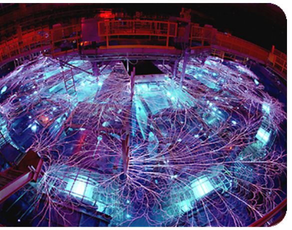
SAND2024-09191C



PYOMO

Pyomo: Design Paradigms and Lessons Learned

Bethany Nicholson
Sandia National Laboratories

on behalf of the Pyomo Team
(Michael Bynum, Bill Hart, Emma Johnson, Carl Laird, Miranda Mundt, Robby Parker, John Siirola, Jean-Paul Watson, David Woodruff, and the 20+ other people who have committed to the code in the last year)

JuMP-dev
July 2024

Sandia National Laboratories

Exceptional service in the national interest

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”?

Sandia National Laboratories

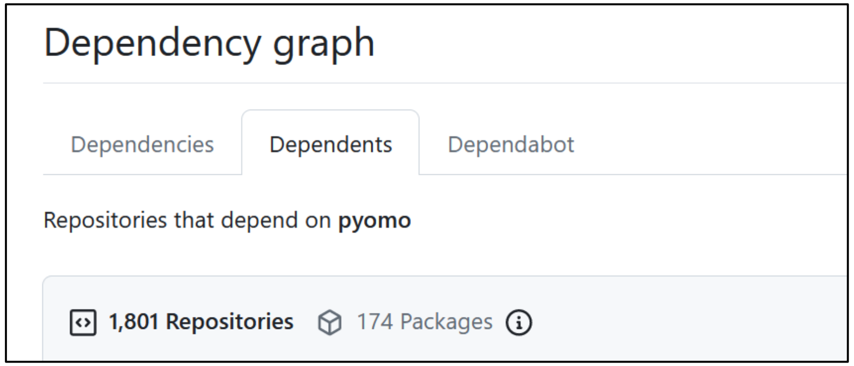
- “An open source object-oriented algebraic modeling language in Python for structured optimization problems.”
  - A Python package
  - A set of objects, classes, methods, and utilities for expressing optimization problems
  - A growing collection of utilities for manipulating optimization problems
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

