Considerations

- Undefined Evaluations
  - Many mathematical functions have a valid domain, and evaluation outside that domain causes errors
  - Add appropriate bounds to variables to keep them inside valid domain
  - Note that solvers use first and second derivatives. While sqrt(x) is valid at x=0, 1/sqrt(x) is not
- Problem Scaling
  - Scale model to avoid variables, constraints, derivatives with different scales.
- Formulation Matters



Sandia National Laboratories

Exceptional service in the national interest

Section 6

Introduction to Blocks: Structured Modeling in Pyomo

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

What's the problem with Math Programming?

Sandia National Laboratories

$$\text{Minimize}: \quad \sum_t \sum_g \left(c_g P_{g0t} + c_g^{SU} v_{gt} + c_g^{SD} w_{gt}\right) \tag{1}$$

$$\text{S.t.} \quad \theta^{\min} \le \theta_{nct} \le \theta^{\max}, \quad \forall\, n, c, t \tag{2}$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{g0t} = d_{nt},$$

$$\forall\, n, \quad c = 0, \text{ transmission contingency states } c, t \tag{3a}$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{gct} = d_{nt},$$

$$\forall\, n, \text{ generator contingency states } c, t \tag{3b}$$

$$P_{kc}^{\min} N1_{kc} z_{kt} \le P_{kct} \le P_{kc}^{\max} N1_{kc} z_{kt}, \quad \forall\, k, c, t \tag{4}$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} + (2 - z_{kt} - N1_{kc}) M_k \ge 0, \quad \forall\, k, c, t \tag{5a}$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} - (2 - z_{kt} - N1_{kc}) M_k \le 0, \quad \forall\, k, c, t \tag{5b}$$

$$P_g^{\min} N1_{gc} u_{gt} \le P_{gct} \le P_g^{\max} N1_{gc} u_{gt}, \quad \forall\, g, c, t \tag{6}$$

$$v_{g,t} - w_{g,t} = u_{g,t} - u_{g,t-1}, \quad \forall\, g, t \tag{7}$$

$$\sum_{q=t-UT_g+1}^{t} v_{g,q} \le u_{g,t}, \quad \forall\, g, t \in \{UT_g, \ldots, T\} \tag{8}$$

$$\sum_{q=t-DT_g+1}^{t} w_{g,q} \le 1 - u_{g,t}, \forall\, g, t \in \{DT_g, \ldots, T\} \tag{9}$$

$$P_{g0t} - P_{g0,t-1} \le R_g^{+} u_{g,t-1} + R_g^{SU} v_{g,t}, \quad \forall\, g, t \tag{10}$$

$$P_{g0,t-1} - P_{g0,t} \le R_g^{-} u_{g,t} + R_g^{SD} w_{g,t}, \quad \forall\, g, t \tag{11}$$

$$P_{gct} - P_{g0,t} \le R_g^{+}, \quad \forall\, g, c, t \tag{12}$$

$$P_{g0,t} N1_{gc} - P_{gct} \le R_g^{-}, \quad \forall\, g, c, t \tag{13}$$

$$0 \le v_{g,t} \le 1, \quad \forall\, g, t \tag{14}$$

$$0 \le w_{g,t} \le 1, \quad \forall\, g, t \tag{15}$$

$$u_{g,t} \in \{0, 1\}, \quad \forall\, g, t \tag{16}$$

CCR Center for Computing Research

Hedman, et al., "Co-Optimization of Generation Unit Commitment and Transmission Switching With N-1 Reliability," *IEEE Trans Power Systems*, 25(2), pp.1052-1063, 2010

What's the problem with Math Programming?

$$\text{Minimize}: \quad \sum_t \sum_g \left(c_g P_{g0t} + c_g^{SU} v_{gt} + c_g^{SD} w_{gt}\right) \tag{1}$$

$$\text{S.t.} \quad \theta^{\min} \le \theta_{nct} \le \theta^{\max}, \quad \forall\, n, c, t \tag{2}$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{g0t} = d_{nt},$$

$$\forall\, n, \quad c = 0, \text{ transmission contingency states } c, t \tag{3a}$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{gct} = d_{nt},$$

$$\forall\, n, \text{ generator contingency states } c, t \tag{3b}$$

$$P_{kc}^{\min} N1_{kc} z_{kt} \le P_{kct} \le P_{kc}^{\max} N1_{kc} z_{kt}, \quad \forall\, k, c, t \tag{4}$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} + (2 - z_{kt} - N1_{kc}) M_k \ge 0, \quad \forall\, k, c, t \tag{5a}$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} - (2 - z_{kt} - N1_{kc}) M_k \le 0, \quad \forall\, k, c, t \tag{5b}$$

$$P_g^{\min} N1_{gc} u_{gt} \le P_{gct} \le P_g^{\max} N1_{gc} u_{gt}, \quad \forall\, g, c, t \tag{6}$$

$$v_{g,t} - w_{g,t} = u_{g,t} - u_{g,t-1}, \quad \forall\, g, t \tag{7}$$

$$\sum_{q=t-UT_g+1}^{t} v_{g,q} \le u_{g,t}, \quad \forall\, g, t \in \{UT_g, \ldots, T\} \tag{8}$$

$$\sum_{q=t-DT_g+1}^{t} w_{g,q} \le 1 - u_{g,t}, \forall\, g, t \in \{DT_g, \ldots, T\} \tag{9}$$

$$P_{g0t} - P_{g0,t-1} \le R_g^{+} u_{g,t-1} + R_g^{SU} v_{g,t}, \quad \forall\, g, t \tag{10}$$

$$P_{g0,t-1} - P_{g0,t} \le R_g^{-} u_{g,t} + R_g^{SD} w_{g,t}, \quad \forall\, g, t \tag{11}$$

$$P_{gct} - P_{g0,t} \le R_g^{+}, \quad \forall\, g, c, t \tag{12}$$

$$P_{g0,t} N1_{gc} - P_{gct} \le R_g^{-}, \quad \forall\, g, c, t \tag{13}$$

$$0 \le v_{g,t} \le 1, \quad \forall\, g, t \tag{14}$$

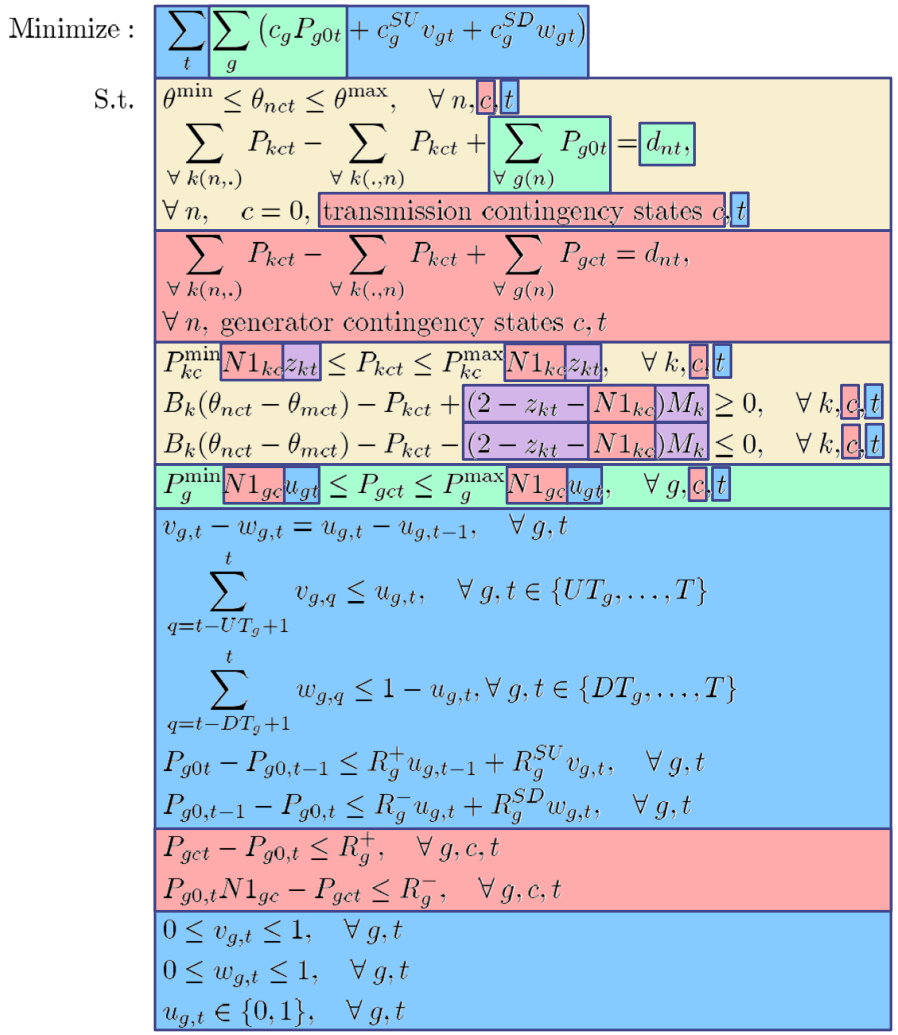
$$0 \le w_{g,t} \le 1, \quad \forall\, g, t \tag{15}$$

$$u_{g,t} \in \{0, 1\}, \quad \forall\, g, t \tag{16}$$

- This is a "solution" for Unit Commitment + Transmission Switching + explicit N-1 reliability
- This is a very concise notation
  - Excellent for papers
  - But obscures the original problem

Hedman, et al., "Co-Optimization of Generation Unit Commitment and Transmission Switching With N-1 Reliability," *IEEE Trans Power Systems*, 25(2), pp.1052-1063, 2010

The challenge: MP is dense and subtle



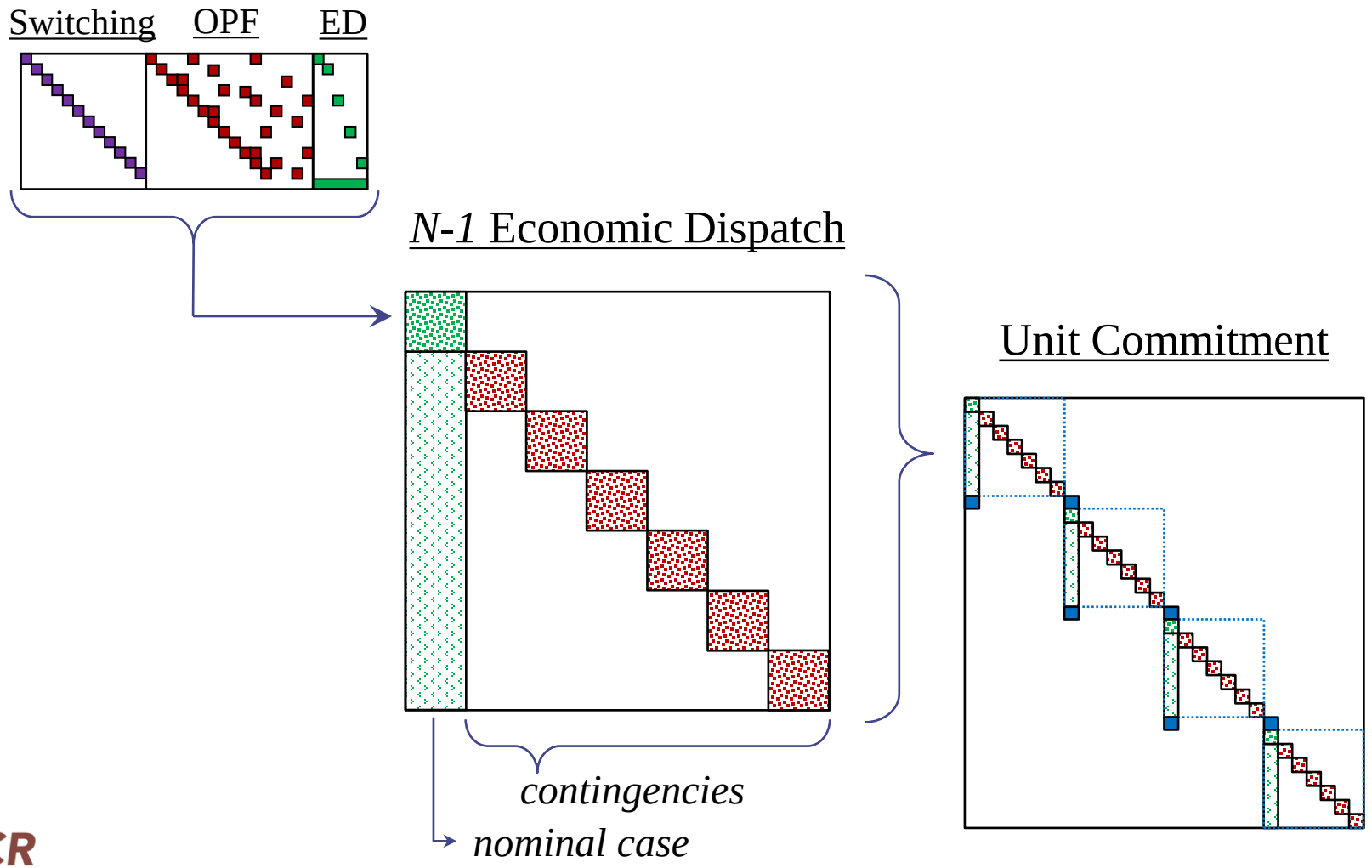
To a first approximation:

- DCOPF
- Economic dispatch
- Unit commitment
- Transmission switching
- N-1 contingency

Hedman, et al., "Co-Optimization of Generation Unit Commitment and Transmission Switching With N-1 Reliability," *IEEE Trans Power Systems*, 25(2), pp.1052-1063, 2010

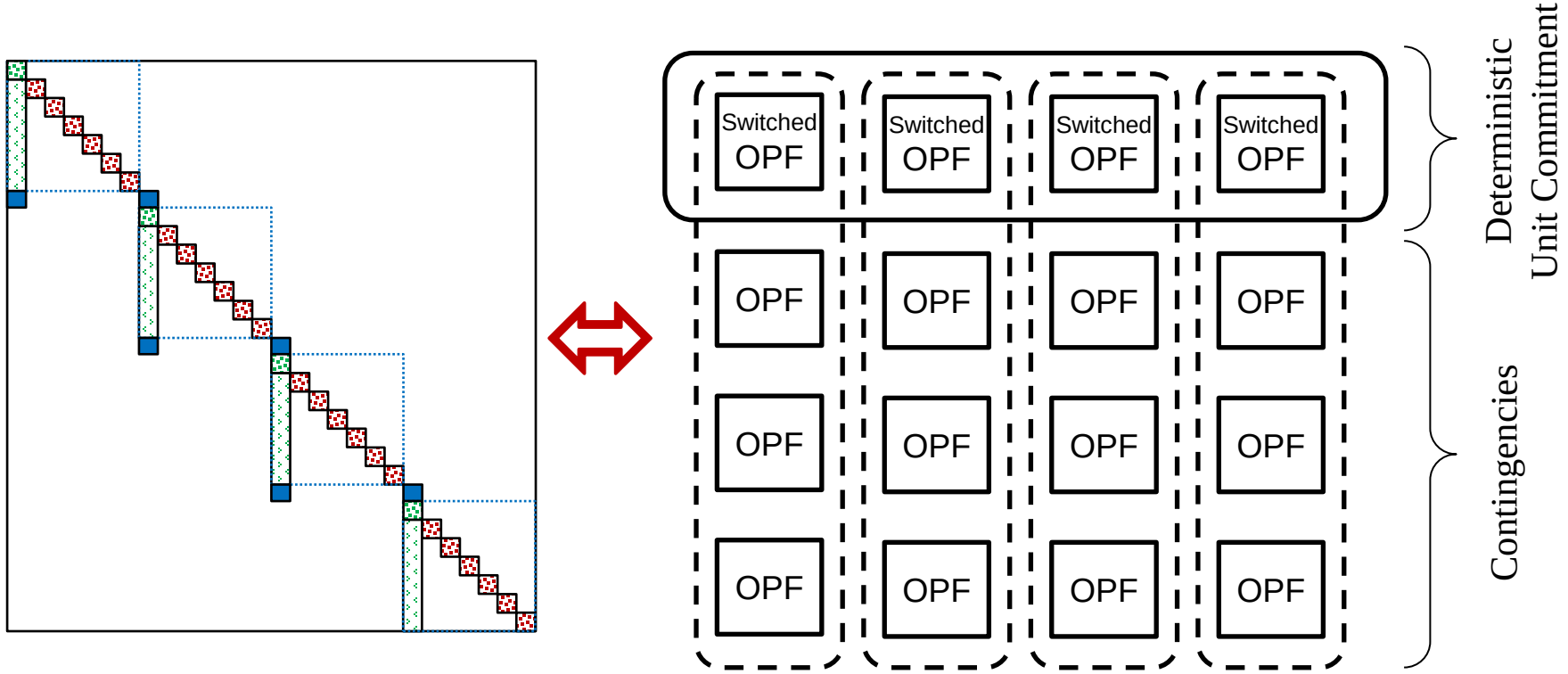
(Nonobvious) Inherent structure

(Sparsity pattern in model constraint matrix)

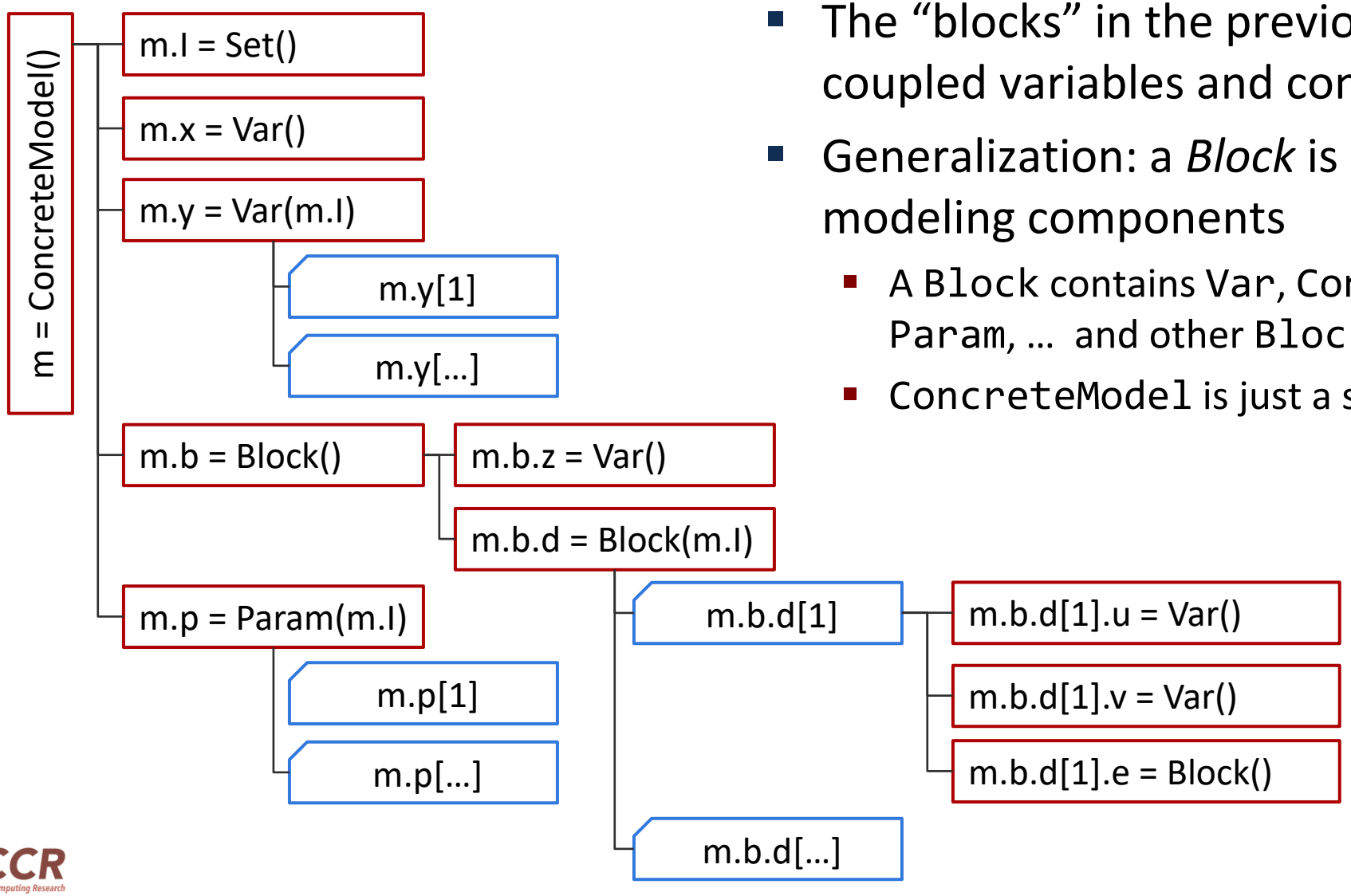


UC + N-1 + Switching block structure

- "2-D" grid of linked optimal power flow models



How can we capture *structure* in optimization models?



- The "blocks" in the previous model are (tightly) coupled variables and constraints
- Generalization: a *Block* is a collection of modeling components
  - A `Block` contains `Var`, `Constraint`, `Objective`, `Param`, … and other `Block` components
  - `ConcreteModel` is just a special form of a `Block`

Why blocks

- Independence
  - You can solve them! (Independently of the rest of the model!)
    - Debugging, initialization
  - You can “turn on / off” portions of the model
- Namespacing
  - You can “reserve” modeling spaces where you have complete control
- Composition
  - You can assemble multiple blocks together into a single model
  - You can add new indices (time, space, scenarios) to a submodel without changing the model
  - You can build “general” model libraries
- Algorithms
  - You can use blocks to inform the solution process
    - Decomposition (e.g. progressive hedging, parapint)
    - Branching Logic (GDPopt)
    - Iterative algorithms (PyROS, column generation)

Blocks Provide *Namespacing*

- Each block defines its own namespace

```
print(model.x.local_name)         # x
print(model.x.name)               # x
print(model.b.x.local_name)       # x
print(model.b.x.name)             # b.x
print(model.b.b[1].x.local_name) # x
print(model.b.b[1].x.name)        # b.b[1].x
```

- Blocks can be created (and "solved") and later added to a parent model

```
new_b = pyo.Block()
new_b.x = pyo.Var()
new_b.P = pyo.Param(initialize=5)
new_b.I = pyo.RangeSet(10)
model = pyo.ConcreteModel()
model.b = new_b
model.x = pyo.Var(model.b.I)
```

Defining Blocks: Rules

- Blocks may be indexed (and defined by rules)

```
model = pyo.ConcreteModel()
model.P = pyo.Param(initialize=3)
model.T = pyo.RangeSet(model.P)
model.z = pyo.Var()

def xyb_rule(b, t):
    b.x = pyo.Var()
    b.I = pyo.RangeSet(t)
    b.y = pyo.Var(b.I, initialize=1.0)
    b.c = pyo.Constraint(expr= b.x == 1.0 - sum(b.y[i] for i in b.I))

    def d_rule(_b, i):
        return _b.y[i] >= _b.model().z
    b.d = pyo.Constraint(b.I, rule=d_rule)

model.xyb = pyo.Block(model.T, rule=xyb_rule)
```

Defining Blocks: Rules

- Blocks may be indexed (and defined by rules)

```
model = pyo.ConcreteModel()
model.P = pyo.Param(initialize=3)
model.T = pyo.RangeSet(model.P)
model.z = pyo.Var()

def xyb_rule(b, t):
    b.x = pyo.Var()
    b.I = pyo.RangeSet(t)
    b.y = pyo.Var(b.I, initialize=1.0)
    b.c = pyo.Constraint(expr= b.x == 1.0 - sum(b.y[i] for i in b.I))

    def d_rule(_b, i):
        return _b.y[i] >= _b.model().z
    b.d = pyo.Constraint(b.I, rule=d_rule)

model.xyb = pyo.Block(model.T, rule=xyb_rule)
```

Block rules get the *block to be populated* as the first argument

Pyomo rules actually pass the *owning block* as the first argument (prior to this the owning block was always the model)

Rules can be defined within rules (for convenience, but may prevent pickling)

Block Rules Can Return (New) Blocks

- Blocks may be indexed (and defined by rules)

```
model = pyo.ConcreteModel()
model.P = pyo.Param(initialize=3)
model.T = pyo.RangeSet(model.P)

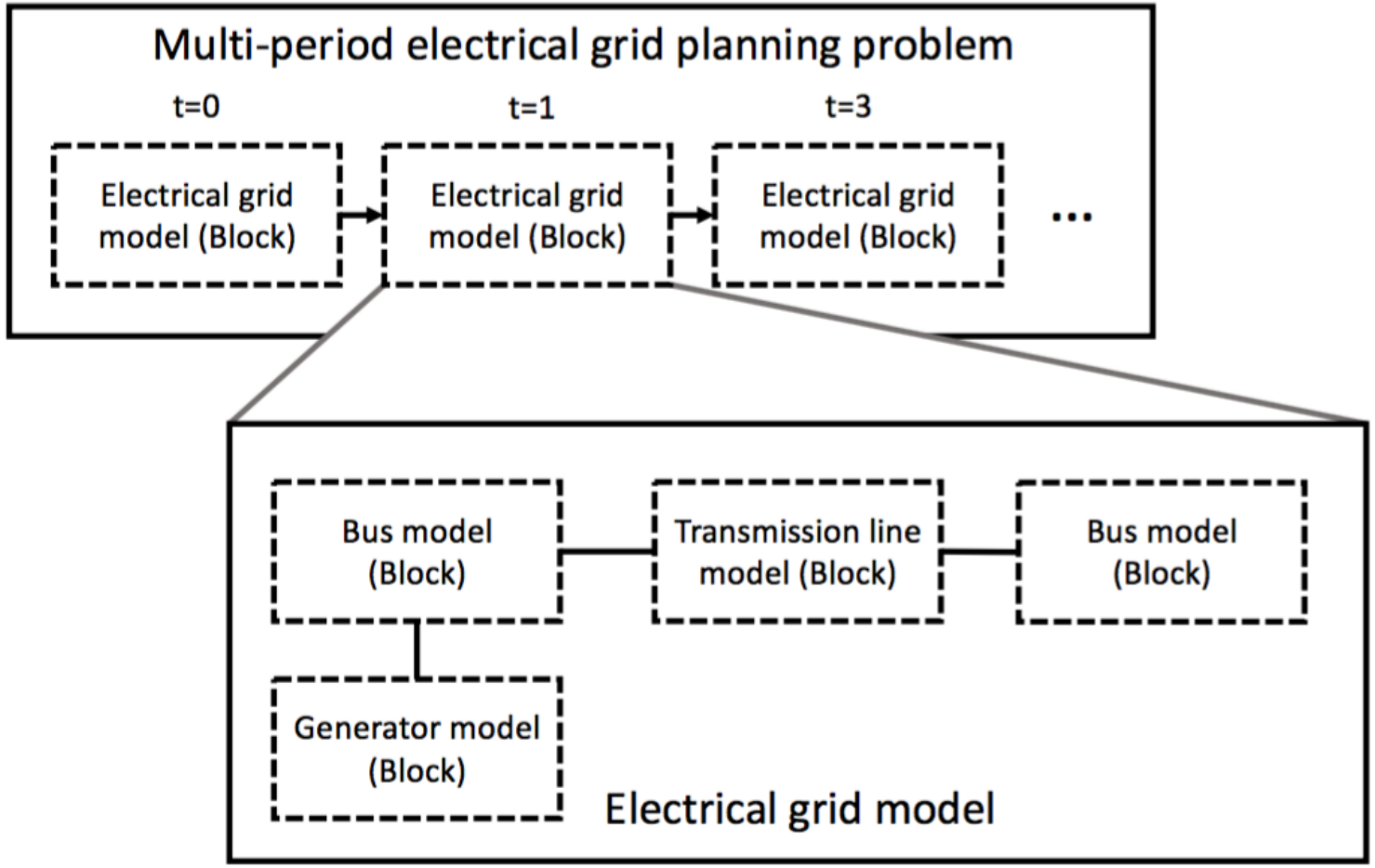
def xyb_rule(b, t):
    d = pyo.Block(concrete=True)
    d.x = pyo.Var()
    d.I = pyo.RangeSet(t)
    d.y = pyo.Var(d.I, initialize=1.0)
    d.c = pyo.Constraint(expr= d.x == 1.0 - sum(d.y[i] for i in d.I))
    return d
model.xyb = pyo.Block(model.T, rule=xyb_rule)
```

Block rules get the *block to be populated* as the first argument

Block rules can *ignore* the block that was passed in and return an independent block back. If you do this, Pyomo will take everything off the block you return and transfer it to the block we wanted populated.

Object-Oriented Model Composition

- Blocks enable building *model libraries*
- Individual blocks capture discrete aspects of the problem, usually corresponding to physical components, processes, or concepts
- Blocks are "connected together" using linking constraints (or other higher-level constraints; e.g. `pyomo.network`)



(Complete!) Solution

```
import pyomo.environ as pyo
import dcopf_decl as dcopf
from uc_example_data import data

model = pyo.ConcreteModel()
model.T = pyo.Set(initialize=range(2), ordered=True)
model.G = pyo.Set(initialize=sorted(data['gens'].keys()), ordered=True)

# create blocks of the dcopf model for each time period
def period_rule(b, t):
    return dcopf.create_dcopf_model(data[t])
model.period = pyo.Block(model.T, rule=period_rule)

# create the ramping constraints between time periods
def ramp_con_rule(m, t, g):
    if t == m.T.first():
        return pyo.Constraint.Skip
    return (-15.0, m.period[t-1].pg[g] - m.period[t].pg[g], 15.0)
model.ramp_con = pyo.Constraint(model.T, model.G, rule=ramp_con_rule)

# create the new objective (sum over time)
model.period[:].objective.deactivate()
model.obj = pyo.Objective(expr=sum(model.period[:].objective.expr))

# solve the new multi-period model
solver = pyo.SolverFactory('ipopt')
solver.solve(model, tee=True)

# print the solution
model.period[:].pg.display()
```

(Complete!) Solution

```
import pyomo.environ as pyo
import dcopf_decl as dcopf
from uc_example_data import data

model = pyo.ConcreteModel()
model.T = pyo.Set(initialize=range(2), ordered=True)
model.G = pyo.Set(initialize=sorted(data['gens'].keys()), ordered=True)

# create blocks of the dcopf model for each time period
def period_rule(b, t):
    return dcopf.create_dcopf_model(data[t])
model.period = pyo.Block(model.T, rule=period_rule)

# create the ramping constraints between time periods
def ramp_con_rule(m, t, g):
    if t == m.T.first():
        return pyo.Constraint.Skip
    return (-15.0, m.period[t-1].pg[g] - m.period[t].pg[g], 15.0)
```


```
import dcopf_decl as dcopf
from uc_example_data import data
```


```
model.obj = pyo.Objective(expr=sum(model.period[:].objective.expr))

# solve the new multi-period model
solver = pyo.SolverFactory('ipopt')
solver.solve(model, tee=True)

# print the solution
model.period[:].pg.display()
```

(Complete!) Solution

```
import pyomo.environ as pyo
import dcopf_decl as dcopf
from uc_example_data import data

model = pyo.ConcreteModel()
model.T = pyo.Set(initialize=range(2), ordered=True)
model.G = pyo.Set(initialize=sorted(data['gens'].keys()), ordered=True)

# create blocks of the dcopf model for each time period
def period_rule(b, t):
    return dcopf.create_dcopf_model(data[t])
model.period = pyo.Block(model.T, rule=period_rule)

# create the ramping constraints between time periods
def ramp_con_rule(m, t, g):
    if t == m.T.first():
        return pyo.Constraint.Skip
    return (-15.0, m.period[t-1].pg[g] - m.period[t].pg[g], 15.0)
```


```
# create blocks of the dcopf model for each time period
def period_rule(b, t):
    return dcopf.create_dcopf_model(data[t])
model.period = pyo.Block(model.T, rule=period_rule)
```


```
solver = pyo.SolverFactory('ipopt')
solver.solve(model, tee=True)

# print the solution
model.period[:].pg.display()
```

(Complete!) Solution

```
import pyomo.environ as pyo

# create the ramping constraints between time periods
def ramp_con_rule(m, t, g):
    if t == m.T.first():
        return pyo.Constraint.Skip
    return (-15.0, m.period[t-1].pg[g] - m.period[t].pg[g], 15.0)
model.ramp_con = pyo.Constraint(model.T, model.G, rule=ramp_con_rule)
```

```
model.period = pyo.Block(model.T, rule=period_rule)

# create the ramping constraints between time periods
def ramp_con_rule(m, t, g):
    if t == m.T.first():
        return pyo.Constraint.Skip
    return (-15.0, m.period[t-1].pg[g] - m.period[t].pg[g], 15.0)
model.ramp_con = pyo.Constraint(model.T, model.G, rule=ramp_con_rule)

# create the new objective (sum over time)
model.period[:].objective.deactivate()
model.obj = pyo.Objective(expr=sum(model.period[:].objective.expr))

# solve the new multi-period model
solver = pyo.SolverFactory('ipopt')
solver.solve(model, tee=True)

# print the solution
model.period[:].pg.display()
```

(Complete!) Solution

```
import pyomo.environ as pyo
```

```
# create the new objective (sum over time)
model.period[:].objective.deactivate()
model.obj = pyo.Objective(expr=sum(model.period[:].objective.expr))
```

```
# create blocks of the dcopf model for each time period
def period_rule(b, t):
    return dcopf.create_dcopf_model(data[t])
model.period = pyo.Block(model.T, rule=period_rule)

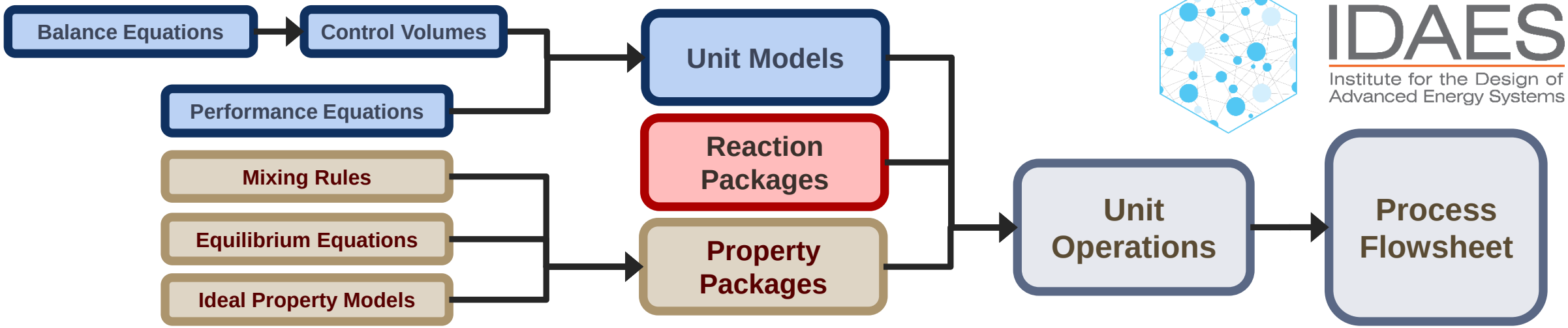
# create the ramping constraints between time periods
def ramp_con_rule(m, t, g):
    if t == m.T.first():
        return pyo.Constraint.Skip
    return (-15.0, m.period[t-1].pg[g] - m.period[t].pg[g], 15.0)
model.ramp_con = pyo.Constraint(model.T, model.G, rule=ramp_con_rule)
```


```
# create the new objective (sum over time)
model.period[:].objective.deactivate()
model.obj = pyo.Objective(expr=sum(model.period[:].objective.expr))
```


```
# solve the new multi-period model
solver = pyo.SolverFactory('ipopt')
solver.solve(model, tee=True)

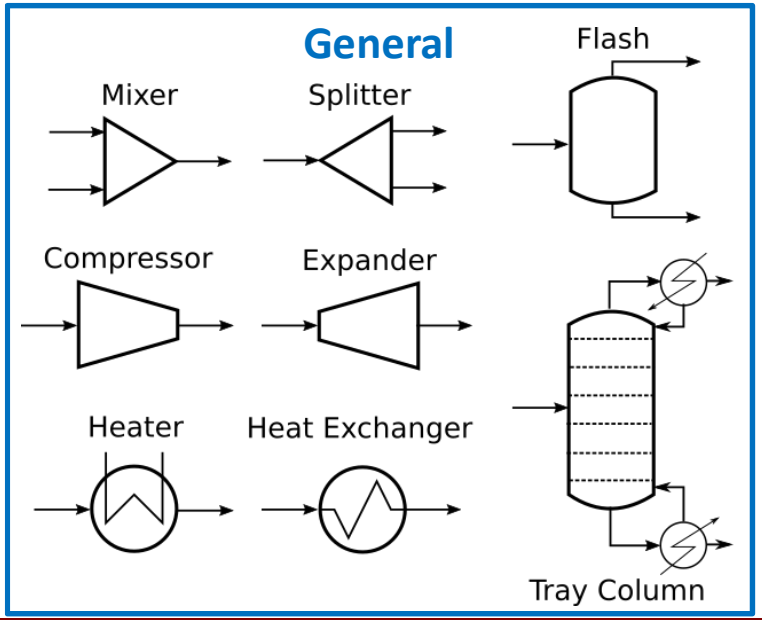
# print the solution
model.period[:].pg.display()
```

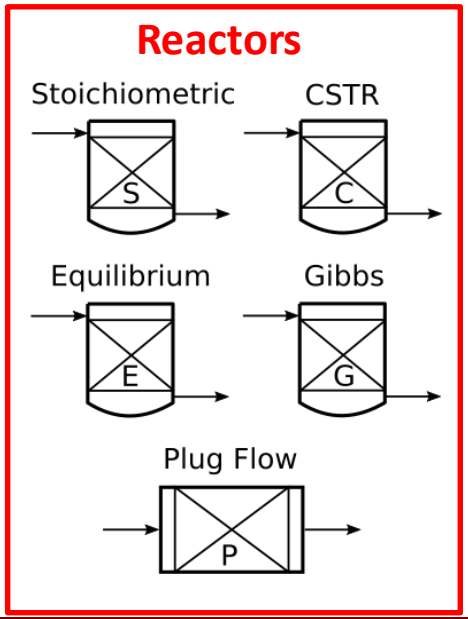
Building EO models without writing equations



IDAES
Institute for the Design of Advanced Energy Systems

- Unit operation models are fundamental process model building blocks
  - Numerous models, variants, subtypes
  - Different levels of rigor
  - Different ways to formulate equations
- Leverage unique Pyomo modeling capabilities (blocks, ports, arcs) to *compose* unit models (and flowsheets) from common modeling components





What does this mean for the user?

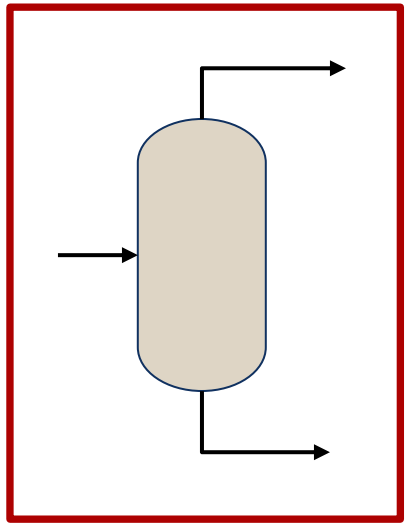
- The "hello, world" flowsheet: flash a 50/50 mixture of toluene and benzene @ 368K

```
from pyomo.environ import ConcreteModel, SolverFactory, units
from idaes.core import FlowsheetBlock
from idaes.generic_models.properties.activity_coeff_models.BTX_activity_coeff_VLE import BTXParameterBlock
from idaes.generic_models.unit_models import Flash

model = ConcreteModel()
model.fs = FlowsheetBlock(default={"dynamic": False})
model.fs.properties = BTXParameterBlock(default={
    "valid_phase": ('Liq', 'Vap'), "activity_coeff_model": "Ideal", "state_vars": "FTPz",
})

model.fs.flash = Flash(default={"property_package": model.fs.properties})

model.fs.flash.inlet.flow_mol.fix(1 * units.mol/units.s)
model.fs.flash.inlet.temperature.fix(368 * units.K)
model.fs.flash.inlet.pressure.fix(101325 * units.Pa)
model.fs.flash.inlet.mole_frac_comp[0, "benzene"].fix(0.5)
model.fs.flash.inlet.mole_frac_comp[0, "toluene"].fix(0.5)
model.fs.flash.heat_duty.fix(0 * units.J/units.s)
model.fs.flash.deltaP.fix(0 * units.Pa)
```



Pyomo Blocks: informing solvers

- Some modeling approaches are not algebraic
  - Consider selecting between sets of mutually infeasible constraints
    - Sequencing: "A before B or B before A"
    - Selection: build a unit or not
  - One representation is as a *logic model*
    - Using, e.g., disjunctive programming
    - Representing individual *disjuncts* (more in a later module)

- Some solvers can *directly leverage special structure*
  - Consider problem decomposition
    - Stochastic programs can be decomposed using *scenario-based decomposition*
    - Need to annotate "what is a scenario"?

Capturing logical relationships

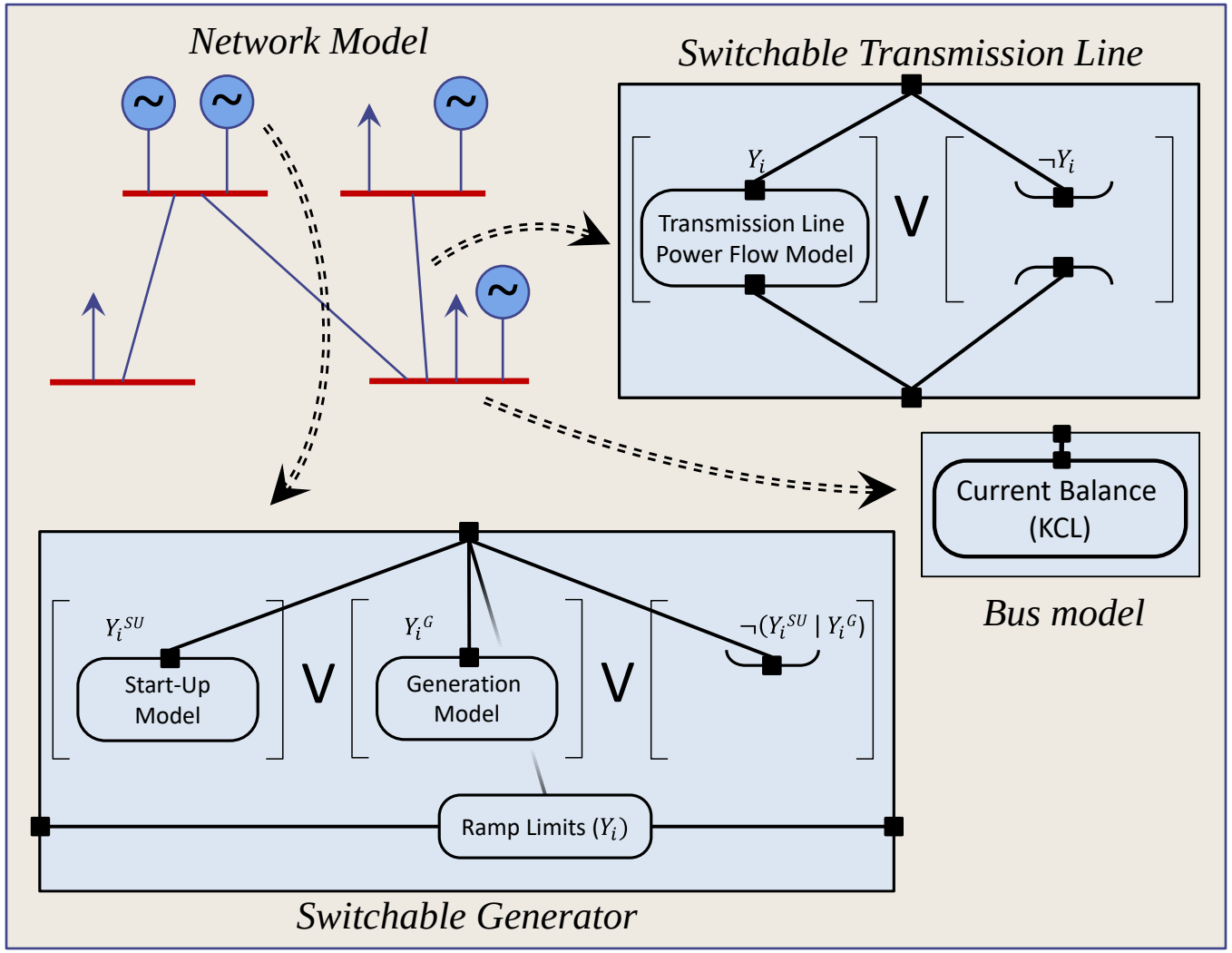
- Implement logical decisions through *disjunctive programming*
  - Transmission switching:

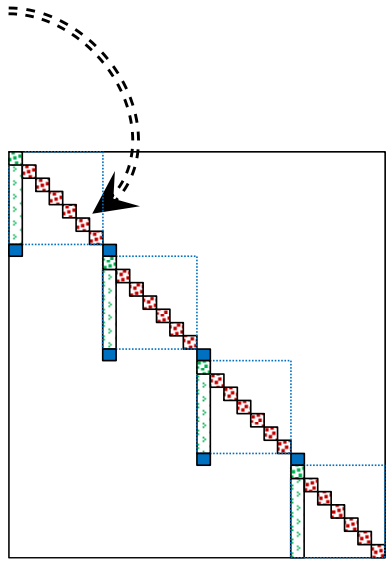
$$\begin{bmatrix} z_{kct} \\ P_{kct} = B_k(\theta_{k1} - \theta_{k2}) \end{bmatrix} \vee \begin{bmatrix} \neg z_{kct} \\ P_{kct} = 0 \end{bmatrix}$$

  - Generation

$$\begin{bmatrix} u_{gt} \\ C_{gt} = P_{gt} c_g \\ R_g^+ \geq P_{gt} - P_{gt-1} \\ R_g^- \geq P_{gt-1} - P_{gt} \end{bmatrix} \vee \begin{bmatrix} v_{kt} \\ C_{gt} = P_{gt} c_g + c_g^{SU} \\ R_g^{SU} \geq P_{gt} - P_{gt-1} \end{bmatrix} \vee \begin{bmatrix} \neg(u_{kt} \mid v_{kt}) \\ C_{gt} = c_g^{SD} u_{kt-1} \\ R_g^{SD} \geq P_{gt-1} - P_{gt} \\ P_{gt} = 0 \end{bmatrix}$$

Embed within a structured model





Pyomo Blocks

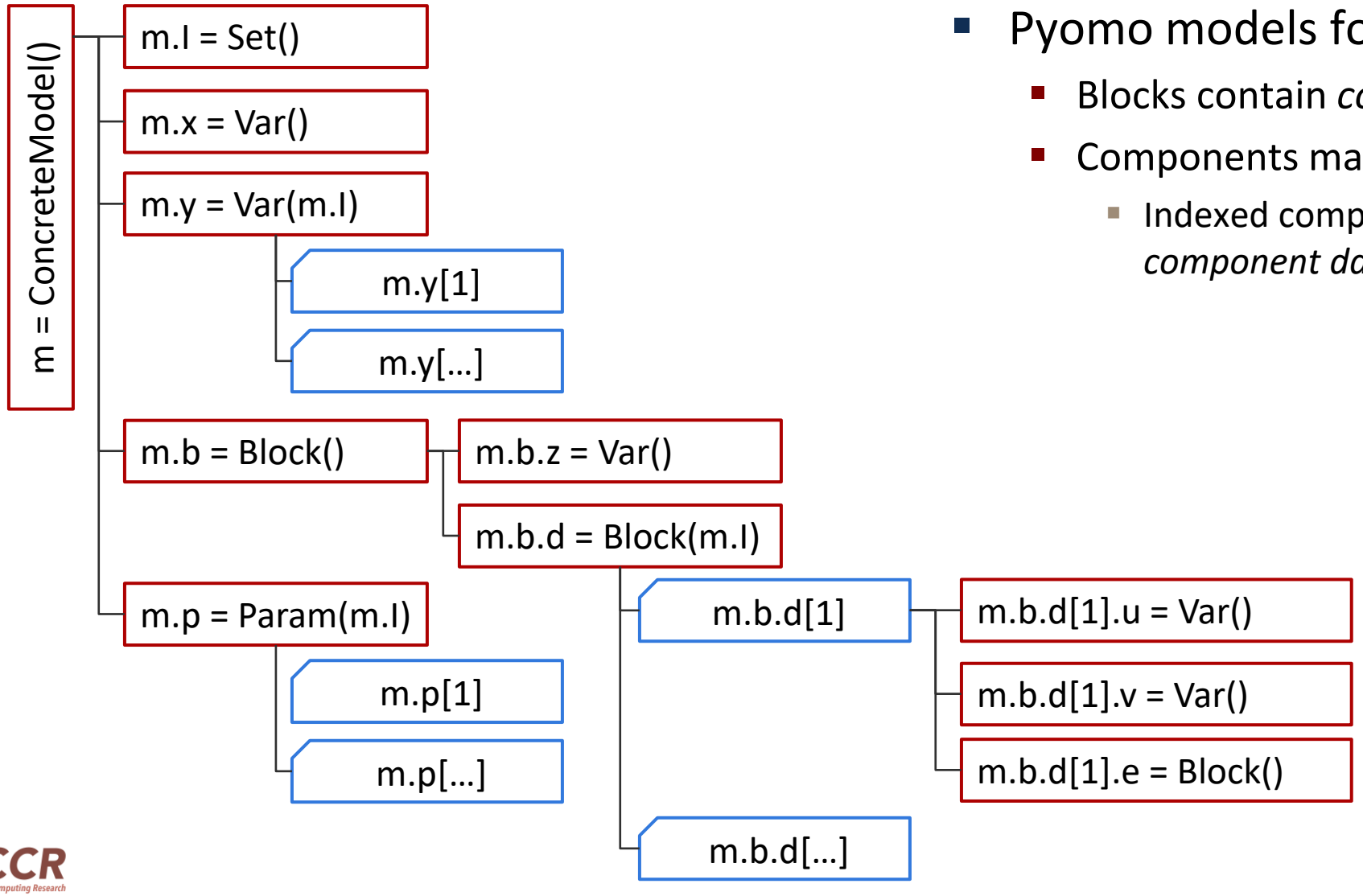
- The Pyomo ”Block” allows us to construct hierarchical models
- A Block can be treated in much the same way as a model
  - Modeling components can be added to blocks (e.g., Var, Constraint)
  - Blocks can be added to other Blocks

```
model = pyo.ConcreteModel()
model.I = pyo.Set()
model.x = pyo.Var()
model.y = pyo.Var(m.I)
model.p = pyo.Param(m.I)

model.b = pyo.Block()
model.b.z = pyo.Var()

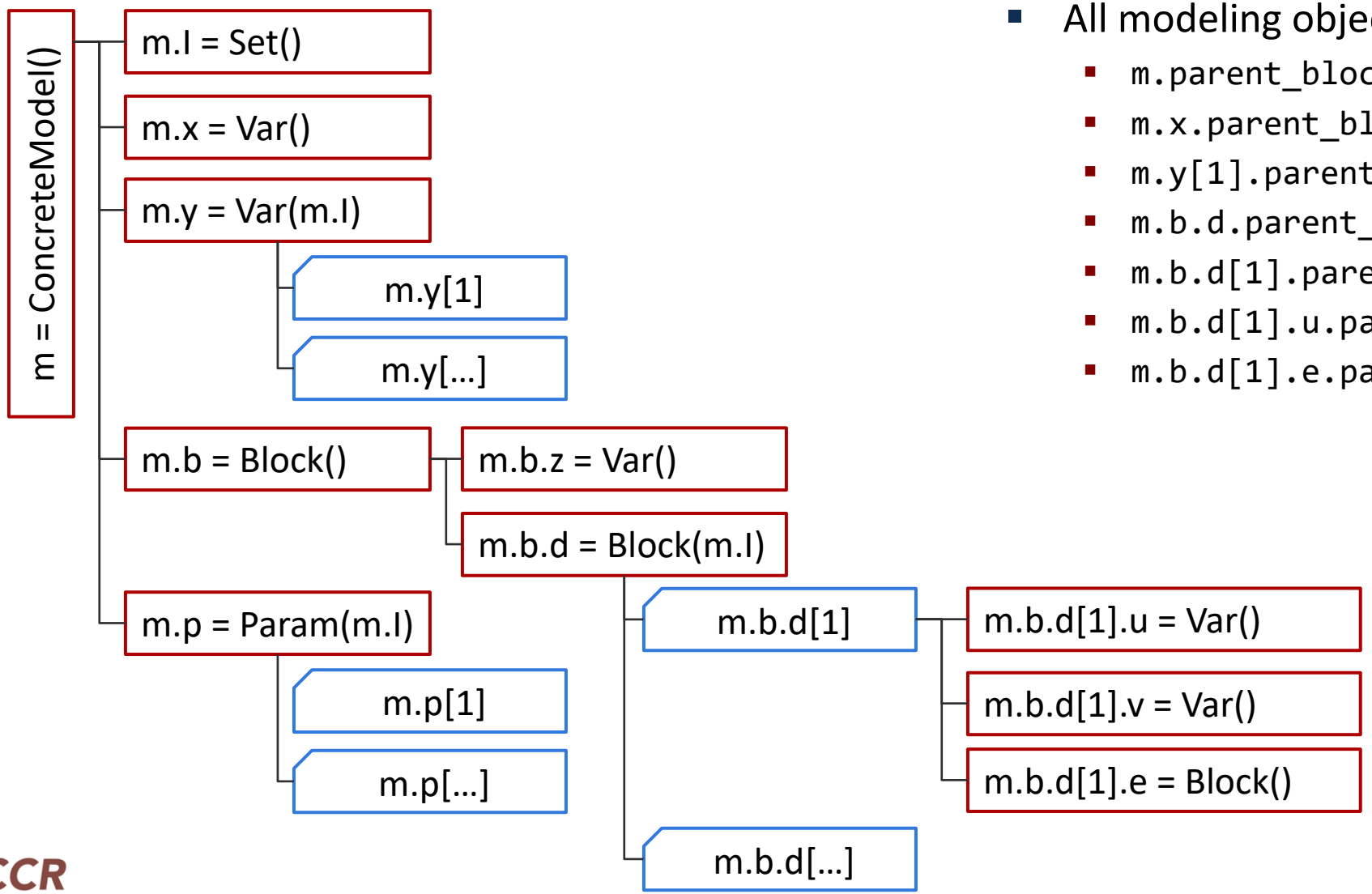
model.b.d = pyo.Block(m.I)
model.b.d[1].u = pyo.Var(m.I)
model.b.d[1].v = pyo.Var()
model.b.d[1].e = pyo.Block()
```

Working with the Block Hierarchy



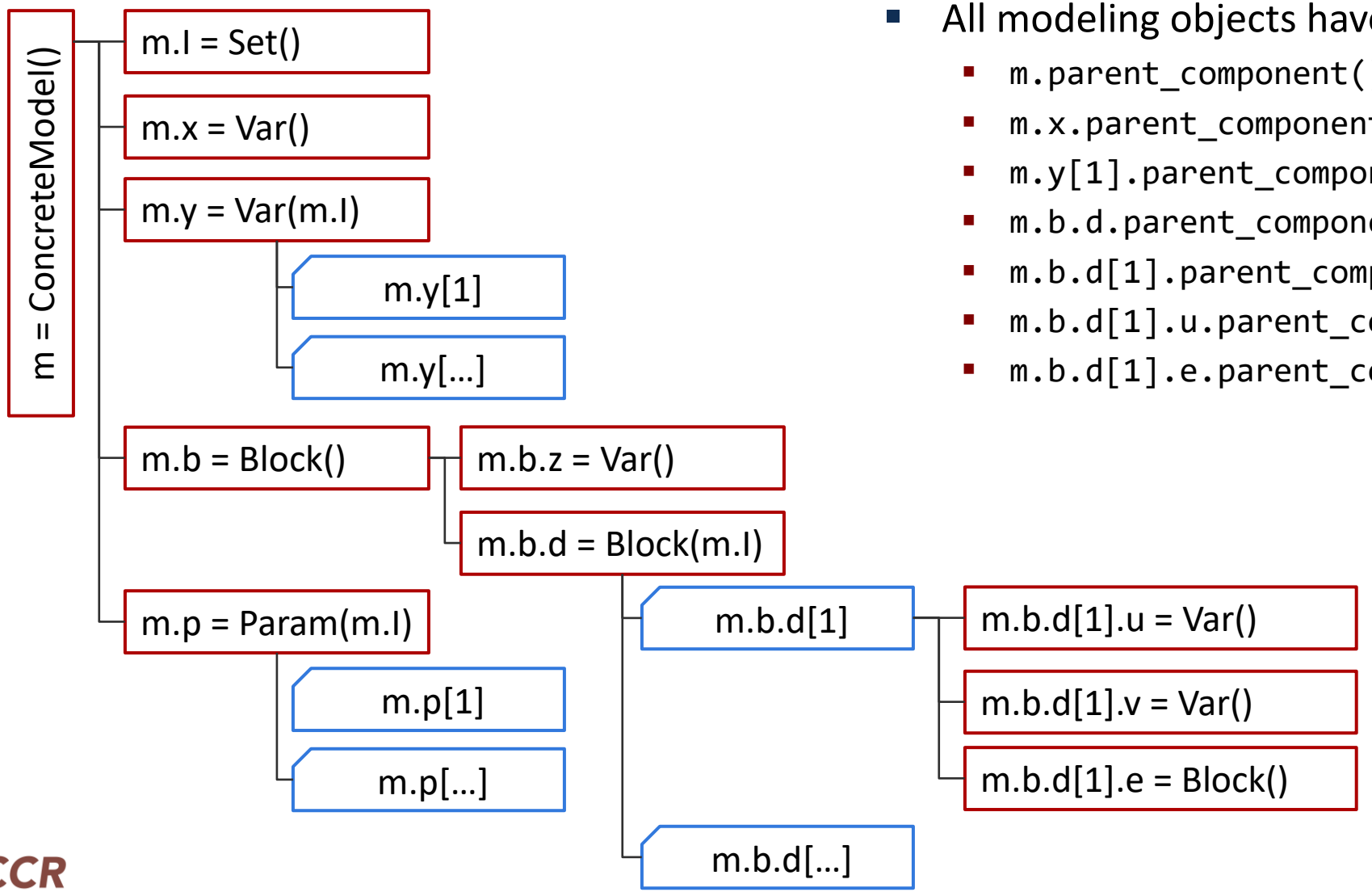
- Pyomo models form a *tree*:
  - Blocks contain *components*
  - Components may be simple (scalar) or indexed
    - Indexed components contain individual *component data* objects

Working with the Block Hierarchy



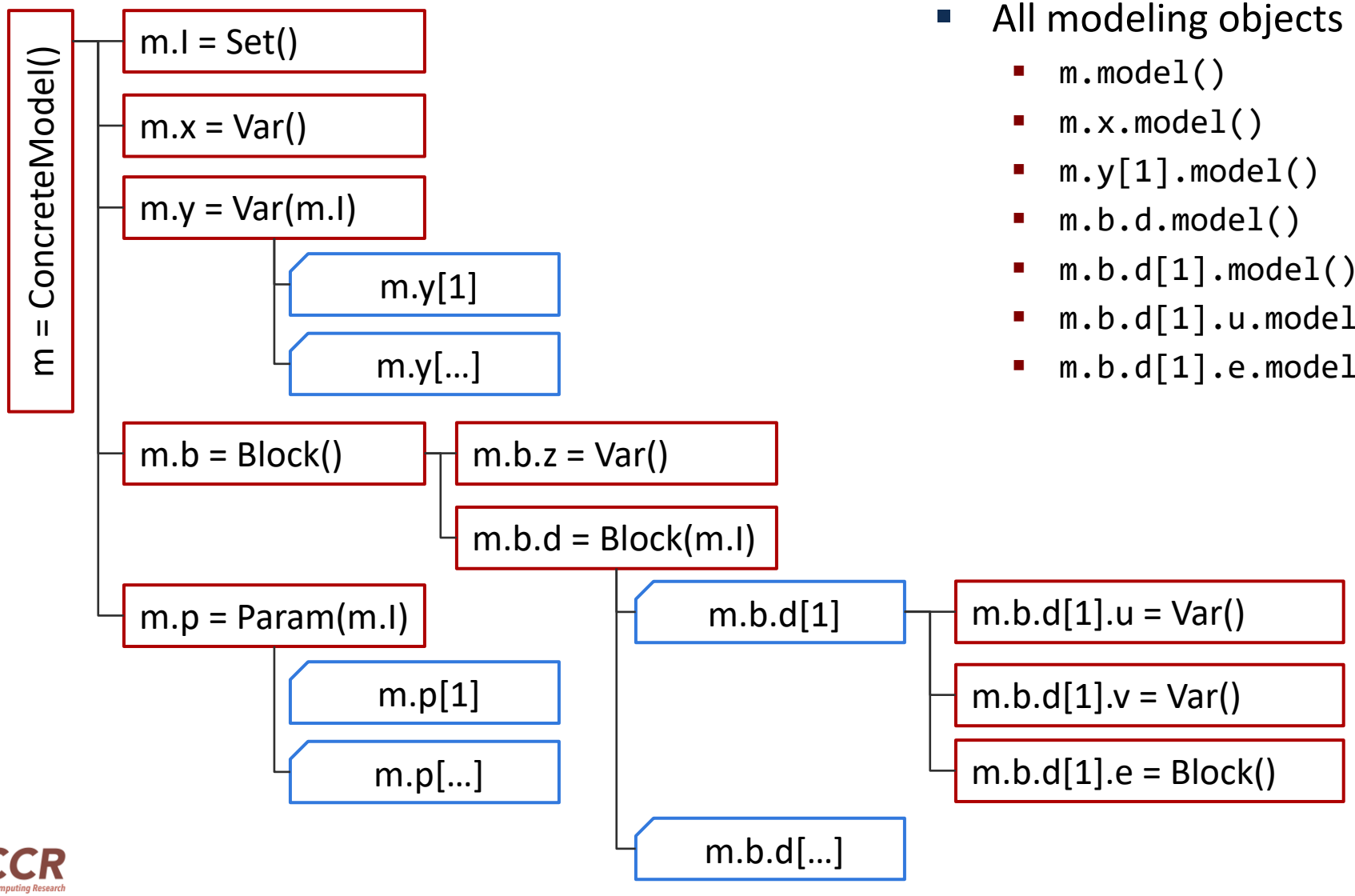
- All modeling objects have a *parent block*:
  - `m.parent_block()` `is None`
  - `m.x.parent_block()` `is m`
  - `m.y[1].parent_block()` `is m`
  - `m.b.d.parent_block()` `is m.b`
  - `m.b.d[1].parent_block()` `is m.b`
  - `m.b.d[1].u.parent_block()` `is m.b.d[1]`
  - `m.b.d[1].e.parent_block()` `is m.b.d[1]`

Working with the Block Hierarchy



- All modeling objects have a *parent component*:
  - `m.parent_component()` `is m`
  - `m.x.parent_component()` `is m.x`
  - `m.y[1].parent_component()` `is m.y`
  - `m.b.d.parent_component()` `is m.b.d`
  - `m.b.d[1].parent_component()` `is m.b.d`
  - `m.b.d[1].u.parent_component()` `is m.b.d[1].u`
  - `m.b.d[1].e.parent_component()` `is m.b.d[1].e`

Working with the Block Hierarchy



- All modeling objects have a *model*:
  - `m.model()` is `m`
  - `m.x.model()` is `m`
  - `m.y[1].model()` is `m`
  - `m.b.d.model()` is `m`
  - `m.b.d[1].model()` is `m`
  - `m.b.d[1].u.model()` is `m`
  - `m.b.d[1].e.model()` is `m`

How to find components in (deep) hierarchies?

- A common need is to find all of the components of a certain type
  - `block.component_objects(ctype, active, sort, descend_into, descent_order)`
    - `ctype`: the "declared component type" that we are looking for (can be a list)
    - `active`: if True, only return "active" components
    - `sort`: options to control the ordering of the components
    - `descend_into`: ctype (or list of ctypes) that define the component types that define the block tree
    - `descent_order`: options to specift the algorithm for walking the tree
- Frequently, we are not interested in the components (containers), but the individual *data objects*
  - `block.component_data_objects(...)`

Accessing objects within blocks

- Accessing components within Blocks
  - Recall:

```
m.xyb = Block(m.T)
for t in m.T:
    m.xyb[t].I = RangeSet(t)
    m.xyb[t].y = Var(m.xyb[t].I)
```

  - Looping to print variables

```
for t in m.xyb:
    for i in m.xyb[t].y:
        print("%s %f" % (m.xyb[t].y[i], value(m.xyb[t].y[i])
```

- Slicing for convenient iteration
  - Pyomo supports a special "slice-like" wildcard notation
    - ":" is a wildcard matching a single index
    - "..." is a wildcard that matches 0 or more indices

```
for v in model.xyb[:].y[:]:
    print("%s %f" % v, value(v))
```

Slicing can apply methods

- Looping over slices and applying methods is *very* common

```
y_val = []
for v in model.xyb[:].y[:]:
    y_val.append(value(v))
```

- Slices can apply methods / access attributes:
  - Extract all values:

```
y_val = list(model.xyb[:].y[:].value)
```

  - Set all values (to the same value):

```
model.xyb[:].y[:].set_value(5)
```

(or)

```
model.xyb[:].y[:] = 5
```

Sometimes slicing is inconvenient

- Consider

```
m.b = Block([1,2,3])
for i in [1,2,3]:
    m.b[i].x = Var(initialize=i)
m.b[:].x.pprint()
```

- Results in:

```
x : Size=1, Index=None
    Key  : Lower : Value : Upper : Fixed : Stale : Domain
    None :  None :     1 :  None : False : False :  Reals
x : Size=1, Index=None
    Key  : Lower : Value : Upper : Fixed : Stale : Domain
    None :  None :     2 :  None : False : False :  Reals
x : Size=1, Index=None
    Key  : Lower : Value : Upper : Fixed : Stale : Domain
    None :  None :     3 :  None : False : False :  Reals
[None, None, None]
```

References: views into slices

- Consider

```
m.b = Block([1,2,3])
for i in [1,2,3]:
    m.b[i].x = Var(initialize=i)
X = Reference(m.b[:].x)
X.pprint()
```

- Results in:

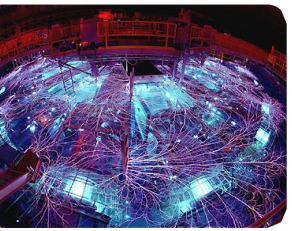
```
IndexedVar : Size=3, Index=b_index
    Key : Lower : Value : Upper : Fixed : Stale : Domain
      1 :  None :     1 :  None : False : False :  Reals
      2 :  None :     2 :  None : False : False :  Reals
      3 :  None :     3 :  None : False : False :  Reals
```

Note: Reference() creates a new Pyomo component with the referents' type (by default). If you add it to your model, it will cause the referents to appear to exist twice.

Not a huge issue for Vars, but can cause problems for Constraints

Why Blocks?

- We use blocks in three common situations
  - "Namespacing"
    - Insulating sub-model development from each other
    - Simplify the process of systematically manipulating a model
  - Object-oriented modeling
    - Getting the model to more directly represent physical system
  - Informing the solution process
    - Representing non-algebraic concepts
    - Annotate the model to assist automated solution procedures



Sandia National Laboratories

Exceptional service in the national interest

Section 7
Model Transformations

U.S. DEPARTMENT OF ENERGY
NNSA National Nuclear Security Administration
CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

What do these have in common?

$$a = \sqrt{(x-3)^2+\epsilon}$$

$$\begin{aligned} a &= b + c \\ b &\leq M \cdot y \\ c &\leq M(1-y) \\ x-3 &= c-b \\ b &\geq 0 \\ c &\geq 0 \\ y &\in \{0,1\} \end{aligned}$$

$$a = |x-3|$$

$$\begin{aligned} a &\geq x-3 \\ a &\geq 3-x \end{aligned}$$

$$\begin{aligned} a &= b + c \\ x-3 &= c-b \\ b \geq 0 &\perp c \geq 0 \end{aligned}$$

$$a = \frac{2(x-3)}{1+e^{-\frac{x-3}{h}}} - x + 3$$

If we *mean* "$a = |x-3|$", why don't we *write* that in our models?

Models vs. Formulations

- We often refer to this as an *optimization model*:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R}^n \end{aligned}$$

- But this is only one of many possible *formulations* that represents that model.
- Another rather famous one that could be used in this case is:

$$\begin{aligned} \min \quad & a^T y \\ \text{s.t.} \quad & By = d \\ & y \geq 0 \end{aligned}$$

Models are for *Modelers*

- So, what's an *optimization model*?
  - A general representation of a class of optimization problems
    - Data (instance) independent
  - Represents the modeler's understanding of the class of problems
    - Explicitly annotates and conveys the class structure
    - Valid representation of the problem the modeler aims to solve
  - Incorporates assumptions and simplifications

Models are for *Modelers*

- So, what's an *optimization model*?
  - A general representation of a class of optimization problems
    - Data (instance) independent
  - Represents the modeler's understanding of the class of problems
    - Explicitly annotates and conveys the class structure
    - Valid representation of the problem the modeler aims to solve
  - Incorporates assumptions and simplifications
- …And what is a *formulation?*
  - A particular mathematical representation of a model
    - E.g., standard form linear program, Big-M representation of a disjunction, etc.
  - We typically like these tractable, i.e., we choose a formulation we think we will be able to solve.

Models are for *Modelers*

- So, what's an *optimization model*?
  - A general representation of a class of optimization problems
    - Data (instance) independent
  - Represents the modeler's understanding of the class of problems
    - Explicitly annotates and conveys the class structure
    - Valid representation of the problem the modeler aims to solve
  - Incorporates assumptions and simplifications
- …And what is a *formulation?*
  - A particular mathematical representation of a model
    - E.g., standard form linear program, Big-M representation of a disjunction, etc.
  - We typically like these tractable, i.e., we choose a formulation we think we will be able to solve.

What's in a Model?

- Some modelers don't think in terms of "A":
  - Can think in terms of sets and repeated (indexed) variables, constraints, etc.
  - Groups of related constraints (Blocks):
    - Block diagonal structure (e.g., scenarios in stochastic programming)
    - Graph-based (e.g., network flow)
    - Hierarchically defined (e.g., a model composed of sub-models)

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R}^n \end{aligned}$$

What's in a Model?

- Some modelers don't think in terms of "A":
  - Can think in terms of sets and repeated (indexed) variables, constraints, etc.
  - Groups of related constraints (Blocks):
    - Block diagonal structure (e.g., scenarios in stochastic programming)
    - Graph-based (e.g., network flow)
    - Hierarchically defined (e.g., a model composed of sub-models)
- Some modelers don't even think terms of algebra, e.g.:
  - Generalized Disjunctive Programming (GDP)
  - Differential Algebraic Equations (DAE)
  - Constraint Programming (CP)

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R}^n \end{aligned}$$

What's in a Model?

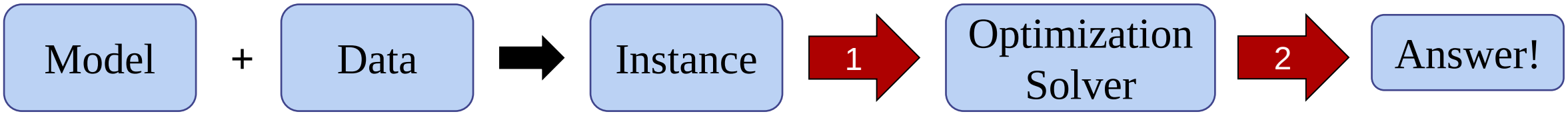
- Some modelers don't think in terms of "A":
  - Can think in terms of sets and repeated (indexed) variables, constraints, etc.
  - Groups of related constraints (Blocks):
    - Block diagonal structure (e.g., scenarios in stochastic programming)
    - Graph-based (e.g., network flow)
    - Hierarchically defined (e.g., a model composed of sub-models)

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R}^n \end{aligned}$$

- Some modelers don't even think terms of algebra, e.g.:
  - Generalized Disjunctive Programming (GDP)
  - Differential Algebraic Equations (DAE)
  - Constraint Programming (CP)
- Most solvers, and some modelers and algorithm developers **do** think in terms of "A":
  - Vectors of variables and parameters, matrices of parameters

Do You Speak Solver?

- If you found yourself in one of the first two categories, probably not…



1. What *do* solvers speak? Depends on the solver:
   - Does your instance need to be linear?
   - Does it need to be continuous?
   - Does your instance need to be algebraic?
   - Can it have logical structures?
2. For difficult instances, to get answers, we need to speak solver *well:*
   - Well-scaled representation
   - Well-structured representation
   - Sparse representation
   - Tight representation

Transformations are for getting from your (intuitive, modeler-friendly) model instance to a (hopefully) tractable formulation that your solver understands and performs well on.

Example

- Consider:

$$\begin{aligned} \max \quad & |x - 3| \\ \text{s.t.} \quad & x \in X \end{aligned}$$

- In an integer programming class, you learn that this is MIP-representable:

$$\begin{aligned} \max \quad & z \\ \text{s.t.} \quad & z = x^+ + x^- \\ & x^+ \leq Uy \\ & x^- \leq |L|(1-y) \\ & x - 3 = x^+ - x^- \\ & x^+ \geq 0 \\ & x^- \geq 0 \\ & y \in \{0,1\} \\ & x \in X \end{aligned}$$

Example

- Consider:

$$\begin{array}{ll} \max & |x-3| \\ \text{s.t.} & x \in X \end{array}$$

- In an integer programming class, you learn that this is MIP-representable:

$$\begin{array}{ll} \max & z \\ \text{s.t.} & z = x^+ + x^- \\ & x^+ \leq Uy \\ & x^- \leq |L|(1-y) \\ & x - 3 = x^+ - x^- \\ & x^+ \geq 0 \\ & x^- \geq 0 \\ & y \in \{0,1\} \\ & x \in X \end{array}$$

- But what if X isn't polyhedral?
- It might make sense to write:
  - $z = \sqrt{(x-3)^2+\epsilon}$, or
  - $z = \frac{2(x-3)}{1+e^{-(x-3)/h}} - x + 3$

Example

- Consider:

$$\begin{array}{ll} \max & |x-3| \\ \text{s.t.} & x \in X \end{array}$$

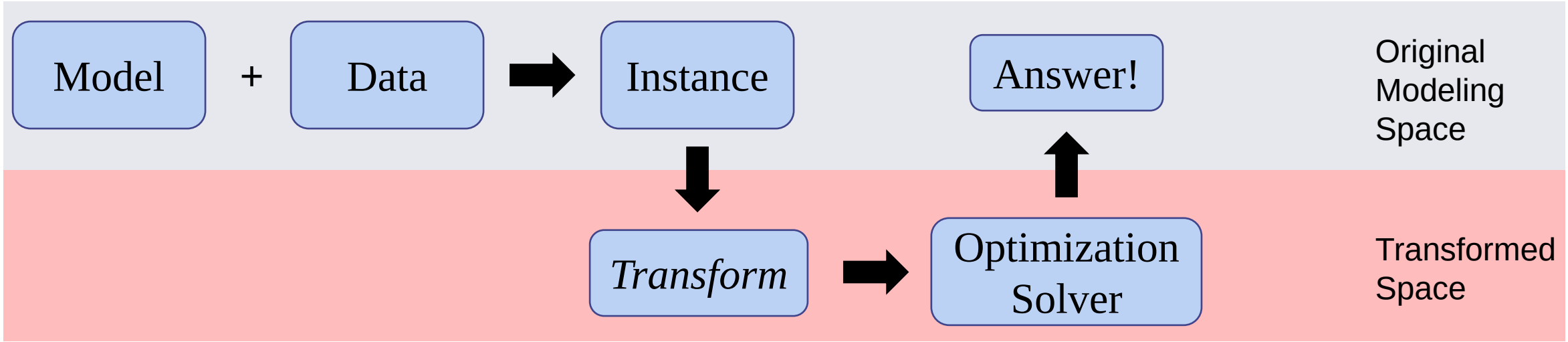
- In an integer programming class, you learn that this is MIP-representable:

$$\begin{array}{ll} \max & z \\ \text{s.t.} & z = x^+ + x^- \\ & x^+ \leq Uy \\ & x^- \leq |L|(1-y) \\ & x - 3 = x^+ - x^- \\ & x^+ \geq 0 \\ & x^- \geq 0 \\ & y \in \{0,1\} \\ & x \in X \end{array}$$

- But what if X isn't polyhedral?
- It might make sense to write:
  - $z = \sqrt{(x-3)^2 + \epsilon}$, or
  - $z = \frac{2(x-3)}{1+e^{-(x-3)/h}} - x + 3$

- NOTE: All of these machinations are about making the *solver* happy. *We* understood the problem when we wrote the original formulation
  - Transformations allow us to separate modeling from the process of finding a good formulation

The Point



- Separate model expression from how we intend to solve it
  - Support non-algebraic modeling constructs (e.g., Piecewise expressions, GDP, DAE, etc.)
  - Defer decisions that improve tractability until solution time
  - Explore alternative reformulations or representations
  - Support *solver-specific* modeling constructs (e.g., indicator constraints)
  - Support iterative methods that use different solvers requiring different representations (e.g., initializing NLP from MIP)
- Reduce "mechanical" errors due to manual transformation

Growing Library of Transformations

- Disjunctive programming
  - Big-M reformulation
  - Hull reformulation
  - Cutting planes-based strengthened Big-M
  - Hybrid Basic-Step based algorithm
  - Transform current disjunctive state
  - Between steps
  - Bound “pre-transformation”
- Dynamic systems
  - Collocation on finite elements
  - Finite difference discretization
- Logical Models
  - Logical to conjunctive normal form
  - Logical to disjunctive form
- Complementarity / Equilibrium constraints
  - Nonlinear relaxation
  - Disjunctive relaxation
  - “Standard” form relaxation
- Structural transformations
  - Relax discrete variables
  - Standard linear form
  - Dual transformation
  - Fix discrete variables
  - Nonnegative variables
  - Expand connectors
  - Add slack variables
- Contributed transformations
  - Constraints to var bounds
  - Deactivate trivial constraints
  - Detect implicitly fixed vars
  - Variable initialization
  - Remove zero terms
  - Propagate var bounds, fixed flags
  - Projection via Fourier-Motzkin elimination

Applying Transformations

- Transformations can be retrieved from a global registry

```
import pyomo.environ as pyo
transform = pyo.TransformationFactory('core.linear_dual')
```

- Transformations can be applied
  - "in place" (modifying the model passed in)

```
transform.apply_to(model)
```

  - "out of place" (return a new model leaving original unchanged)

```
new_model = transform.create_using(model)
```

Failing to Apply Transformations

- This is what it might look like when you forget to transform something that cannot be written to the solver you are using:

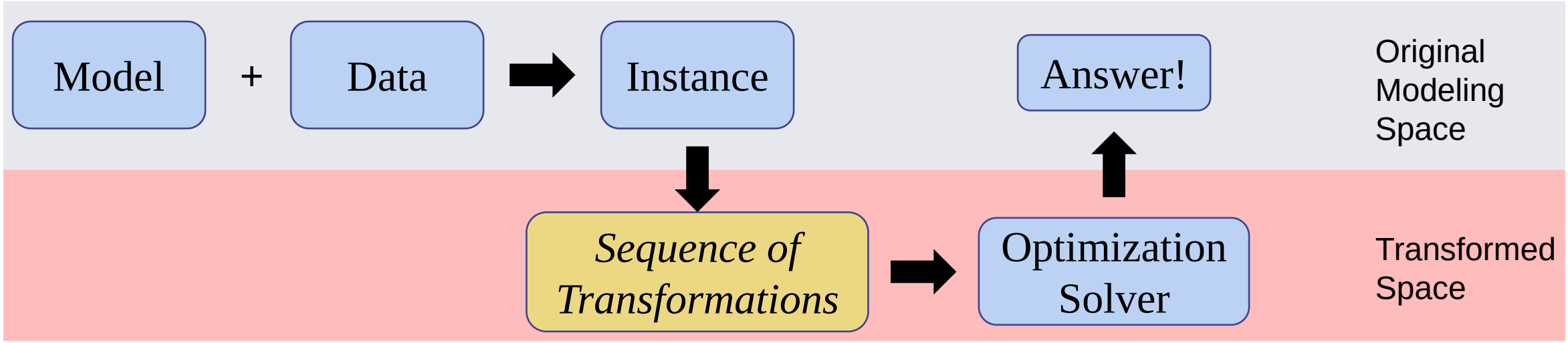
```
ValueError: The model ('unknown') contains the following active components that the LP
writer does not know how to process:
    <class 'pyomo.gdp.disjunct.Disjunct'>:
            d_disjuncts[0]
            d_disjuncts[1]
    <class 'pyomo.gdp.disjunct.Disjunction'>:
            d
```

Translating Solutions from Transformed Models

- How do you translate a solution to a transformed instance back into your original instance?
- This is still a work in progress, in many cases.
- Several ways we do (or don't do) it:

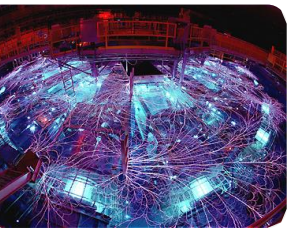
1. Transformation uses original variables or we rely on magic (Autolinking)	• GDP transformations (of models without non-indicator BooleanVars) • `contrib.preprocessing` transformations
2. Explicit Function to Make Translation	• `core.logical_to_linear` and `contrib.logical_to_disjunctive` require a call to `update_boolean_vars_from_binary` • GDP transformations *with* non-indicator BooleanVars do, too
3. Transformation solution doesn't map to original formulation, i.e., transformation is a new (not equivalent) model	• DAE transformations • `core.add_slack_variables` • `core.relax_integer_vars`

A Word to the Wise



- Some transformations are intended to be intermediate:
  - We would rather chain together simple transformations than make "universal" transformations that can do everything
  - Some transformations don't get you all the way to something you can send to the solver (e.g. `gdp.bound_pretransformation`)
- When chaining transformations, not all sequences make sense.

➢ If you're going to further modify a model that has been transformed, you need to understand what the transformation(s) did!



Sandia National Laboratories

Exceptional service in the national interest

Section 8
Dynamic Systems

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration | CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

Examples of Dynamic Optimization

- Parameter Estimation
- Nonlinear model predictive control
- Batch process operation
- Reactor design

$$\min \Psi(x(t), y(t), u(t))$$

$$\dot{x}(t) = f(x(t), y(t), u(t), t, p)$$

DAE model

$$0 = g(x(t), y(t), u(t), t, p)$$

$$x(t) \in \Re^n$$

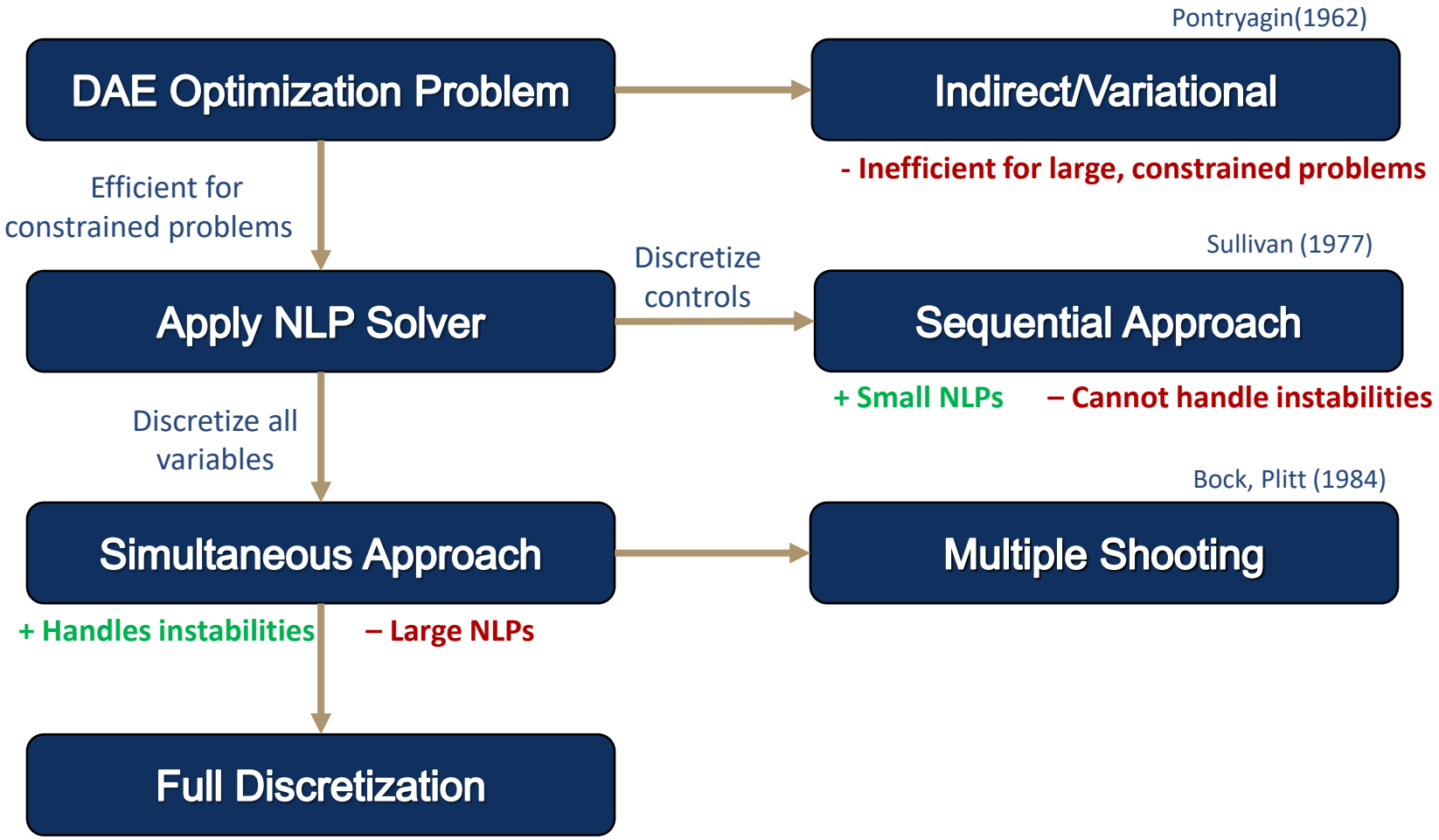
$$y(t) \in \Re^n$$

$$u(t) \in \Re^m$$

x: State (differential) variables
u: Control (input) variables
y: Algebraic variables

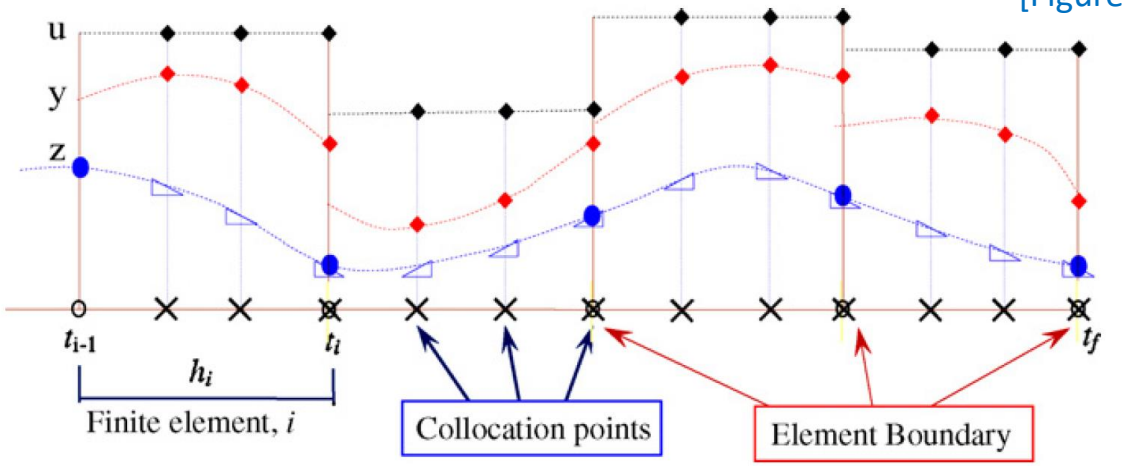
Solution Approaches

[Figure from L.T. Biegler (2007)]



Simultaneous Approach

[Figure from L.T. Biegler (2007)]



- Multiple Shooting
  - Discretize controls and initial conditions for each finite element
  - ✓ Embeds DAE Solvers/Sensitivity
  - ✓ Can solve unstable systems
  - ✓ Good for problems with long time horizons and few dynamic states
  - X Dense sensitivity blocks
  - X Difficult to enforce path constraints
- Full Discretization
  - Discretize all variables
  - ✓ Can solve unstable systems
  - ✓ Good for problems with many dynamic states and degrees of freedom
  - ✓ Sparse NLP
  - X Large-scale NLP

Full Discretization

- Finite Difference Methods

$$\frac{df}{dt}(t) = \lim_{h \to 0} \frac{f(t+h) - f(t)}{h}$$

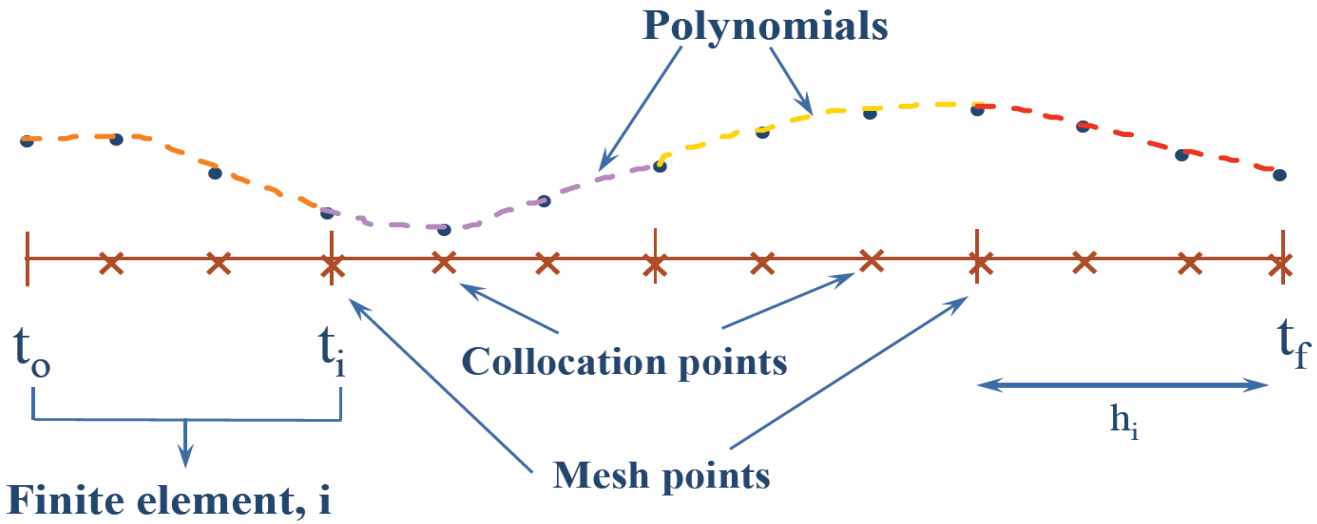
Forward Difference

$$\frac{df}{dt}(t) = \frac{f(t+h) - f(t)}{h}$$

Backward Difference

$$\frac{df}{dt}(t) = \frac{f(t) - f(t-h)}{h}$$

- Collocation over finite elements



Small Example

$$\frac{dz}{dt} = z^2 - 2z + 1, \qquad z(0) = -3$$

- Exercise: Solve the differential equation using a backward finite difference scheme over the time interval $t \in [0,1]$

Backward Difference $\frac{df}{dt}(t) = \frac{f(t) - f(t-h)}{h}$

- Exercise: Plot the solution against the analytic solution

Analytic solution $z(t) = (4t - 3)/(4t + 1)$

Small Example – Exercise Solution

- Solve the differential equation using a backward finite difference scheme

```
import pyomo.environ as pyo

numpoints = 10
m = pyo.ConcreteModel()
m.points = pyo.RangeSet(0, numpoints)
m.h = pyo.Param(initialize=1.0/numpoints)

m.z = pyo.Var(m.points)
m.dzdt = pyo.Var(m.points)

m.obj = pyo.Objective(expr=1) # Dummy Objective

def _zdot(m, i):
    return m.dzdt[i] == m.z[i]**2 - 2*m.z[i] + 1
m.zdot = pyo.Constraint(m.points, rule=_zdot)

def _back_diff(m,i):
    if i == 0:
        return pyo.Constraint.Skip
    return m.dzdt[i] == (m.z[i]-m.z[i-1])/m.h
m.back_diff = pyo.Constraint(m.points, rule=_back_diff)

def _init_con(m):
    return m.z[0] == -3
m.init_con = pyo.Constraint(rule=_init_con)
```

Small Example – Exercise Solution

- Plotting

```
solver = pyo.SolverFactory('ipopt')
solver.solve(m, tee=True)

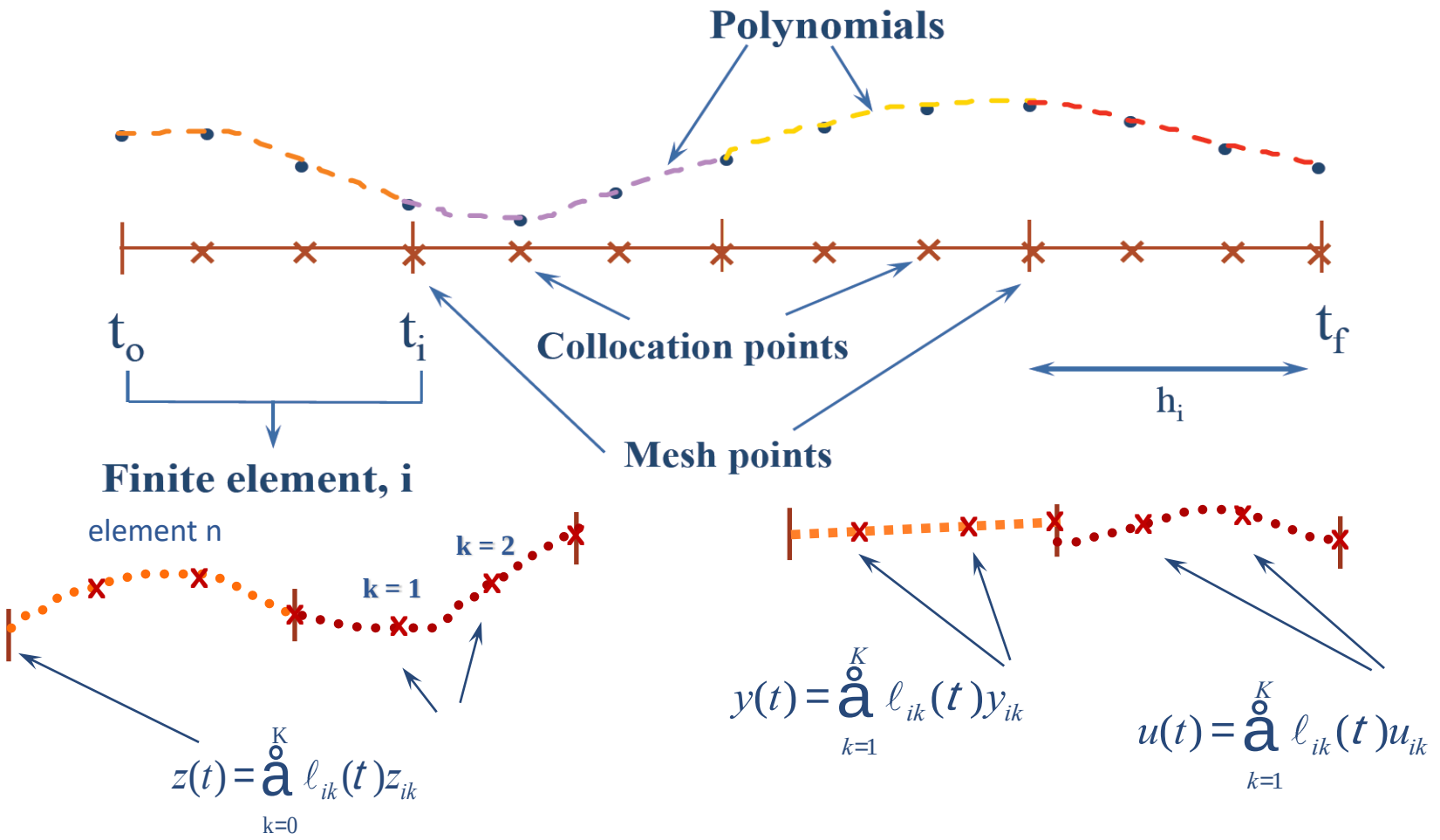
import matplotlib.pyplot as plt

analytical_t = [0.01*i for i in range(0,101)]
analytical_z = [(4*t-3)/(4*t+1) for t in analytical_t]

findiff_t = [m.h*i for i in m.points]
findiff_z = [pyo.value(m.z[i]) for i in m.points]

plt.plot(analytical_t, analytical_z, 'b', label='analytical solution')
plt.plot(findiff_t, findiff_z, 'ro–', label='finite difference solution')
plt.legend(loc='best')
plt.xlabel('t')
plt.show()
```

Collocation over finite elements



Differential variables
(Continuous)

Algebraic and
Control variables
(Discontinuous)

Collocation over finite elements

Given: $$\frac{dz}{dt} = f(z(t),t), \quad z(0) = z_0,$$

Approximate z by Lagrange interpolation polynomials (order K+1) with interpolation points, t_k

$$\left.\begin{aligned} t &= t_{i-1} + h_i \tau, \\ z^K(t) &= \sum_{j=0}^{K} \ell_j(\tau) z_{ij}, \end{aligned}\right\} t \in [t_{i-1}, t_i], \quad \tau \in [0,1], \quad \ell_j(\tau) = \prod_{k=0,\neq j}^{K} \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)},$$

Collocation Coefficients to be solved for

Evaluated at t not **τ**

Collocation Equations → $$\sum_{j=0}^{K} z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h_i f(z_{ik}, t_{ik}), \quad k = 1, \ldots, K,$$

Known Collocation Points

Evaluated at the element boundary

Continuity Equations → $$z_{i+1,0} = \sum_{j=0}^{K} \ell_j(1) z_{ij}, \quad i = 1, \ldots, N-1$$

Collocation points

Degree K	Legendre Roots	Radau Roots
1	0.500000	1.000000
2	0.211325 0.788675	0.333333 1.000000
3	0.112702 0.500000 0.887298	0.155051 0.644949 1.000000
4	0.069432 0.330009 0.669991 0.930568	0.088588 0.409467 0.787659 1.000000
5	0.046910 0.230765 0.500000 0.769235 0.953090	0.057104 0.276843 0.583590 0.860240 1.000000

Shifted Gauss-Legendre and Radau roots as collocation points

Small Example - Collocation

$$\frac{dz}{dt} = z^2 - 2z + 1, \qquad z(0) = -3$$

- Solve the differential equation using collocation with a single finite element and 3 radau collocation points over the time interval $t \in [0,1]$

$$\sum_{j=0}^{3} z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h(z_{ik}^2 - 2z_{ik} + 1), \; k = 1, \ldots, 3, \; i = 1, \ldots, N$$

$$z_{i+1,0} = \sum_{j=0}^{3} \ell_j(1) z_{ij}, \quad i = 1, \ldots, N-1$$

Using Radau collocation, we have $\tau_0 = 0$, $\tau_1 = 0.155051$, $\tau_2 = 0.644949$ and $\tau_3 = 1$.

Small Example - Collocation

$$\sum_{j=0}^{3} z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h(z_{ik}^2 - 2z_{ik} + 1),\ k = 1, \ldots, 3,\ i = 1, \ldots, N$$

$$z_{i+1,0} = \sum_{j=0}^{3} \ell_j(1) z_{ij}, \quad i = 1, \ldots, N-1$$

Using Radau collocation, we have $\tau_0 = 0$, $\tau_1 = 0.155051$, $\tau_2 = 0.644949$ and $\tau_3 = 1$.

$$\ell_0(\tau) = \frac{(\tau - \tau_1)(\tau - \tau_2)(\tau - \tau_3)}{(\tau_0 - \tau_1)(\tau_0 - \tau_2)(\tau_0 - \tau_3)} = a_3\tau^3 + a_2\tau^2 + a_1\tau + a_0$$

$$\ell_0(\tau) = -10\tau^3 + 18\tau^2 - 9\tau + 1$$

$$\dot{\ell}_0(\tau) = -30\tau^2 + 36\tau - 9$$

Other Lagrange polynomials found similarly

Small Example - Collocation

For $N = 1$, and $z_0 = -3$ the collocation equations are given by:

$$\sum_{j=0}^{3} z_j \frac{d\ell_j(\tau_k)}{d\tau} = (z_k^2 - 2z_k + 1),\ k = 1, \ldots, 3,$$

which can be written out as:

$$\begin{aligned} &z_0(-30\tau_k^2 + 36\tau_k - 9) + z_1(46.7423\tau_k^2 - 51.2592\tau_k + 10.0488) \\ &+ z_2(-26.7423\tau_k^2 + 20.5925\tau_k - 1.38214) + z_3(10\tau_k^2 - \frac{16}{3}\tau_k + \frac{1}{3}) \\ &= (z_k^2 - 2z_k + 1),\ k = 1, \ldots, 3. \end{aligned}$$

Solving these three equations gives $z_1 = -1.65701$, $z_2 = 0.032053$, $z_3 = 0.207272$

Collocation Matrix

$$\sum_{j=0}^{3} z_j \frac{d\ell_j(\tau_k)}{d\tau} = (z_k^2 - 2z_k + 1),\ k = 1, \ldots, 3,$$

Constant

which can be written out as:

$$z_0(-30\tau_k^2 + 36\tau_k - 9) + z_1(46.7423\tau_k^2 - 51.2592\tau_k + 10.0488)$$
$$+ z_2(-26.7423\tau_k^2 + 20.5925\tau_k - 1.38214) + z_3(10\tau_k^2 - \frac{16}{3}\tau_k + \frac{1}{3})$$
$$= (z_k^2 - 2z_k + 1),\ k = 1, \ldots, 3.$$

- $adot(k,k) = \begin{bmatrix} \dot{\ell}_0(\tau_0) & \cdots & \dot{\ell}_0(\tau_k) \\ \vdots & \ddots & \vdots \\ \dot{\ell}_k(\tau_0) & \cdots & \dot{\ell}_k(\tau_k) \end{bmatrix}$

Python code for generating collocation matrix

```
import numpy
# Specify collocation points
cp = [0, 0.155051, 0.644949, 1]

a = []
for i in range(len(cp)):
    ptmp = []
    tmp = 0
    for j in range(len(cp)):
        if j != i:
            row = []
            row.insert(0,1/(cp[i]-cp[j]))
            row.insert(1,-cp[j]/(cp[i]-cp[j]))
            ptmp.insert(tmp,row)
            tmp += 1
    p=[1]
    for j in range(len(cp)-1):
        p = numpy.convolve(p,ptmp[j])
    pder = numpy.polyder(p,1)
    arow = []
    for j in range(len(cp)):
        arow.append(numpy.polyval(pder,cp[j]))
    a.append(arow)
    print(arow)
```

Small Example - Collocation

$$\frac{dz}{dt} = z^2 - 2z + 1, \qquad z(0) = -3$$

- Exercise: Solve the differential equation using collocation over a single finite element with $t \in [0,1]$

$$\left.\begin{aligned} t &= t_{i-1} + h_i \tau, \\ z^K(t) &= \sum_{j=0}^{K} \ell_j(\tau) z_{ij}, \end{aligned}\right\} t \in [t_{i-1}, t_i], \quad \tau \in [0,1], \quad \ell_j(\tau) = \prod_{k=0, \neq j}^{K} \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)},$$

Collocation Equations $\rightarrow$

$$\sum_{j=0}^{K} z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h_i f(z_{ik}, t_{ik}), \quad k = 1, \ldots, K,$$

- Exercise: Plot the solution against the analytic solution

Analytic solution $z(t) = (4t - 3)/(4t + 1)$

Implementation is challenging!

- Common theme: significant effort to rework formulation
  - Time: first ~6 months of a grad student's research
  - Error prone: many ways to make subtle mistakes
  - Inflexible: formulation specific to selected solution approach

- Difficult to know apriori the best solution approach for a particular model

Expressing dynamical systems

- Model dynamical systems in a natural form
  - Systems of Differential Algebraic Equations (DAE)
  - Extend the Pyomo component model
    - `ContinuousSet`: A virtual set over which you can take a derivative
    - `DerivativeVar`: The derivative of a `Var` with respect to a `ContinuousSet`

$$
\begin{aligned}
& \min \Psi(x(t), y(t), u(t)) \\
& \dot{x}(t) = f(x(t), y(t), u(t), t, p) \\
\text{DAE} \quad & 0 = g(x(t), y(t), u(t), t, p) \\
\text{model} \quad & x(t) \in \Re^n \\
& y(t) \in \Re^n \\
& u(t) \in \Re^m
\end{aligned}
$$

Simple Example – pyomo.dae

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.t     = ContinuousSet(bounds=(0, 1))

m.z    = pyo.Var(m.t)
m.dzdt = DerivativeVar(m.z, wrt=m.t)

m.obj = pyo.Objective(expr=1) # Dummy Objective

def _zdot(m, t):
    return m.dzdt[t] == m.z[t]**2 - 2*m.z[t] + 1
m.zdot = pyo.Constraint(m.t, rule=_zdot)

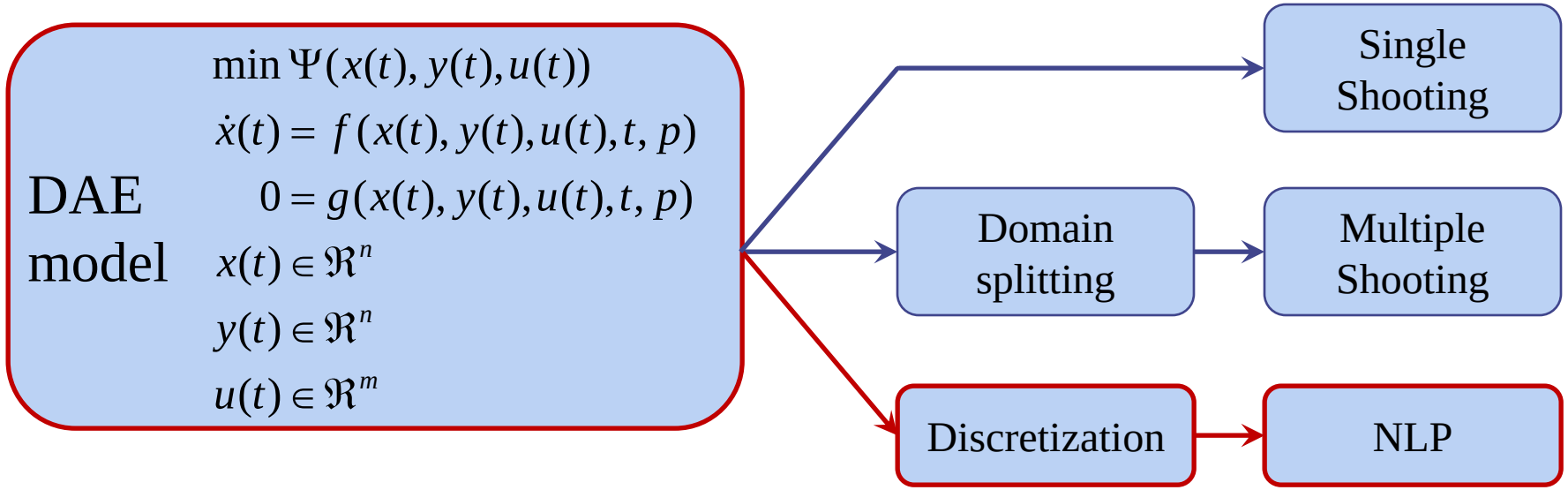
def _init_con(m):
    return m.z[0] == -3
m.init_con = pyo.Constraint(rule=_init_con)
```

$$\frac{dz}{dt} = z^2 - 2z + 1$$

$$z(0) = -3$$

Solving dynamical systems

- Given that we have a DAE model in Pyomo… now what?
  - How to optimize?
    - Simulation-based / Multiple shooting methods / Simultaneous
  - Common theme: significant effort to rework formulation
    - Time / Error prone / Inflexible
- Our approach: separate the *declaration* of dynamical models from the *solution approach* using (nearly) *automatic transformations*



Simple Example – pyomo.dae

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.t     = ContinuousSet(bounds=(0, 1))

m.z    = pyo.Var(m.t)
m.dzdt = DerivativeVar(m.z, wrt=m.t)

m.obj = pyo.Objective(expr=1) # Dummy Objective

def _zdot(m, t):
    return m.dzdt[t] == m.z[t]**2 - 2*m.z[t] + 1
m.zdot = pyo.Constraint(m.t, rule=_zdot)

def _init_con(m):
    return m.z[0] == -3
m.init_con = pyo.Constraint(rule=_init_con)

# Discretize model using backward finite difference
discretizer = pyo.TransformationFactory('dae.finite_difference')
discretizer.apply_to(m, nfe=10, scheme='BACKWARD')

# Discretize model using radau collocation
discretizer = pyo.TransformationFactory('dae.collocation')
discretizer.apply_to(m, nfe=1, ncp=3, scheme='LAGRANGE-RADAU')
```

Small example: optimal control (1/8)

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

[Jacobson and Lele (1969)]

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.tf    = pyo.Param(initialize = 1)
m.t     = ContinuousSet(bounds=(0, m.tf))

m.u     = pyo.Var(m.t, initialize=0)
m.x1    = pyo.Var(m.t)
m.x2    = pyo.Var(m.t)
m.x3    = pyo.Var(m.t)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

m.obj = pyo.Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
m.x1dot = pyo.Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = pyo.Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)

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

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

Small example: optimal control (2/8)

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

[Jacobson and Lele (1969)]

```python
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.tf    = pyo.Param(initialize = 1)
m.t     = ContinuousSet(bounds=(0, m.tf))

m.u     = pyo.Var(m.t, initialize=0)
m.x1    = pyo.Var(m.t)
m.x2    = pyo.Var(m.t)
m.x3    = pyo.Var(m.t)
m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

m.obj = pyo.Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
m.x1dot = pyo.Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = pyo.Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)

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

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


```python
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar
```

Small example: optimal control (3/8)

$$
\begin{aligned}
\min \quad & \\
s.t \quad & \\
\dot{x}_2 &= u \\
\dot{x}_3 &= x_1^2 + x_2^2 + 0.005 * u^2 \\
x_2 - 8 * (t - 0.5)^2 + 0.5 &\leq 0 \\
x_1(0) &= 0 \\
x_2(0) &= -1 \\
x_3(0) &= 0 \\
t_f &= 1
\end{aligned}
$$

[Jacobson and Lele (1969)]

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.tf    = pyo.Param(initialize = 1)
m.t     = ContinuousSet(bounds=(0, m.tf))

m.u     = pyo.Var(m.t, initialize=0)
m.x1    = pyo.Var(m.t)

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
m.x1dot = pyo.Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = pyo.Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)

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

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

Small example: optimal control (4/8)

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

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

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

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

$$x_2 - 8 * (t -$$

$$x_1($$

$$x_2(0$$

$$x_3($$

$$t_f$$

[Jacobson a

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.tf    = pyo.Param(initialize = 1)
m.t     = ContinuousSet(bounds=(0, m.tf))

m.u     = pyo.Var(m.t, initialize=0)
m.x1    = pyo.Var(m.t)
m.x2    = pyo.Var(m.t)
m.x3    = pyo.Var(m.t)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

m.obj = pyo.Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
```


```
m.u     = pyo.Var(m.t, initialize=0)
m.x1    = pyo.Var(m.t)
m.x2    = pyo.Var(m.t)
m.x3    = pyo.Var(m.t)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)
```


```
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = pyo.ConstraintList(rule=_init)
```

Small example: optimal control (5/8)

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

$$\begin{aligned} s.t. \quad \dot{x_1} &= x_2 \\ \dot{x_2} &= -x_2 + u \\ \dot{x_3} &= x_1^2 + x_2^2 + 0.005 * u^2 \\ x_2 - 8 * (t - 0.5)^2 &\\ x_1(0) &= 0 \\ x_2(0) &= -1 \\ x_3(0) &= 0 \\ t_f &= 1 \end{aligned}$$

[Jacobson and Lele (1969)]

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.tf    = pyo.Param(initialize = 1)
m.t     = ContinuousSet(bounds=(0, m.tf))

m.u     = pyo.Var(m.t, initialize=0)
m.x1    = pyo.Var(m.t)
m.x2    = pyo.Var(m.t)
m.x3    = pyo.Var(m.t)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

m.obj = pyo.Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
```


```
m.obj = pyo.Objective(expr=m.x3[m.tf])
```


```
def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)

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

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

Small example: optimal control (6/8)

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

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

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

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

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

$$x_1(0) = 0$$

$$x_2(0) = -1$$

$$x_3(0) = 0$$

$$t_f = 1$$

[Jacobson and Lele (1969)]

```
import pyomo.environ as pyo
```

```
def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)
```

```
m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

m.obj = pyo.Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
m.x1dot = pyo.Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = pyo.Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)

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

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

Small example: optimal control (7/8)

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

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

[Jacobson and Lele (1969)]

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()
m.t = ContinuousSet(bounds=(0, 1))
m.x1 = pyo.Var(m.t)
m.x2 = pyo.Var(m.t)
m.x3 = pyo.Var(m.t)
m.u = pyo.Var(m.t)
m.tf = pyo.Param(initialize=1)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

m.obj = pyo.Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
m.x1dot = pyo.Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = pyo.Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)

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

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

Small example: optimal control (8/8)

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

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

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

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

$$x_1(0) = 0$$
$$x_2(0) = -1$$
$$x_3(0) = 0$$

$$t_f = 1$$

[Jacobson and Lele (1969)]

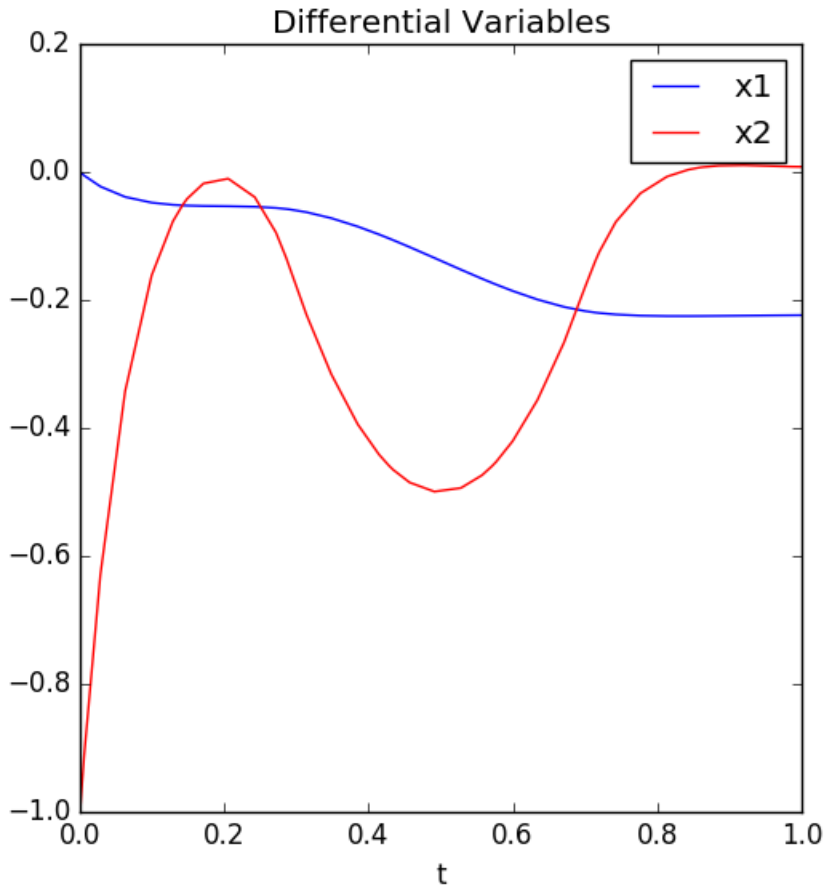
```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

model = m = pyo.ConcreteModel()

m.obj = pyo.Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
    return m.dx1dt[t] == m.x2[t]
m.x1dot = pyo.Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
    return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = pyo.Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
    return m.dx3dt[t] == m.x1[t]**2 + \
        m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = pyo.Constraint(m.t, rule=_x3dot)

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

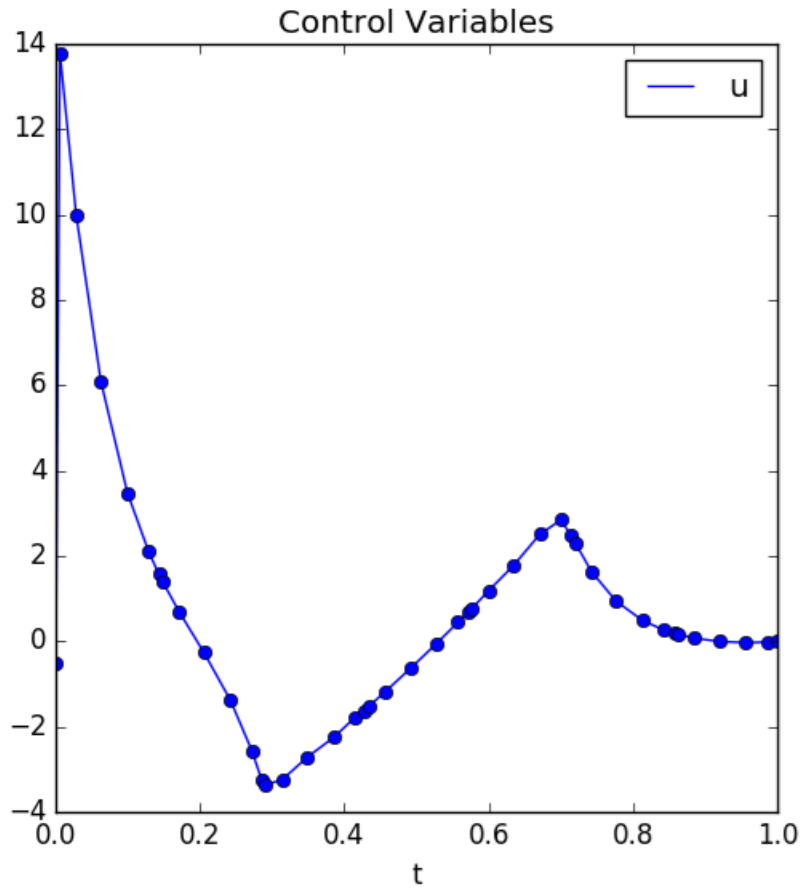

```
def _init(m):
    yield m.x1[0] == 0
    yield m.x2[0] == -1
    yield m.x3[0] == 0
m.init_conditions = pyo.ConstraintList(rule=_init)
```

Optimal Control Example - Script

```
import pyomo.environ as pyo
# Import dynamic model
from optimalControl import m

# Discretize model using radau collocation
pyo.TransformationFactory('dae.collocation').apply_to(m, nfe=7, ncp=6,
                                                      scheme='LAGRANGE-RADAU' )

# Solve algebraic model
results = pyo.SolverFactory('ipopt').solve(m)

def plotter(subplot, x, *series, **kwds):
    plt.subplot(subplot)
    for i,y in enumerate(series):
        plt.plot(x, [pyo.value(y[t]) for t in x], 'brgcmk'[i%6]+kwds.get('points','')
    plt.title(kwds.get('title',''))
    plt.legend(tuple(y.cname() for y in series))
    plt.xlabel(x.cname())

import matplotlib.pyplot as plt
plotter(121, m.t, m.x1, m.x2, title='Differential Variables')
plotter(122, m.t, m.u, title='Control Variable', points='o')
plt.show()
```

Optimal Control Example - Results

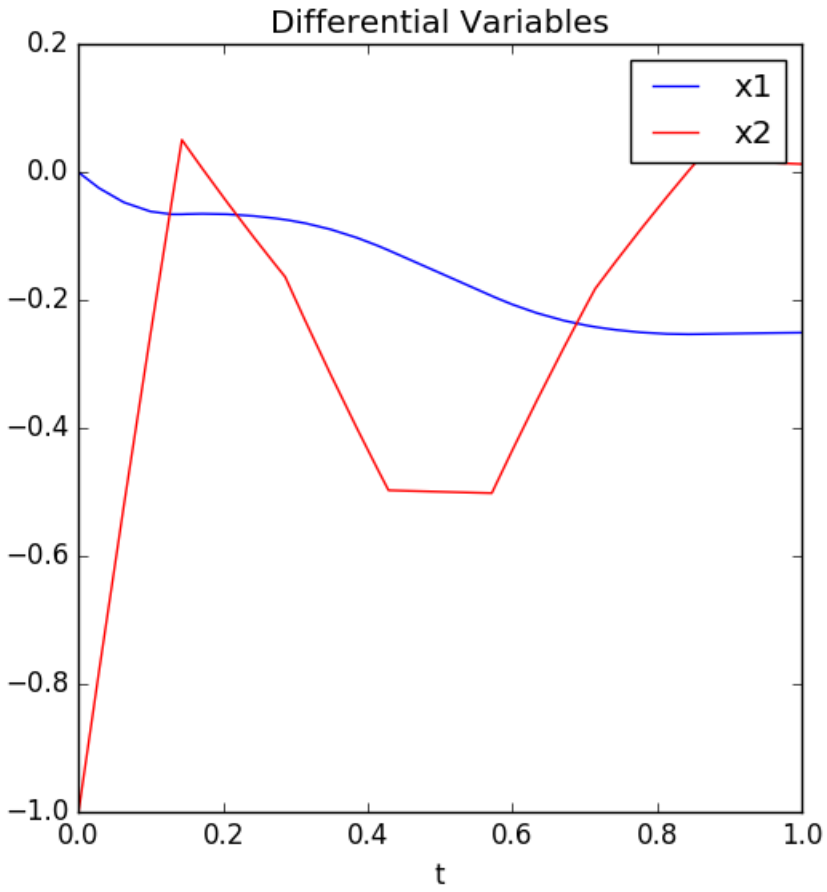


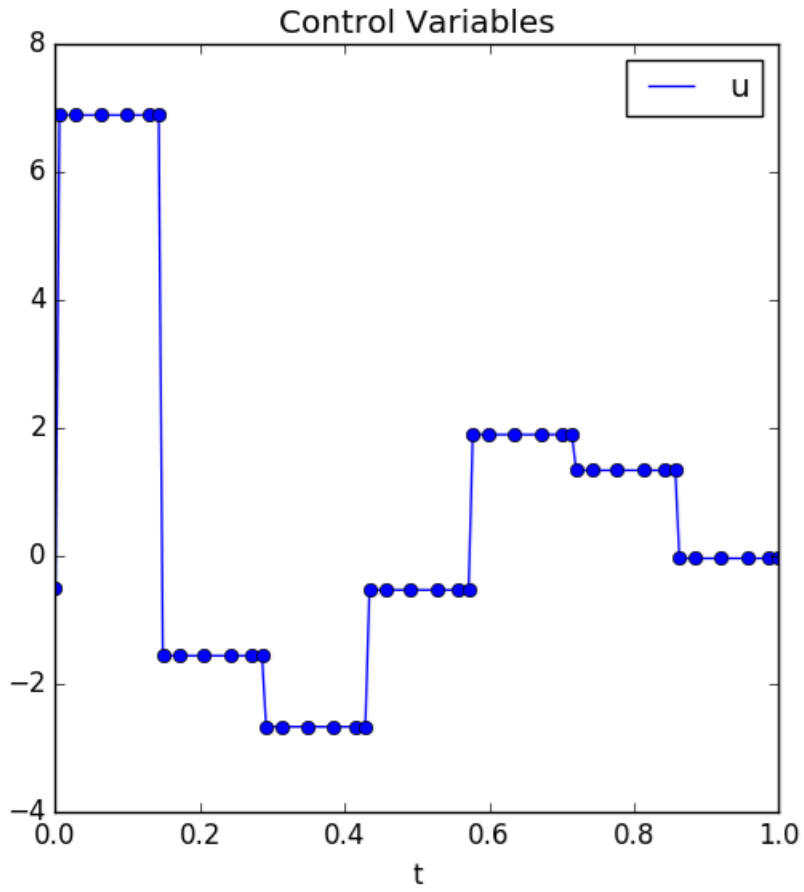


Optimal Control Example - Script

```
import pyomo.environ as pyo
# Import dynamic model
from optimalControl import m

# Discretize model using radau collocation
discretizer = pyo.TransformationFactory('dae.collocation')
discretizer.apply_to( m, nfe=7, ncp=6, scheme='LAGRANGE-RADAU' )

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

# Solve algebraic model
results = pyo.SolverFactory('ipopt').solve(m)

def plotter(subplot, x, *series, **kwds):
    plt.subplot(subplot)
    for i,y in enumerate(series):
        plt.plot(x, [pyo.value(y[t]) for t in x], 'brgcmk'[i%6]+kwds.get('points','')
    plt.title(kwds.get('title',''))
    plt.legend(tuple(y.cname() for y in series))
    plt.xlabel(x.cname())

import matplotlib.pyplot as plt
plotter(121, m.t, m.x1, m.x2, title='Differential Variables')
plotter(122, m.t, m.u, title='Control Variable', points='o')
plt.show()
```

Optimal Control Example - Results 2





Parameter Estimation Example 1

$$\min \sum_{t_i} \left(x_1(t_i) - x_{1_{meas}}(t_i) \right)^2$$

$$s.t. \quad \frac{dx_1}{dt} = x_2$$

$$\frac{dx_2}{dt} = 1 - 2x_2 - x_1$$

$$-1.5 \leq p_1, p_2 \leq 1.5$$

$$x_1(0) = p_1, \qquad x_2(0) = p_2$$

Want to ensure that these time points are included as discretization points

Time	1	2	3	5
$x_{1_{meas}}$	0.264	0.594	0.801	0.959

Initialize with these points:

```
measT = [1, 2, 3, 5]
m = pyo.ConcreteModel()
m.t = ContinuousSet(bounds=(0, 6),
                    initialize=measT)
```

Parameter Estimation Example 1

$$\min \sum_{t_i} \left(x_1(t_i) - x_{1_{meas}}(t_i) \right)^2$$

$$s.t. \quad \frac{dx_1}{dt} = x_2$$

$$\frac{dx_2}{dt} = 1 - 2x_2 - x_1$$

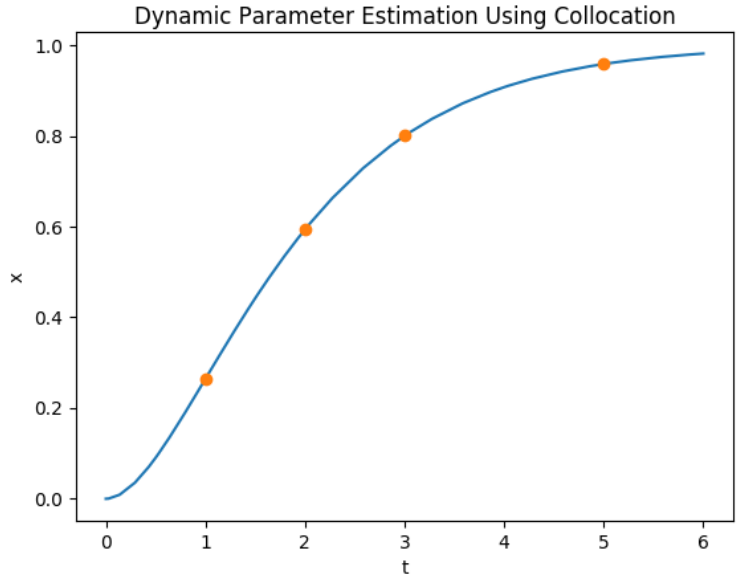
$$-1.5 \leq p_1, p_2 \leq 1.5$$

$$x_1(0) = p_1, \qquad x_2(0) = p_2$$

```
measurements = {1:0.264, 2:0.594, 3:0.801, 5:0.959}

model = pyo.ConcreteModel()
model.t = ContinuousSet(initialize=measurements.keys(), bounds=(0, 6))

model.x1 = pyo.Var(model.t)
model.x2 = pyo.Var(model.t)
model.p1 = pyo.Var(bounds=(-1.5,1.5))
model.p2 = pyo.Var(bounds=(-1.5,1.5))

model.x1dot = DerivativeVar(model.x1, wrt=model.t)
model.x2dot = DerivativeVar(model.x2)

def _init_conditions(model):
      yield model.x1[0] == model.p1
      yield model.x2[0] == model.p2
model.init_conditions = pyo.ConstraintList(rule=_init_conditions)

def _x1dot(model,i):
      return model.x1dot[i] == model.x2[i]
model.x1dotcon = pyo.Constraint(model.t, rule=_x1dot)

def _x2dot(model,i):
      return model.x2dot[i] == 1-2*model.x2[i]-model.x1[i]
model.x2dotcon = pyo.Constraint(model.t, rule=_x2dot)

def _obj(model):
      return sum((model.x1[i]-measurements[i])**2 for i in measurements.keys())
model.obj = pyo.Objective(rule=_obj)

# Discretize model using Orthogonal Collocation
discretizer = pyo.TransformationFactory('dae.collocation')
discretizer.apply_to(model, nfe=8, ncp=5)

results = pyo.SolverFactory('ipopt’).solve(model, tee=True)
```

Parameter Estimation Example 1

```
t_meas = sorted(list(measurements.keys()))
x1_meas = [pyo.value(measurements[i]) for i in sorted(measurements.keys())]

t = list(model.t)
x1 = [pyo.value(model.x1[i]) for i in model.t]

import matplotlib.pyplot as plt

plt.plot(t,x1)
plt.plot(t_meas,x1_meas,'o')
plt.xlabel('t')
plt.ylabel('x')
plt.title('Dynamic Parameter Estimation Using Collocation')
plt.show()
```



Parameter Estimation Example 2

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C$$

$$\frac{dA}{dt} = -k_1 A$$

$$\frac{dB}{dt} = k_1 A - k_2 B$$

$$A(0) = 1, \qquad B(0) = 0$$

Time	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
A	0.606	0.368	0.223	0.135	0.082	0.050	0.030	0.018	0.011	0.007
B	0.373	0.564	0.647	0.669	0.656	0.624	0.583	0.539	0.494	0.451

- Exercise: Solve for k_1 and k_2 given the concentration measurements in the table. Start with `start_param_est.py`

Solution to this problem is k1 = 5, k2 = 1

Disease Transmission Example

- Parameter estimation of a S-I-R disease transmission model[*]

$$
\begin{aligned}
\min \quad & \omega_M \sum_{i\in\mathcal{F}} (\ln(\varepsilon_{M_i}))^2 + \omega_Q \sum_{k\in\mathcal{T}} (\varepsilon_{Q_k})^2 \\
\text{s.t.} \quad & \frac{\mathrm{d}S}{\mathrm{d}t} = \frac{-\beta(y(t))S(t)I(t)}{N(t)} \cdot \varepsilon_M(t) + B(t), \\
& \frac{\mathrm{d}I}{\mathrm{d}t} = \frac{\beta(y(t))S(t)I(t)}{N(t)} \cdot \varepsilon_M(t) - \gamma I(t), \\
& \frac{\mathrm{d}Q}{\mathrm{d}t} = \frac{\beta(y(t))S(t)I(t)}{N(t)} \cdot \varepsilon_M(t), \\
& R_k^\star = \eta_k(Q_{i,k} - Q_{i,k-1}) + \varepsilon_{Q_k}, \\
& \bar{S} = \frac{\sum_{i\in\mathcal{F}} S_i}{\mathrm{len}(\mathcal{F})}, \\
& \bar{\beta} = \frac{\sum_{i\in\tau} \beta_i}{\mathrm{len}(\tau)}, \\
& 0 \le I(t), S(t) \le N(t) \\
\text{and} \quad & 0 \le \beta(y(t)), Q(t),
\end{aligned}
$$

Discretized problem:

- 3 differential equations
- 520 finite elements
- 3 collocation points

~ 10,500 variables

~ 10,000 constraints

[*] Word, Cummings, Burke, Iamsirithaworn, and Laird, *Journal of the Royal Society Interface*, 2012, pp. 1983-1997.

Performance impact

- Comparing the automated discretization to a manually discretized model implemented (and tuned) by a person

	Manual Discretization	Using pyomo.dae (Radau Collocation)
Model Creation Time (CPU secs)	1.79	5.29
Solve Time (CPU secs)	1.35	0.86
IPOPT Iterations	27	26
Objective ($x10^{-5}$)	1.4716	1.4716

- The bulk of the added model creation time is the model transformation that implements the discretization

Performance impact

- Comparing the automated discretization to a manually discretized model implemented (and tuned) by a person

	Manual Discretization	Using pyomo.dae (Radau Collocation)	Pyomo 6.7.0 Python 3.11
Model Creation Time (CPU secs)	1.79	5.29	0.89
Solve Time (CPU secs)	1.35	0.86	0.582
IPOPT Iterations	27	26	24
Objective ($x10^{-5}$)	1.4716	1.4716	1.4716

- The bulk of the added model creation time is the model transformation that implements the discretization

Distillation Example

```
model.S_TRAYS = pyo.Set()
model.S_RECTIFICATION = pyo.Set(within = model.S_TRAYS)
model.S_STRIPPING = pyo.Set(within = model.S_TRAYS)

model.t = ContinuousSet(initialize=range(1,52))
model.x = pyo.Var(model.S_TRAYS, model.t, initialize=x_init_rule)
model.dx = DerivativeVar(model.x)



def _diffeq(m,n,t):
    if n == 1:
        return m.dx[n,t] == 1/m.acond*m.V[t]*(m.y[n+1,t]-m.x[n,t])
    elif n in m.S_RECTIFICATION:
        return m.dx[n,t] == 1/m.atray*(m.L[t]*(m.x[n-1,t]-m.x[n,t])-m.V[t]*(m.y[n,t]-m.y[n+1,t]))
    elif n == 17:
        return m.dx[n,t] == 1/m.atray*(m.Feed*m.x_Feed+m.L[t]*m.x[n-1,t]-m.FL[t]*m.x[n,t]- \
     m.V[t]*(m.y[n,t]-m.y[n+1,t]))
    elif n in m.S_STRIPPING:
        return m.dx[n,t] == 1/m.atray*(m.FL[t]*(m.x[n-1,t]-m.x[n,t])-m.V[t]*(m.y[n,t]-m.y[n+1,t]))
    else :
        return m.dx[n,t] == 1/m.areb*(m.FL[t]*m.x[n-1,t]-(m.Feed-m.D)*m.x[n,t]-m.V[t]*m.y[n,t])
model.diffeq = pyo.Constraint(model.S_TRAYS, model.t, rule=_diffeq)
```

PDE Example

- Illustrative example[1]
  - PDE

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

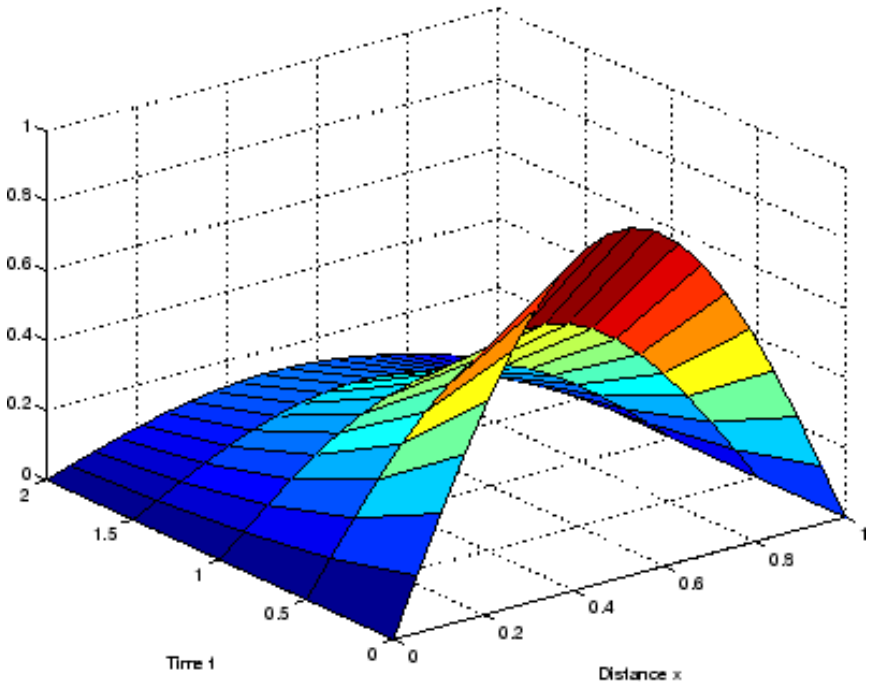
  - Initial Condition

$$u(x,0) = \sin(\pi x)$$

  - Boundary Conditions

$$u(0,t) = 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0$$



[1] Example 1 from http://www.mathworks.com/help/matlab/ref/pdepe.html

PDE Example

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

m = pyo.ConcreteModel()
m.pi = pyo.Param(initialize=3.1416)
m.t = ContinuousSet(bounds=(0, 2))
m.x = ContinuousSet(bounds=(0, 1))
m.u = pyo.Var(m.x, m.t)

m.dudx = DerivativeVar(m.u, wrt=m.x)
m.dudx2 = DerivativeVar(m.u, wrt=(m.x, m.x))
m.dudt = DerivativeVar(m.u, wrt=m.t)
```

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \qquad u(x,0) = \sin(\pi x) \qquad u(0,t) = 0 \qquad \pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0$$

PDE Example

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

```
def _pde(m, i, j):
    if i == 0 or j == 0:
        return pyo.Constraint.Skip
    return m.pi**2 * m.dudt[i, j] == m.dudx2[i, j]
m.pde = pyo.Constraint(m.x, m.t, rule=_pde)
```

$$u(x, 0) = \sin(\pi x)$$

```
def _initcon(m, i):
    if i == 0 or i == 1:
        return pyo.Constraint.Skip
    return m.u[i, 0] == pyo.sin(m.pi * i)
m.initcon = pyo.Constraint(m.x, rule=_initcon)
```

$$u(0, t) = 0$$

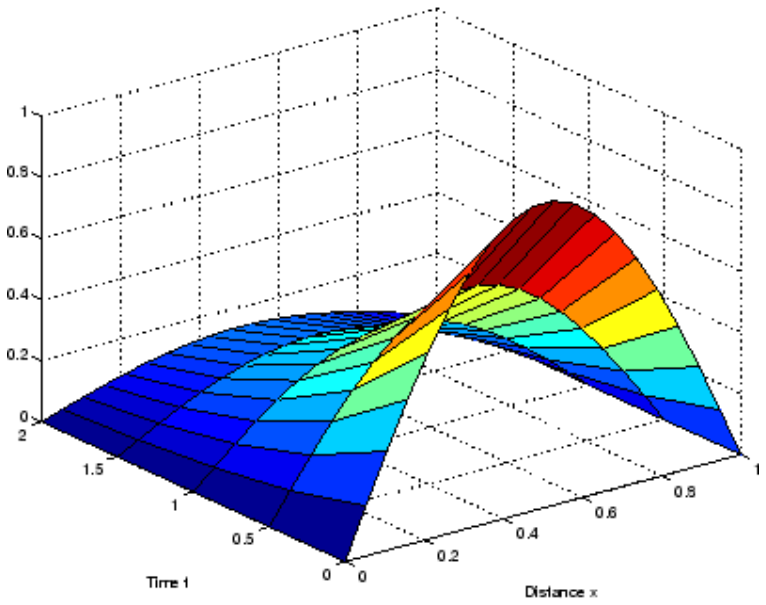
```
def _lowerbound(m, j):
    return m.u[0, j] == 0
m.lowerbound = pyo.Constraint(m.t, rule=_lowerbound)
```

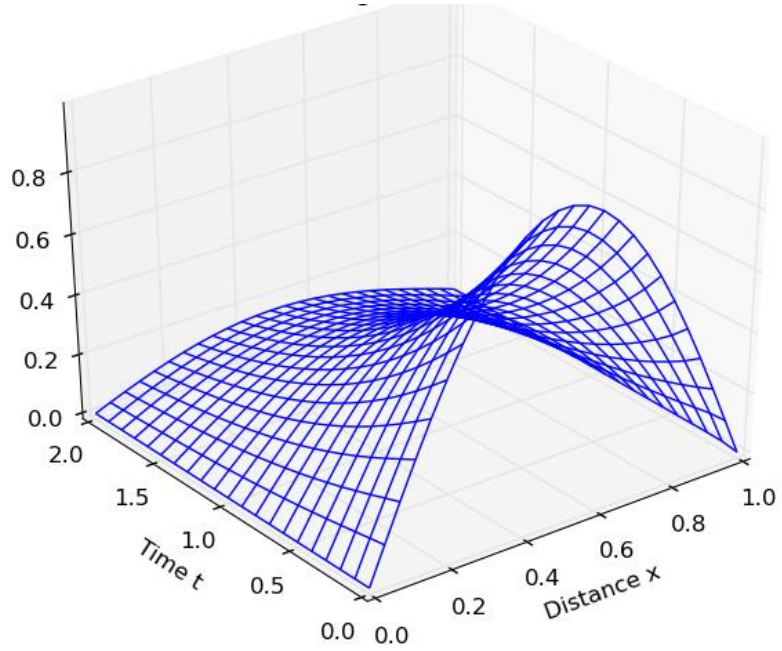
$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$

```
def _upperbound(m, j):
    return m.pi * pyo.exp(-j) + m.dudx[1, j] == 0
m.upperbound = pyo.Constraint(m.t, rule=_upperbound)
```

PDE Example

```
# Discretize using Finite Difference Method
discretizer = pyo.TransformationFactory('dae.finite_difference')
discretizer.apply_to(m, nfe=25, wrt=m.x, scheme='BACKWARD')
discretizer.apply_to(m, nfe=20, wrt=m.t, scheme='BACKWARD')

solver = pyo.SolverFactory('ipopt')
solver.solve(m, tee=True)
```





Interfacing with ODE/DAE integrators

$$\frac{d\theta}{dt} = \omega$$

$$\frac{d\omega}{dt} = -b * \omega - c * \sin\theta$$

$$\theta(0) = \pi - 0.1$$

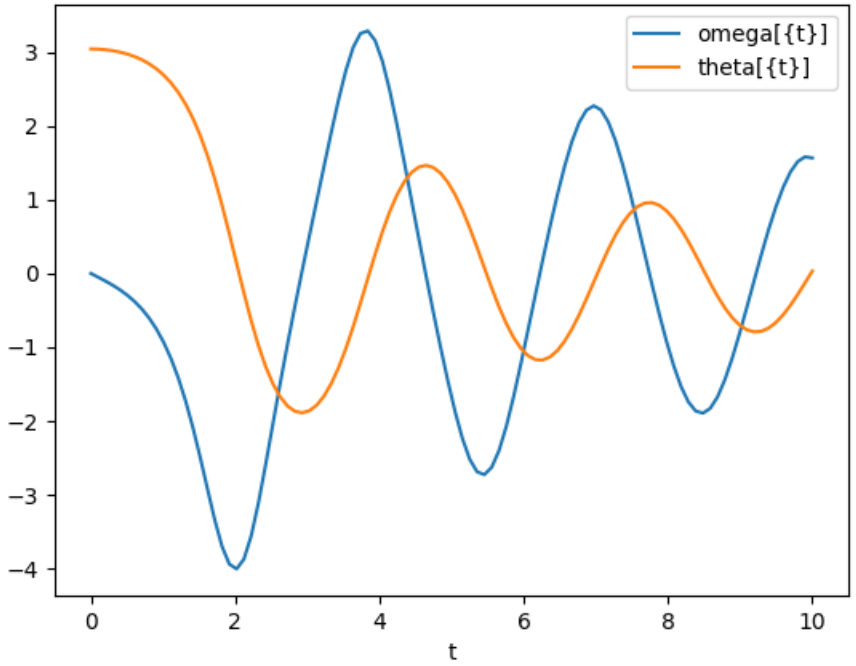
$$\omega(0) = 0$$

```
import pyomo.environ as pyo
from pyomo.dae import ContinuousSet, DerivativeVar

m = model = pyo.ConcreteModel()

m.t = ContinuousSet(bounds=(0.0, 10.0))
m.b = pyo.Param(initialize=0.25)
m.c = pyo.Param(initialize=5.0)

m.theta = pyo.Var(m.t)
m.omega = pyo.Var(m.t)
m.dthetadt = DerivativeVar(m.theta, wrt=m.t)
m.domegadt = DerivativeVar(m.omega, wrt=m.t)

m.theta[0].fix(3.14 - 0.1)
m.omega[0].fix(0.0)

def _diffeq1(m,t):
    return m.dthetadt[t] == m.omega[t]
m.diffeq1 = pyo.Constraint(m.t, rule=_diffeq1)

def _diffeq2(m,t):
    return m.domegadt[t] == -m.b*m.omega[t] \
          - m.c*pyo.sin(m.theta[t])
m.diffeq2 = pyo.Constraint(m.t, rule=_diffeq2)
```

Interface with ODE/DAE integrators

- ODE model simulation

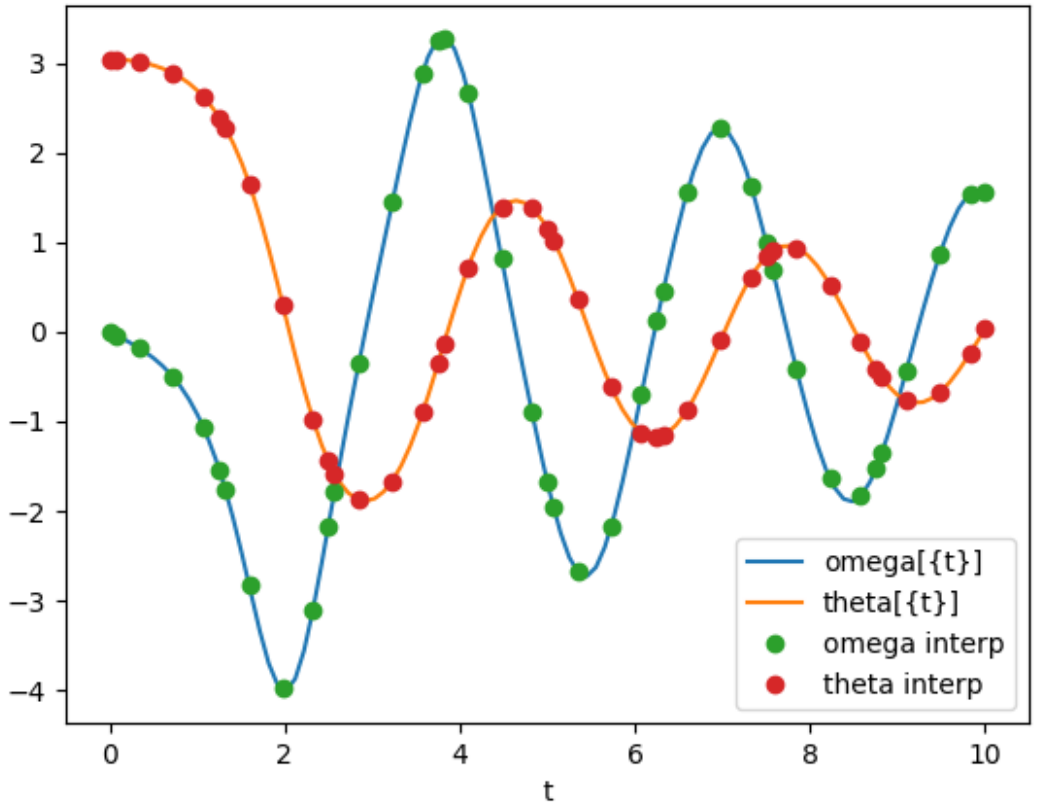
```
from pyomo.dae import Simulator
mysim = Simulator(model, package='scipy')
tsim, profiles = mysim.simulate(integrator='vode', numpoints=100)
varorder = mysim.get_variable_order()
for idx, v in enumerate(varorder):
    plt.plot(tsim, profiles[:, idx], label=v)
```



Interface with ODE/DAE integrators

- Model initialization

```
discretizer = pyo.TransformationFactory('dae.collocation')
discretizer.apply_to(model, nfe=8, ncp=5)

mysim.initialize_model()
```



Use the same Pyomo model for simulation and optimization!

Simulator Exercise

- Simulate the following kinetic model using Scipy and plot the resulting profiles

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C$$

$$\frac{dA}{dt} = -k_1 A$$

$$\frac{dB}{dt} = k_1 A - k_2 B$$

$$A(0) = 1, B(0) = 0,$$

$$k_1 = 5, k_2 = 1$$

- Reminder:

```
mysim = Simulator(model, package='scipy')
tsim, profiles = mysim.simulate(integrator='vode', numpoints=100)
varorder = mysim.get_variable_order()
for idx, v in enumerate(varorder):
    plt.plot(tsim, profiles[:, idx], label=v)
```

Farm Layout Formulation

$$
\begin{aligned}
\min \quad & 2(L+W) && \\
\text{s.t.} \quad & L \geq x_i + \ell_i && \forall i \in N \\
& W \geq y_i + w_i && \forall i \in N \\
& \frac{A_i}{w_i} - \ell_i \leq 0 && \forall i \in N \\
& \begin{bmatrix} Z_{ij}^1 \\ x_i + \ell_i \leq x_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^2 \\ x_j + \ell_j \leq x_i \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^3 \\ y_i + w_i \leq y_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^4 \\ y_j + w_j \leq y_i \end{bmatrix} && \forall i, j \in N,\ i < j \\
& \text{exactly\_one}(Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4) && \forall i, j \in N,\ i < j \\
& 0 \leq x_i \leq X && \forall i \in N \\
& 0 \leq y_i \leq Y && \forall i \in N \\
& 0 \leq w_i \leq U && \forall i \in N \\
& 0 \leq \ell_i \leq V && \forall i \in N \\
& Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4 \in \{\text{True, False}\} && \forall i, j \in N,\ i < j
\end{aligned}
$$

Generalized Disjunctive Programs

- Disjunctions: selectively enforce sets of constraints
- Examples:
  - Sequencing decisions: x ends before y or y ends before x
  - Switching decisions: a process unit is built or not, choosing operational states over time
  - Alternative selection: selecting from a set of pricing policies

$$
\begin{aligned}
\min \quad & f(x) \\
\text{s.t.} \quad & \bigvee_{j \in D_i} \begin{bmatrix} Y_{ij} \\ h_{ijk}(x) \leq 0, \ \forall k \in K_{ij} \end{bmatrix} && \forall i \in I \\
& g_\ell(x) \leq 0 && \forall \ell \in \mathcal{L} \\
& \Omega(Y) = \text{True} \\
& x \in \mathbb{R}^n \\
& Y_{ij} \in \{\text{True}, \text{False}\} && \forall i \in I, \ j \in D_i
\end{aligned}
$$

Generalized Disjunctive Programs

- Disjunctions: selectively enforce sets of constraints
- Examples:
  - Sequencing decisions: x ends before y or y ends before x
  - Switching decisions: a process unit is built or not, choosing operational states over time
  - Alternative selection: selecting from a set of pricing policies

$$\begin{aligned}
\min \quad & f(x) \\
\text{s.t.} \quad & \bigvee_{j \in D_i} \begin{bmatrix} Y_{ij} \\ h_{ijk}(x) \le 0, \ \forall k \in K_{ij} \end{bmatrix} && \forall i \in I \\
& g_\ell(x) \le 0 && \forall \ell \in \mathcal{L} \\
& \Omega(Y) = \text{True} \\
& x \in \mathbb{R}^n \\
& Y_{ij} \in \{\text{True}, \text{False}\} && \forall i \in I, \ j \in D_i
\end{aligned}$$

- Disjunctions enforce a logical relationship between sets of constraints
- Additional logical constraints on the indicator variables*

- Boolean "indicator variable"
- Constraints enforced when indicator variable is True
- Global constraints

* Almost always includes the constraints: $\text{exactly\_one}(Y_{ij}: j \in D_i), \forall i \in I$. We allow for $\text{at\_least\_one}(Y_{ij}: j \in D_i), \forall i \in I$ as well, but have never used it.

Implementing GDPs in Pyomo

- Modeling components implemented in pyomo.gdp
  - `Disjunct`:
    - Block of Pyomo components
    - Implicit Boolean "`indicator_var`" determines if block is enforced
    - Implicit binary "`binary_indicator_var`" is linked to Boolean `indicator_var`
  - `Disjunction`:
    - Enforces logical OR / `exactly_one` across a set of Disjunct indicator variables
  - (Logical constraints on indicator variables can be expressed using `LogicalConstraint` and `BooleanVar` from pyomo.core)

- Import with
  - `from pyomo.gdp import Disjunct, Disjunction`

Example: Farm Layout (Global Constraints)

```
m = pyo.ConcreteModel(name="Farm Layout model")
m.plots = pyo.RangeSet(len(areas))
m.plot_pairs = pyo.Set(initialize=m.plots * m.plots, filter=lambda _, p1, p2: p1 < p2)

m.plot_area = pyo.Param(m.plots, initialize=areas)

m.TotalLength = pyo.Var(domain=pyo.NonNegativeReals)
m.TotalWidth = pyo.Var(domain=pyo.NonNegativeReals)
m.plot_x = pyo.Var(m.plots, bounds=(0, length_ub))
m.plot_y = pyo.Var(m.plots, bounds=(0, width_ub))
m.plot_length = pyo.Var(m.plots, bounds=(0, length_ub))
m.plot_width = pyo.Var(m.plots, bounds=(0, width_ub))

m.perim = pyo.Objective(expr=2 * (m.TotalLength + m.TotalWidth))

@m.Constraint(m.plots, doc="Length is consistent with lengths of plots")
def length_consistency(m, p):
    return m.TotalLength >= m.plot_x[p] + m.plot_length[p]

@m.Constraint(m.plots, doc="Width is consistent with widths of plots")
def width_consistency(m, p):
    return m.TotalWidth >= m.plot_y[p] + m.plot_width[p]

@m.Constraint(m.plots, doc="Lengths and widths are consistent with area")
def area_consistency(m, p):
    return m.plot_area[p] / m.plot_width[p] - m.plot_length[p] <= 0
```

$$
\begin{aligned}
\min \quad & 2(L+W) && \\
\text{s.t.} \quad & L \geq x_i + \ell_i && \forall i \in N \\
& W \geq y_i + w_i && \forall i \in N \\
& \frac{A_i}{w_i} - \ell_i \leq 0 && \forall i \in N \\
& \begin{bmatrix} Z_{ij}^1 \\ x_i + \ell_i \leq x_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^2 \\ x_j + \ell_j \leq x_i \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^3 \\ y_i + w_i \leq y_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^4 \\ y_j + w_j \leq y_i \end{bmatrix} && \forall i, j \in N,\ i < j \\
& \text{exactly\_one}(Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4) && \forall i, j \in N,\ i < j \\
& 0 \leq x_i \leq X && \forall i \in N \\
& 0 \leq y_i \leq Y && \forall i \in N \\
& 0 \leq w_i \leq U && \forall i \in N \\
& 0 \leq \ell_i \leq V && \forall i \in N \\
& Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4 \in \{\text{True, False}\} && \forall i, j \in N,\ i < j
\end{aligned}
$$

Example: Farm Layout (Disjunctions)

```
@m.Disjunct(m.plot_pairs)
def p1_left_of_p2(disjunct, p1, p2):
    m = disjunct.model()
    disjunct.c = Constraint(
        expr=m.plot_x[p1] + m.plot_length[p1] <= m.plot_x[p2])
@m.Disjunct(m.plot_pairs)
def p1_right_of_p2(disjunct, p1, p2):
    m = disjunct.model()
    disjunct.c = Constraint(
        expr=m.plot_x[p2] + m.plot_length[p2] <= m.plot_x[p1])
@m.Disjunct(m.plot_pairs)
def p1_below_p2(disjunct, p1, p2):
    m = disjunct.model()
    disjunct.c = Constraint(
        expr=m.plot_y[p1] + m.plot_width[p1] <= m.plot_y[p2])
@m.Disjunct(m.plot_pairs)
def p1_above_p2(disjunct, p1, p2):
    m = disjunct.model()
    disjunct.c = Constraint(
        expr=m.plot_y[p2] + m.plot_width[p2] <= m.plot_y[p1])
```

```
@m.Disjunction(m.plot_pairs)
def no_overlap(m, p1, p2):
    return [m.p1_left_of_p2[p1, p2], m.p1_right_of_p2[p1, p2],
            m.p1_below_p2[p1, p2], m.p1_above_p2[p1, p2],
            ]
```

$$
\begin{aligned}
\min \quad & 2(L+W) && \\
\text{s.t.} \quad & L \geq x_i + \ell_i && \forall i \in N \\
& W \geq y_i + w_i && \forall i \in N \\
& \frac{A_i}{w_i} - \ell_i \leq 0 && \forall i \in N \\
& \begin{bmatrix} Z_{ij}^1 \\ x_i + \ell_i \leq x_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^2 \\ x_j + \ell_j \leq x_i \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^3 \\ y_i + w_i \leq y_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^4 \\ y_j + w_j \leq y_i \end{bmatrix} && \forall i, j \in N,\ i < j \\
& \text{exactly\_one}(Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4) && \forall i, j \in N,\ i < j \\
& 0 \leq x_i \leq X && \forall i \in N \\
& 0 \leq y_i \leq Y && \forall i \in N \\
& 0 \leq w_i \leq U && \forall i \in N \\
& 0 \leq \ell_i \leq V && \forall i \in N \\
& Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4 \in \{\text{True, False}\} && \forall i, j \in N,\ i < j
\end{aligned}
$$

Example: Farm Layout (Disjunctions—Compact Syntax)

- The most common Disjunctions contain *only* constraints
  - Disjunction rules can return
    - a list of Disjuncts,
    - a list of relational expressions, or
    - a list of lists of relational expressions
  - If necessary, the Disjunction will *implicitly* create the necessary Disjunct objects

```
@m.Disjunction(
    m.plot_pairs,
    doc="Make sure that none of the plots overlap in "
    "either the x or y dimensions.",
)
def no_overlap(m, p1, p2):
    return [
        m.plot_x[p1] + m.plot_length[p1] <= m.plot_x[p2],
        m.plot_x[p2] + m.plot_length[p2] <= m.plot_x[p1],
        m.plot_y[p1] + m.plot_width[p1] <= m.plot_y[p2],
        m.plot_y[p2] + m.plot_width[p2] <= m.plot_y[p1],
    ]
```

$$
\begin{aligned}
\min \quad & 2(L+W) && \\
\text{s.t.} \quad & L \geq x_i + \ell_i && \forall i \in N \\
& W \geq y_i + w_i && \forall i \in N \\
& \frac{A_i}{w_i} - \ell_i \leq 0 && \forall i \in N \\
& \begin{bmatrix} Z_{ij}^1 \\ x_i + \ell_i \leq x_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^2 \\ x_j + \ell_j \leq x_i \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^3 \\ y_i + w_i \leq y_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^4 \\ y_j + w_j \leq y_i \end{bmatrix} && \forall i, j \in N,\ i < j \\
& \text{exactly\_one}(Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4) && \forall i, j \in N,\ i < j \\
& 0 \leq x_i \leq X && \forall i \in N \\
& 0 \leq y_i \leq Y && \forall i \in N \\
& 0 \leq w_i \leq U && \forall i \in N \\
& 0 \leq \ell_i \leq V && \forall i \in N \\
& Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4 \in \{\text{True, False}\} && \forall i, j \in N,\ i < j
\end{aligned}
$$

Solving GDPs

- Solvers don't "understand" GDPs
- Most common method: Transform to a MI(N)LP. In `pyomo.gdp`, we have:
  - Bigm ('`gdp.bigm`')
  - Hull ('`gdp.hull`')
  - Cutting planes ('`gdp.cutting_plane`')
  - Between steps ('`gdp.between_steps`')
  - Multiple Bigm ('`gdp.mbigm`')
  - GDP-to-GDP transformations:
    - Basic steps (`apply_basic_step`)
    - Bounds pretransformation ('`gdp.bound_pretransformation`')
    - Partition disjuncts ('`gdp.partition_disjuncts`')
- Alternatively, `contrib.gdpopt` implements metasolvers for GDP models:
  - Logic-based outer approximation
  - Global logic-based outer approximation
  - Disjunctive branch and bound

The World's Best Transformation!

$$\begin{bmatrix} Z_{ij}^1 \\ x_i + \ell_i \le x_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^2 \\ x_j + \ell_j \le x_i \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^3 \\ y_i + w_i \le y_j \end{bmatrix} \vee \begin{bmatrix} Z_{ij}^4 \\ y_j + w_j \le y_i \end{bmatrix}$$

$$\text{exactly\_one}(Z_{ij}^1, Z_{ij}^2, Z_{ij}^3, Z_{ij}^4)$$

$$0 \le x_i \le X$$
$$0 \le y_i \le Y$$
$$0 \le w_i \le U$$
$$0 \le \ell_i \le V$$

BigM

$$x_i + \ell_i - x_j \le 0 + M_1\left(1 - z_{ij}^1\right)$$
$$x_j + \ell_j - x_i \le 0 + M_1\left(1 - z_{ij}^2\right)$$
$$y_i + w_i - y_j \le 0 + M_2\left(1 - z_{ij}^3\right)$$
$$y_j + w_j - y_i \le 0 + M_2\left(1 - z_{ij}^4\right)$$
$$z_{ij}^1 + z_{ij}^2 + z_{ij}^3 + z_{ij}^4 = 1$$
$$z_{ij}^k \in \{0,1\}, \forall k \in \{1,2,3,4\},$$

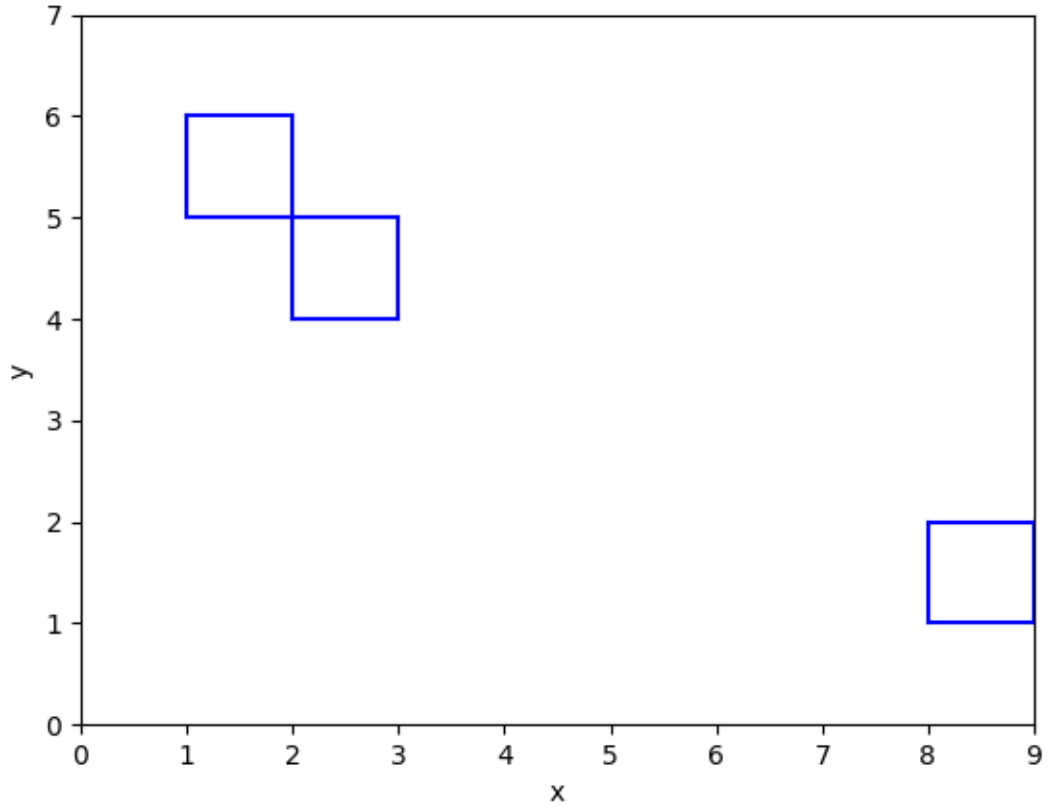
where $M_1 = X + V$ and $M_2 = Y + U$.

…Or is it?

$$\begin{bmatrix} A \\ 1 \le x \le 2 \\ 5 \le y \le 6 \end{bmatrix} \vee \begin{bmatrix} B \\ 2 \le x \le 3 \\ 4 \le y \le 5 \end{bmatrix} \vee \begin{bmatrix} C \\ 8 \le x \le 9 \\ 1 \le y \le 2 \end{bmatrix}$$

$$1 \le x \le 9$$

$$1 \le y \le 6$$



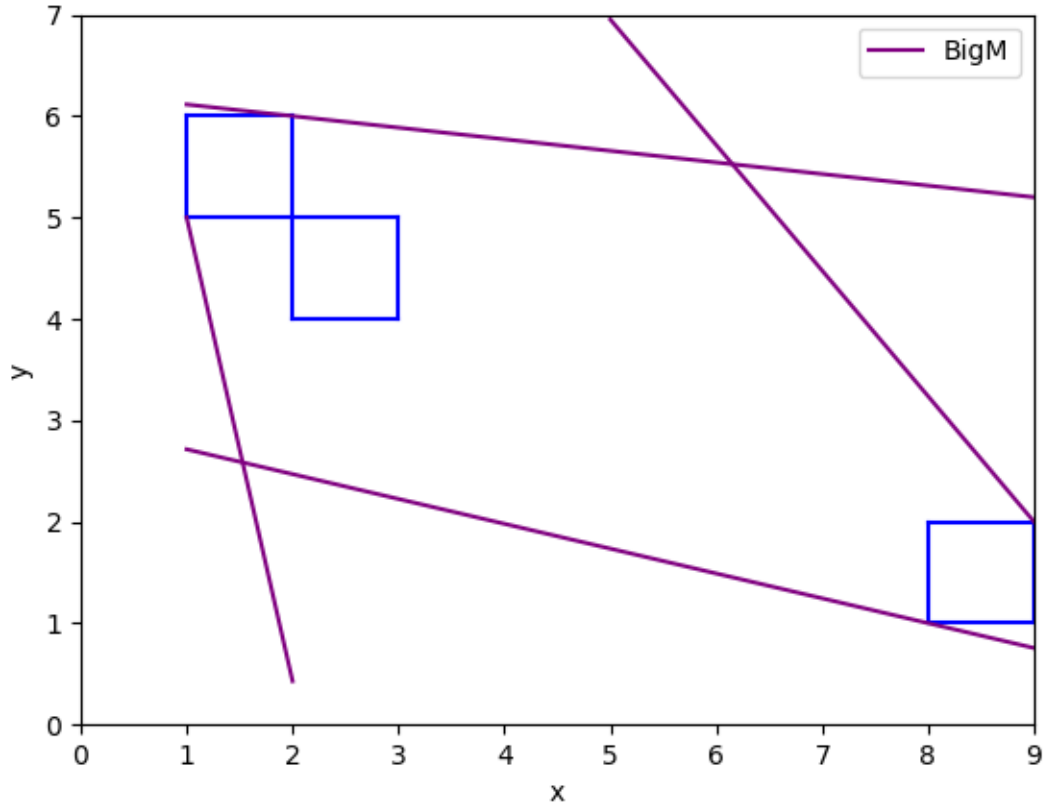
…Or is it?

$$\begin{bmatrix} A \\ 1 \le x \le 2 \\ 5 \le y \le 6 \end{bmatrix} \vee \begin{bmatrix} B \\ 2 \le x \le 3 \\ 4 \le y \le 5 \end{bmatrix} \vee \begin{bmatrix} C \\ 8 \le x \le 9 \\ 1 \le y \le 2 \end{bmatrix}$$

$$1 \le x \le 9$$
$$1 \le y \le 6$$

BigM

$$1 \le x \le 2 + 7(1 - y_1)$$
$$5 - 4(1 - y_1) \le y \le 2 + 4(1 - y_1)$$
$$2 - (1 - y_2) \le x \le 3 + 6(1 - y_2)$$
$$4 - 3(1 - y_2) \le y \le 5 + (1 - y_2)$$
$$8 - 7(1 - y_3) \le x \le 9$$
$$1 \le y \le 2 + 4(1 - y_3)$$
$$y \le 6$$



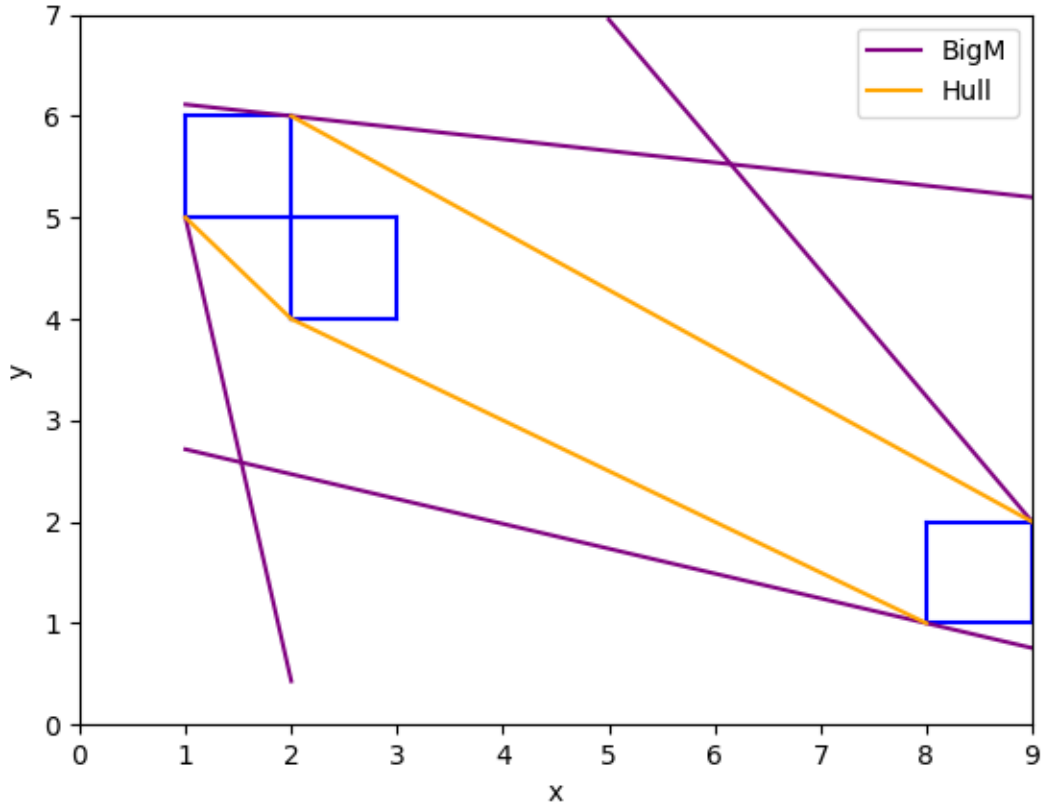
…Or is it?

$$\begin{bmatrix} A \\ 1 \le x \le 2 \\ 5 \le y \le 6 \end{bmatrix} \vee \begin{bmatrix} B \\ 2 \le x \le 3 \\ 4 \le y \le 5 \end{bmatrix} \vee \begin{bmatrix} C \\ 8 \le x \le 9 \\ 1 \le y \le 2 \end{bmatrix}$$

$$1 \le x \le 9$$
$$1 \le y \le 6$$

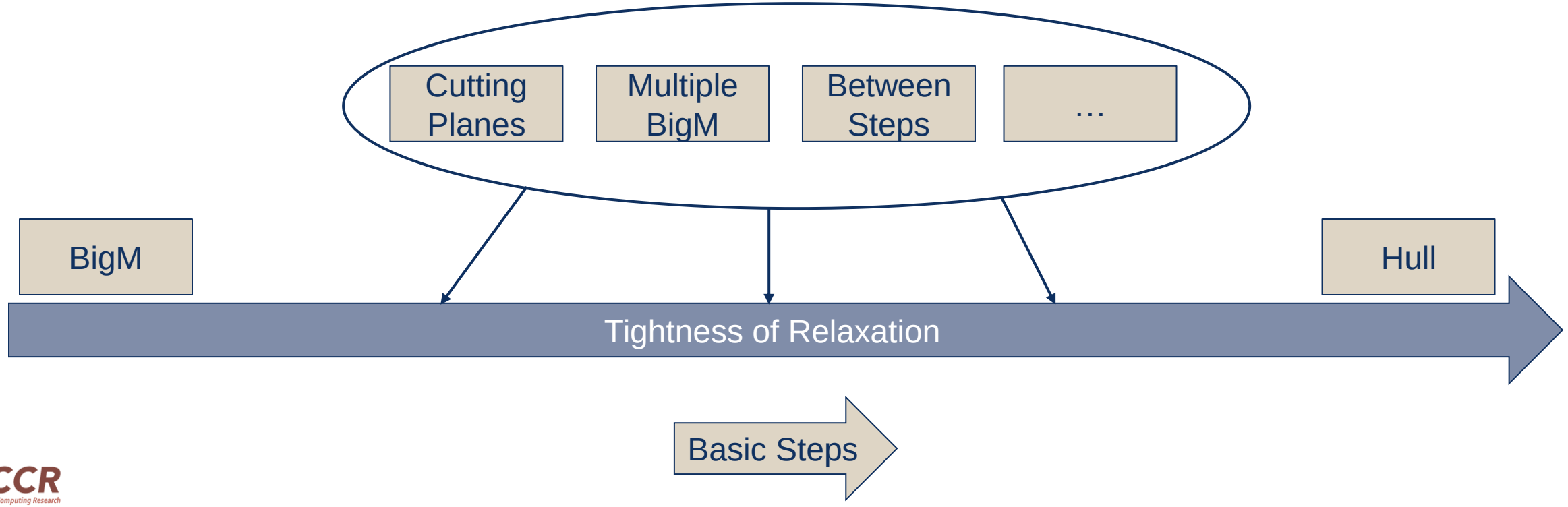
Hull

$$a \le x_A \le 2a$$
$$5a \le y_A \le 6a$$
$$2b \le x_B \le 3b$$
$$4b \le y_B \le 5b$$
$$8c \le x_C \le 9c$$
$$c \le y_C \le 2c$$
$$a \le x_A \le 9a$$
$$a \le y_A \le 6a$$
$$b \le x_B \le 9b$$
$$b \le y_B \le 6b$$
$$c \le x_C \le 9c$$
$$c \le y_C \le 6c$$
$$x = x_A + x_B + x_C$$
$$y = y_A + y_B + y_C$$
$$a + b + c = 1$$
$$a, b, c \in \{0, 1\}$$



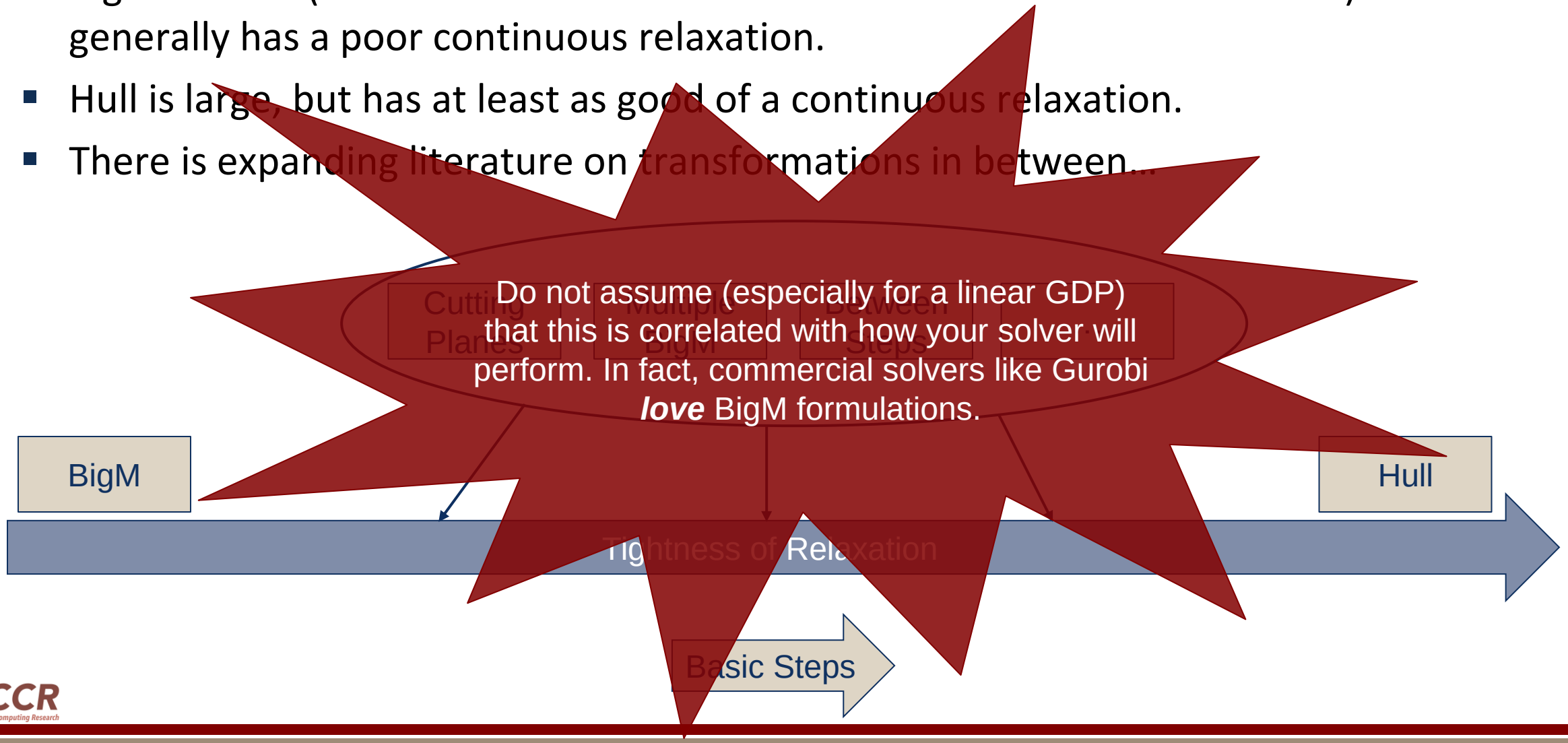
Hierarchy of Relaxations

- BigM is small (in terms of number of variables and number of constraints) but generally has a poor continuous relaxation.
- Hull is large, but has at least as good of a continuous relaxation.
- There is expanding literature on transformations in between.



Hierarchy of Relaxations

- BigM is small (in terms of number of variables and number of constraints) but generally has a poor continuous relaxation.
- Hull is large, but has at least as good of a continuous relaxation.
- There is expanding literature on transformations in between...



Why Formulate Problems as a GDPs?

- (More) Intuitive model formulations
- Model preserves explicit logical structure
- Automated transformations reduce errors
- Staying upwind: Prematurely formulating using one representation makes trying others difficult.
  - Tightness of relaxation is *not* the only factor that determines whether you can solve the problem at scale (especially for MILPs), so you may need to experiment.

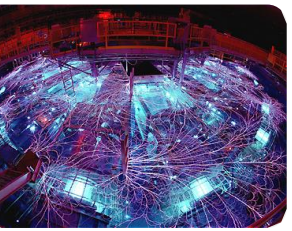
Exercise: Strip Packing

- Given a set of rectangles, and a strip with a fixed width, determine the minimal strip length required such that the rectangles can all be cut out of the single strip (without overlapping).
  - Strip width: 10
  - Rectangles (W*L): [6, 6], [3, 8], [4, 5], [2, 3]

Exercise: Strip Packing

- Given a set of rectangles, and a strip with a fixed width, determine the minimal strip length required such that the rectangles can all be cut out of the single strip (without overlapping).
  - Strip width: 10
  - Rectangles (W*L): [6, 6], [3, 8], [4, 5], [2, 3]
  - "Non overlap":

$$\begin{bmatrix} Y_L \\ x_i + l_i \leq x_j \end{bmatrix} \vee \begin{bmatrix} Y_R \\ x_j + l_j \leq x_i \end{bmatrix} \vee \begin{bmatrix} Y_U \\ y_i + w_i \leq y_j \end{bmatrix} \vee \begin{bmatrix} Y_D \\ y_j + w_j \leq y_i \end{bmatrix}$$



Sandia National Laboratories

Exceptional service in the national interest

Section 14
Development in `pyomo.contrib`

U.S. DEPARTMENT OF ENERGY NNSA National Nuclear Security Administration CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

There is *a lot* in pyomo.contrib

- ampl_function_demo
- appsi
- benders
- community_detection
- cp
- doe
- fbbt
- fme
- gdp_bounds
- gdpopt
- gjh
- iis
- incidence_analysis
- interior_point
- latex_printer
- mcpp
- mindtpy
- mpc
- multistart
- parmest
- piecewise
- preprocessing
- pynumero
- pyros
- satsolver
- sensitivity_toolbox
- trustregion
- viewer

(as of Dec 2023)

APPSI: Auto-Persistent Pyomo Solver Interfaces

- Exploratory solver interface project

- Goals:
  - Automatically (efficiently) update the solver model with any changes to the Pyomo model
    - Relieve the user from manually tracking model changes (difficult, and error prone)
  - Standardize the interface to “persistent” solvers
  - Create “persistent” interfaces to solvers that do not support persistence (e.g., CBC, Ipopt)
  - Explore the use of efficient (C++) data structures and routines for compiling models
  - Allow user to tell solver to bypass specific update checks (additional performance gains)
  - Simplify the solver API and results objects
  - Provide a “LegacySolverInterface” wrapper to make APPSI compatible with old interfaces

- Built prototypes for 5 solvers:
  - Gurobi, HiGHS (Python API)
  - Cplex (LP file + Python API)
  - Ipopt (NL file + subprocess)
  - CBC (LP file + subprocess)

APPSI: Computational Results

(Changing loads from solve to solve; 100 re-solves)

IEEE118 (DCOPF Testcase)

Solver	First solve time	Re-solve time
cbc	0.0531	0.0595
ipopt	0.0483	0.0429
gurobi	0.1654	0.1754
gurobi_direct	0.0277	0.0293
scip	0.0465	0.0452
appsi_cbc	0.0560	0.0457
appsi_ipopt	0.0629	0.0404
appsi_gurobi	0.0319	0.0032
appsi_highs	0.0279	0.0030

← License checks

IEEE1888 (DCOPF Testcase)

Solver	First solve time	Re-solve time
cbc	0.407	0.403
ipopt	0.358	0.283
gurobi	0.408	0.416
gurobi_direct	0.508	0.496
scip	0.356	0.350
appsi_cbc	0.510	0.327
appsi_ipopt	0.603	0.245
appsi_gurobi	0.523	0.027
appsi_highs	0.401	0.029

CP: Constraint Programming in Pyomo

- Logical constraints
  - Boolean expressions
    - `and, or, xor`
    - `not`
    - `implies`
    - `equivalent`
  - General constraint programming-like expressions
    - `atleast, atmost, exactly`

```
m.Y = BooleanVar(range(6))
m.p = LogicalConstraint(expr=lor(m.Y[0], m.Y[1]).implies(
    land(m.Y[2], ~m.Y[3], m.Y[4].lor(m.Y[5])))
)
m.p.pprint()
```

```
p : Size=1, Index=None, Active=True
    Key  : Body                                          : Active
    None : Y[0] ∨ Y[1] --> Y[2] ∧ ~Y[3] ∧ (Y[4] ∨ Y[5]) :   True
```

- Constraint programming (*beta*)
  - `IntervalVar`
  - `Pulse`
  - `Step`
  - *Indirection*

```
m.act = IntervalVar(
    [1, 2, 3], start=(3, 24), end=(5, 24), length=(2, 3)
)
m.i = Var(domain=Integers)
m.c = LogicalConstraint(expr=
    m.act[m.i + 1].start_time.after(m.act[m.i].end_time)
)
```

CP: Solving Logical / Constraint Programming models

- Standard transformations for Logical & General CP-like expressions
  - `core.logical_to_linear`
    - Converts LogicalConstraints to Constraints by constructing the MIP representation of the Conjunctive Normal Form of each LogicalConstraint
      - All logical constraints are converted to MIP equivalents
      - This transformation can be slow (conversion to/from sympy, calculation of the CNF)
  - `contrib.logical_to_disjunctive`
    - Converts LogicalConstraints to a mix of Constraints and Disjunctions by leveraging ideas from *Factorable Programming*, and introducing additional variables to capture values of intermediate expressions in complex constraints.
      - The resulting model may contain disjunctions and require a subsequent GDP transformation (e.g., BigM or Hull)
      - Fast (single pass of each logical expression tree)

- Full Constraint Programming models can be sent to CP solvers
  - Currently, support for IBM ILOG CP Optimizer

CP Scheduling Model

```
import pyomo.environ as pyo
from pyomo.contrib.cp import IntervalVar, Pulse, Step, AlwaysIn

m = pyo.ConcreteModel()
m.eat_cookie = IntervalVar([0, 1], length=8, end=(0, 24), optional=False)
m.eat_cookie[0].start_time.bounds = (0, 4)
m.eat_cookie[1].start_time.bounds = (5, 20)
m.read_story = IntervalVar(start=(15, 24), end=(0, 24), length=(2, 3))
m.sweep_crumbs = IntervalVar(optional=True, length=1, end=(0, 24))
m.do_dishes = IntervalVar(optional=True, length=5, end=(0, 24))
m.num_crumbs = pyo.Var(domain=pyo.Integers, bounds=(0, 100))

## Precedence
m.cookies = pyo.LogicalConstraint(expr=m.eat_cookie[1].start_time.after(m.eat_cookie[0].end_time))
m.cookies_imply_crumbs = pyo.LogicalConstraint(expr=m.eat_cookie[0].is_present.implies(m.num_crumbs == 5))
m.good_mouse = pyo.LogicalConstraint(expr=pyo.implies(m.num_crumbs >= 3, m.sweep_crumbs.is_present))
m.sweep_after = pyo.LogicalConstraint(expr=m.sweep_crumbs.start_time.after(m.eat_cookie[1].end_time))

m.mice_occupied = sum(Pulse((m.eat_cookie[i], 1)) for i in range(2)) + Step(m.read_story.start_time, 1) + \
                  Pulse((m.sweep_crumbs, 1)) - Pulse((m.do_dishes,1))

# Must keep exactly one mouse occupied for a 25-hour day
m.treat_your_mouse_well = pyo.LogicalConstraint(expr=AlwaysIn(cumul_func=m.mice_occupied, bounds=(1, 1), times=(0, 24)))

results = pyo.SolverFactory('cp_optimizer').solve(m, symbolic_solver_labels=True, tee=True)
```

CP Scheduling Model

```
import pyomo.environ as pyo
from pyomo.contrib.cp import IntervalVar, Pulse, Step, AlwaysIn

m = pyo.ConcreteModel()
m.eat_cookie = IntervalVar([0, 1], length=8, end=(0, 24), optional=False)
m.eat_cookie[0].start_time.bounds = (0, 4)
m.eat_cookie[1].start_time.bounds = (5, 20)
m.read_story = IntervalVar(start=(15, 24), end=(0, 24), length=(2, 3))
m.sweep_crumbs = IntervalVar(optional=True, length=1, end=(0, 24))
m.do_dishes = IntervalVar(optional=True, length=5, end=(0, 24))
m.num_crumbs = pyo.Var(domain=pyo.Integers, bounds=(0, 100))

## Precedence
m.cookies = pyo.LogicalConstraint(expr=m.eat_cookie[1].start_time.after(m.eat_cookie[0].end_time))
m.cookies_imply_crumbs = pyo.LogicalConstraint(expr=m.eat_cookie[0].is_present.implies(m.num_crumbs == 5))
m.good_mouse = pyo.LogicalConstraint(expr=pyo.implies(m.num_crumbs >= 3, m.sweep_crumbs.is_present))
m.sweep_after = pyo.LogicalConstraint(expr=m.sweep_crumbs.start_time.after(m.eat_cookie[1].end_time))

m.mice_occupied = sum(Pulse((m.eat_cookie[i], 1)) for i in range(2)) + Step(m.read_story.start_time, 1) + \
                  Pulse((m.sweep_crumbs, 1)) - Pulse((m.do_dishes,1))

# Must keep exactly one mouse occupied for a 25-hour day
m.treat_your_mouse_well = pyo.LogicalConstraint(expr=AlwaysIn(cumul_func=m.mice_occupied, bounds=(1, 1), times=(0, 24)))

results = pyo.SolverFactory('cp_optimizer').solve(m, symbolic_solver_labels=True, tee=True)
```

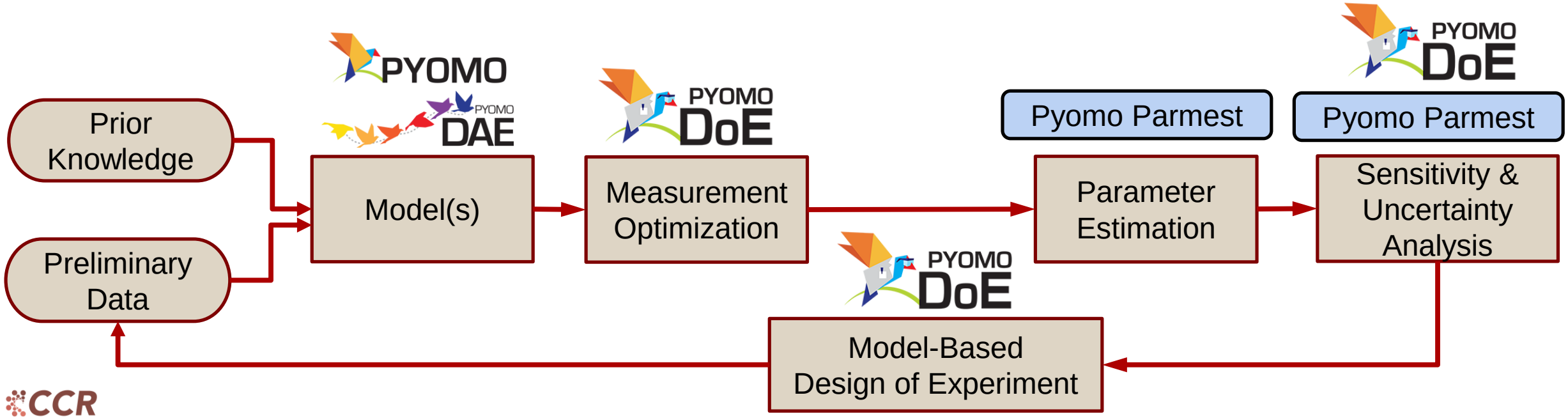
`IntervalVars` specify activities to be scheduled: choose start time and end time, and (if `optional=True`) whether or not it is scheduled

Cumulative functions (`Pulse`, `Step`) track resource usage.

`AlwaysIn` constrains resource usage

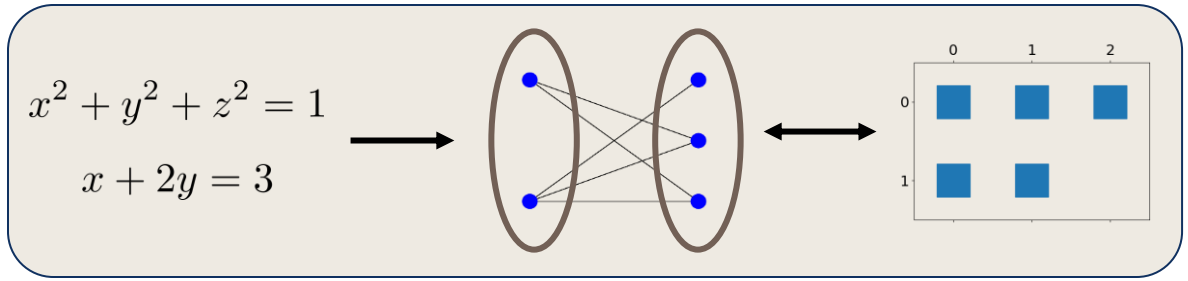
DoE: Model-based Design of Experiments

- Model-based Design of Experiments in Pyomo (J. Wang, A. Dowling)
  - Given:
    - Pyomo model, nominal parameter values, experimental design variables, covariance matrix
  - Compute Fisher information matrix
  - Perform exploratory analysis (enumeration)
  - Compute A- or D-optimal experimental design (via 2-stage stochastic programming)

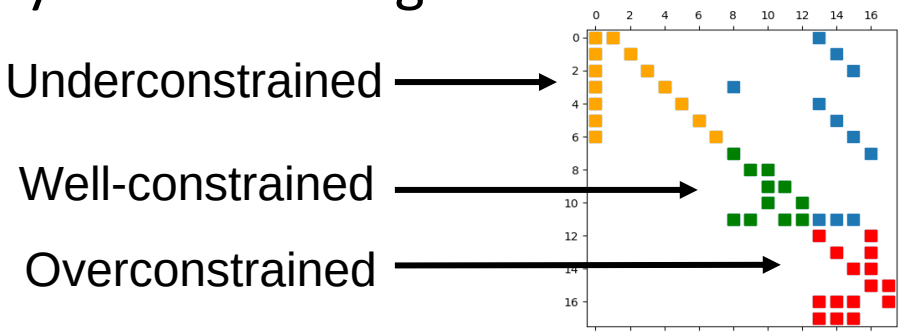


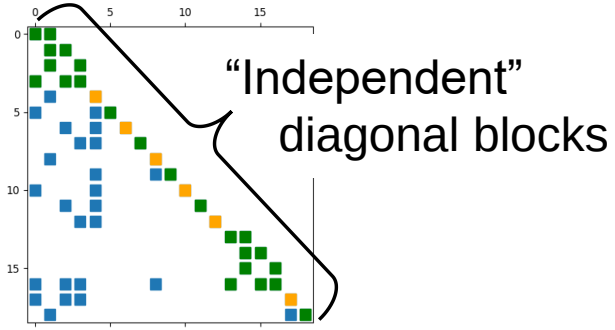
Incidence Analysis: static analysis of nonlinear models

- Motivation: formulating nonlinear chemical process optimization problems *without making mistakes* is difficult
- Goal: develop "static analysis" tools for nonlinear optimization models
  - Move beyond "nonlinear programming folklore" [1]
  - Identify singularities and their sources
- Approach: construct and analyze the bipartite incidence graph of variables and constraints



- Result: Block triangularization and Dulmage-Mendelsohn tell us whether and why systems are singular





[1] Tasseff, Coffrin, Wächter, and Laird. https://arxiv.org/pdf/1909.08104.pdf

Piecewise: Piecewise-linear approximations of MINLP models

- Support for multivariate piecewise linear approximations
  - Implement (and generalize) Vielma et al.'s "library" of MIP formulations of continuous piecewise-linear functions through Generalized Disjunctive Programming (GDP):

	Bigm	Multiple BigM	Hull	Ad-hoc
Inner Repr. GDP			Disaggregated Convex Combination Model	
Reduced-Space Inner Repr. GDP		Convex Combination Model		
Outer Repr. GDP			Multiple Choice Model	
Nested GDP		Logarithmic Convex Combination Model	Logarithmic Disaggregated Convex Combination Model	
Ad-hoc				Incremental Model (includes state-of-the-art decision tree formulations)

PyNumero: nonlinear algorithms in python

- High-level Python interface for creating (performant) serial and parallel block-decomposition approaches (using direct solvers)
  - Algorithms constructed in a high-level language (Python)
  - Computationally expensive operations performed in compiled code

- Key capabilities:
  - Interfaces to ASL and Pyomo models
  - Supports standard NLP interfaces
    - (e.g., expression evaluation, gradients, Jacobians, Hessians, etc.)
  - Supports linear algebra operations with Numpy/SciPy – both dense and sparse
  - Supports block-structured construction of matrices, e.g.:

```
KKT = BlockMatrix(2,2)
KKT.setblock(0,0,H)
KKT.setblock(1,0,J)
```

  - Supports serial and distributed parallel block matrices
  - Interfaces to commonly-used linear solvers
    - MUMPs, ma27, and all of Numpy/SciPy

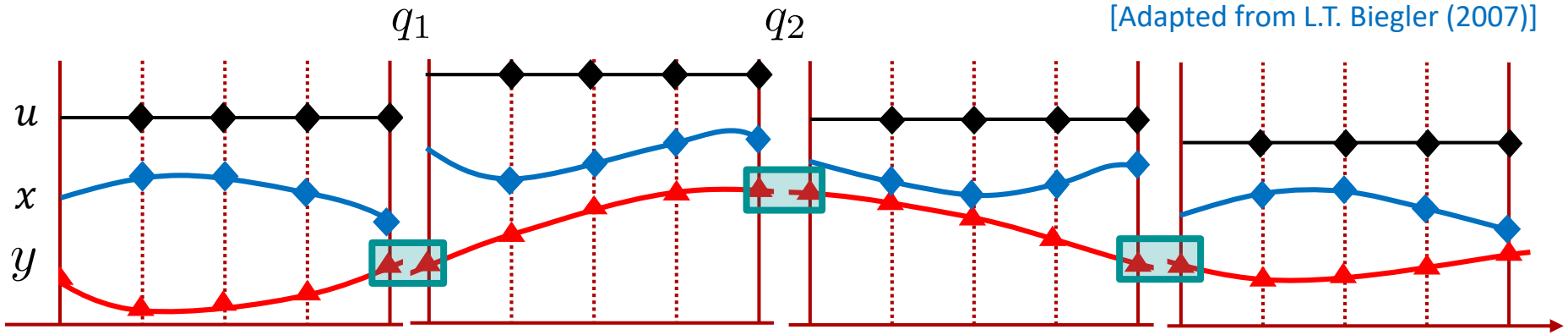
PyNumero: Simple algorithms (Newton's Method)

Algorithm 1: Newton Method

Given nonlinear system $F(x)$,
initial guesses x^0
and tolerance $\epsilon > 0$
for $k = 0, 1, 2, \ldots$ **do**
  Compute step direction:
  $\nabla F(x^k) dx^k = -F(x^k)$
  update variables:
    $x^{k+1} = x^k + dx^k$

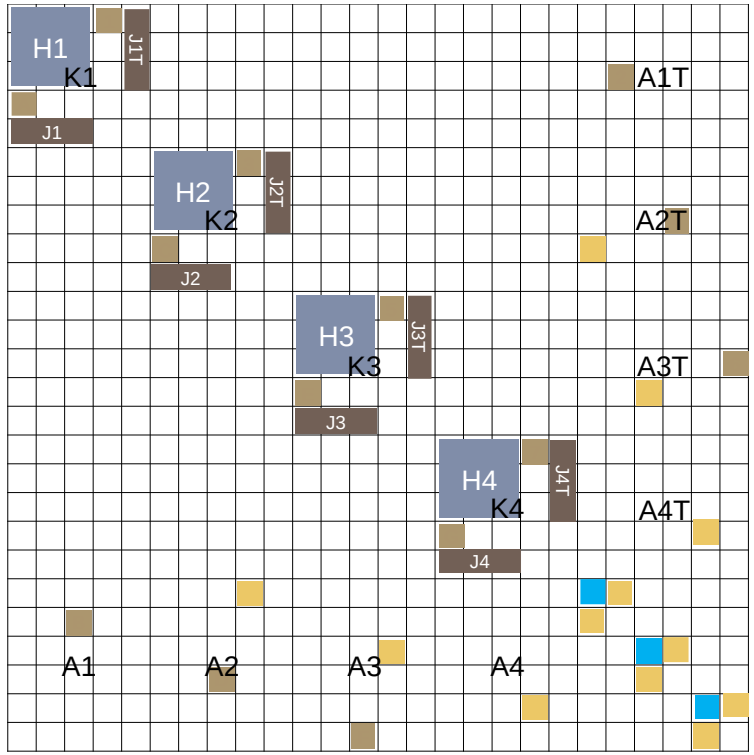
```
from pyomo.contrib.pynumero.interfaces  import PyomoNLP
from scipy.sparse.linalgimport spsolve
import pyomo.environ as aml
import numpy as np

model = create_pyomo_model()
nlp = PyomoNLP(model)
x = nlp.x_init()
dx = np.zeros(nlp.nx)

for k in range(100):
    if np.linalg.norm(rhs) < 1e-5:
        break
    rhs = nlp.evaluate_c(x)
    jac = nlp.jacobian_c(x)
    dx = -spsolve(jac, rhs)
    x += dx
```

PyNumero: parallel solution of DAE systems



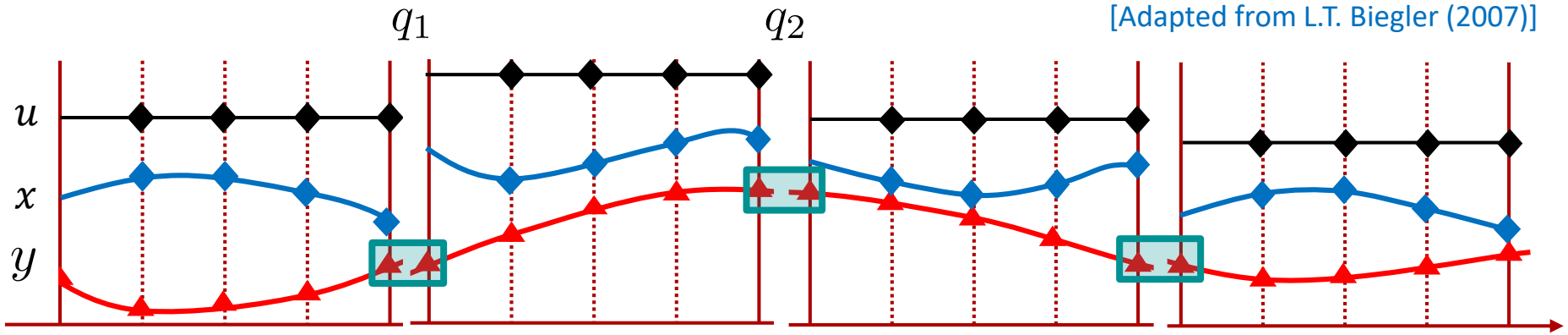
[Adapted from L.T. Biegler (2007)]

- Factor all K_i matrices
- Form Schur-complement
  - Backsolve of each K_i for each nonzero column in A_i to form local Schur-complement S_i
- Compute $S = \sum_i S_i$ (MPI Reduction)
- Solve $S\Delta q = r_s$
- Solve $K_i \Delta z_i = r_i$

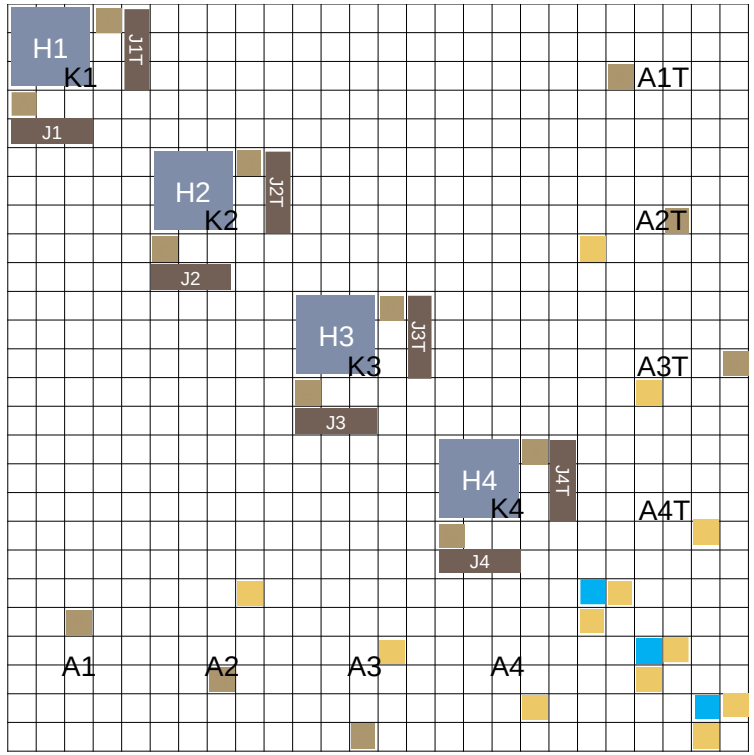


PyNumero: parallel solution of DAE systems

[Adapted from L.T. Biegler (2007)]

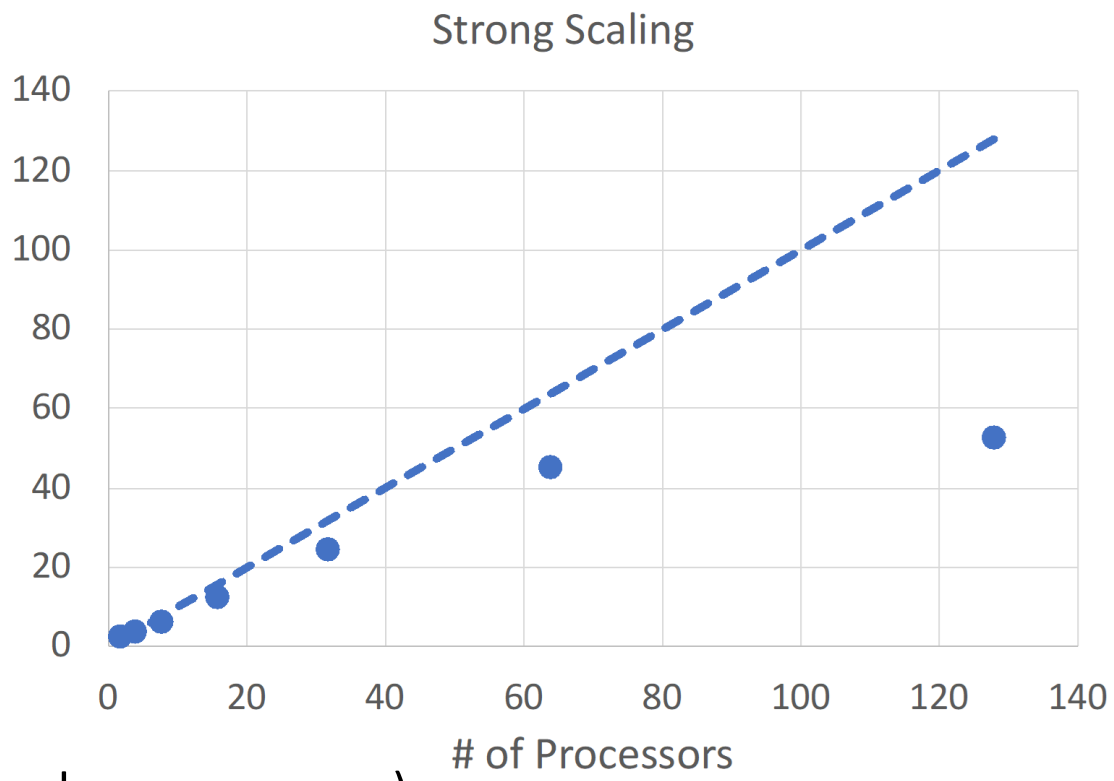


- Factor all K_i matrices ($\downarrow$)
- Form Schur-complement ($\approx$ or $\downarrow$)
  - Backsolve of each K_i for each nonzero column in A_i to form local Schur-complement S_i
- Compute $S = \sum_i S_i$ (MPI Reduction) ($\uparrow$ communication)
- Solve $S\Delta q = r_s$ ($\uparrow$)
- Solve $K_i \Delta z_i = r_i$ ($\downarrow$)



PyNumero: Computational Performance

- Test problem
  - DAE problem with
    5 states / 200 algebraics
  - 12800 finite elements in time
  - x is small (dimension 5)
- Implementation Details
  - Problem implemented in Pyomo
  - Algorithm implemented in PyNumero
  - Chama HPC
  - Using 8 cores per node



- Working on larger test problems (more work per processor)
- Test problems with more coupling (e.g., more states)
  - Increases serial SC factorization time (looking at parallel / implicit approaches)
  - Increases communication time (looking at sparse versions of reduction)

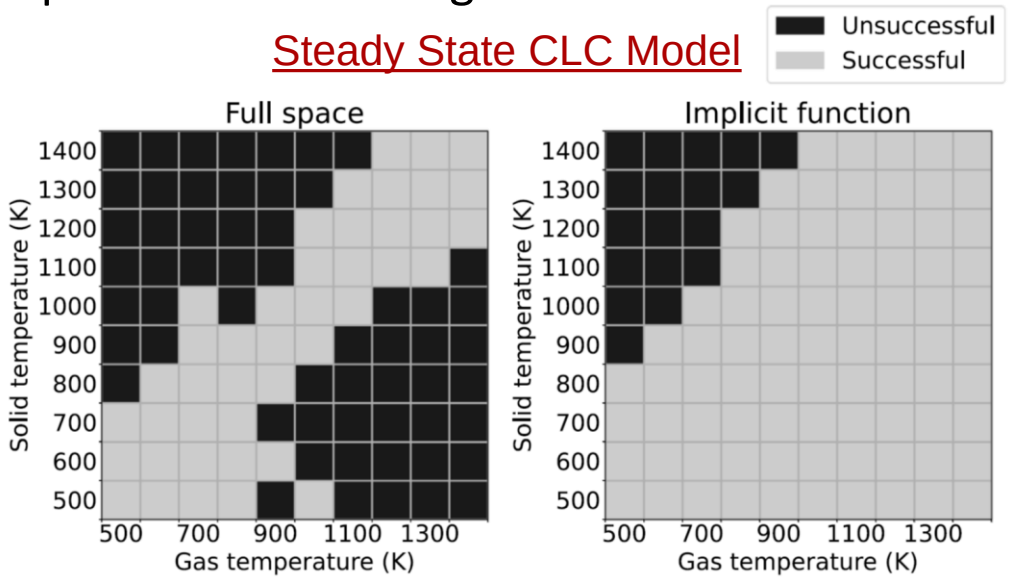
PyNumero: `cyipopt` interface (Python bindings to Ipopt)

- cyipopt provides general python bindings to ipopt.
  - Pyomo has a prototype interface to cyipopt through pynumero
    - Leverages pynumero's bindings to the ASL to get efficient model evaluations
    - Can be combined with other function interfaces
      - Merge multiple NL files into a single problem
      - Support *python* external functions
      - Support *vector-valued* external functions
    - Can generate *masked interfaces* to models that "hide" variables / constraints
  - cyipopt supports callbacks *during the ipopt execution*, enabling
    - Real-time diagnostics
    - Reporting the optimizer's path through state space
- Open challenge: cyipopt is distributed source-only
  - Recent work on idaes-ext is aiming to make building cyipopt against the IDAES ipopt (with HSL) seamless on most (if not all) supported IDAES platforms

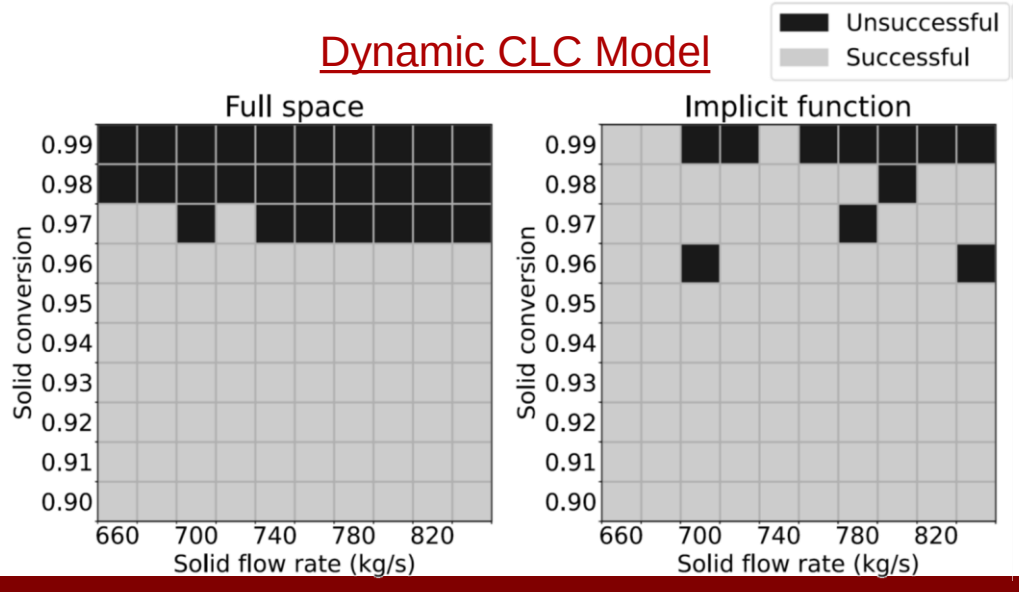
PyNumero: Example cyipopt application: implicit functions

- Using “implicit functions” to improve dynamic model convergence [R. Parker]
- Main idea:
  - Extract algebraic equations from the model Ipopt; solve as a vector-valued external function
- Implementation:
  - Vector-valued external function (through PyNumero’s ExternalGreyBox)
  - PyNumero to “glue” the external function to the rest of the model (written to NL file)
  - Cyipopt to solve the integrated model

Steady State CLC Model



Dynamic CLC Model

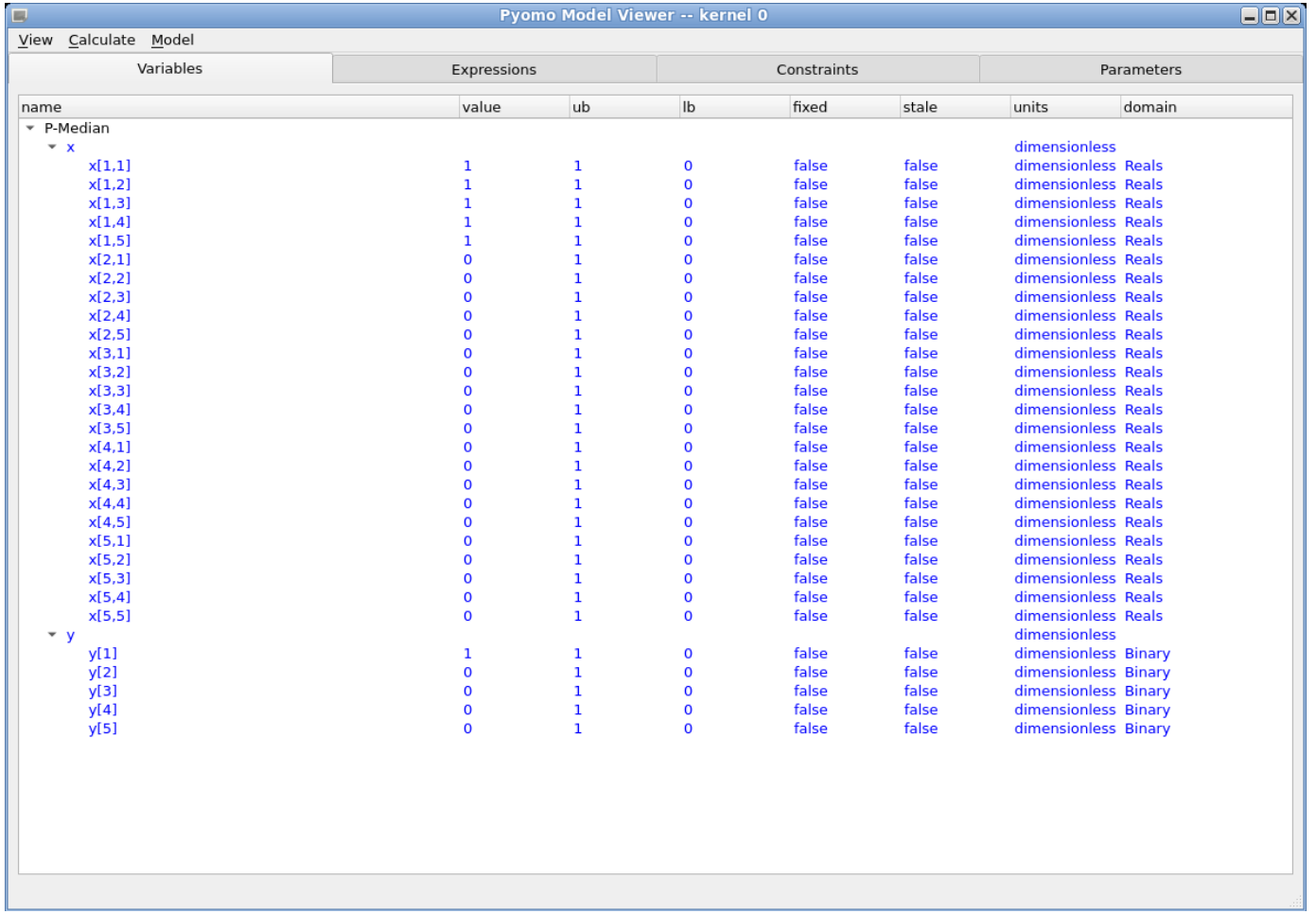


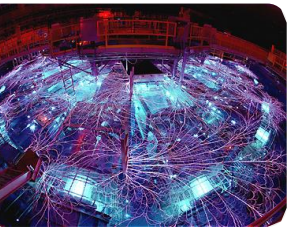
Viewer: QT-based model explorer

- Running pyomo model-viewer brings up an IPython console
  - Running “View…Show Pyomo Model Viewer” brings up the viewer
  - After loading your model into the IPython terminal, load it into the view with

    ```
    ui.set_model(model)
    ```

  - The viewer is *live* and changes in the model are reflected in the viewer
    - And the reverse! Changes in the viewer updated the model!
- This is particularly useful for models with deep block hierarchies





Sandia National Laboratories

Exceptional service in the national interest

Section 15
Debugging Pyomo Models

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration | CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525

Always double check your model understanding

- Does the model have the expected number of degrees of freedom?
- Are the model expressions correct?
- Are you using consistent units?
- Does the solver's assessment of the model match your expectations?
  - i.e. reported number of constraints and variables, types of variables (continuous, binary)

Tools for interrogating Pyomo models

- Pyomo utilities
  - log_close_to_bounds (pyomo.util.infeasible)
  - log_infeasible_constraints (pyomo.util.infeasible)
  - report_scaling (pyomo.util.report_scaling)
  - assert_units_consistent, identify_inconsistent_units (pyomo.util.check_units)
  - …
- IDAES-PSE Diagnostics Toolbox
  - Must start with a square model (zero degrees of freedom)
- Debugging performance issues
  - report_timing (pyomo.common.timing)

Pyomo utility examples

- log_close_to_bounds

```
import logging
logging.basicConfig(level=logging.INFO)

from pyomo.util.infeasible import log_close_to_bounds
log_close_to_bounds(m)
```

- log_infeasible_constraints

```
from pyomo.util.infeasible import log_infeasible_constraints
log_infeasible_constraints(m)
```

Pyomo utility examples

- report_scaling

```
from pyomo.util.report_scaling import report_scaling
report_scaling(m)
```

- assert_units_consistent, identify_inconsistent_units

```
from pyomo.util.check_units import assert_units_consistent
assert_units_consistent(m)
```

IDAES-PSE Diagnostics Toolbox

```
from idaes.core.util import DiagnosticsToolbox
dt = DiagnosticsToolbox()
dt.report_structural_issues(m)
dt.report_numerical_issues(m)
```

- Note: These tools must be applied to square models (no degrees of freedom)

Debugging exercises

- Try these tools on models from the nonlinear exercises

```
import logging
logging.basicConfig(level=logging.INFO)

from pyomo.util.infeasible import log_close_to_bounds, log_infeasible_constraints
log_close_to_bounds(m)
log_infeasible_constraints(m)

from pyomo.util.report_scaling import report_scaling
report_scaling(m)

from pyomo.util.check_units import assert_units_consistent
assert_units_consistent(m)

from idaes.core.util import DiagnosticsToolbox
dt = DiagnosticsToolbox()
dt.report_structural_issues(m)
dt.report_numerical_issues(m)
```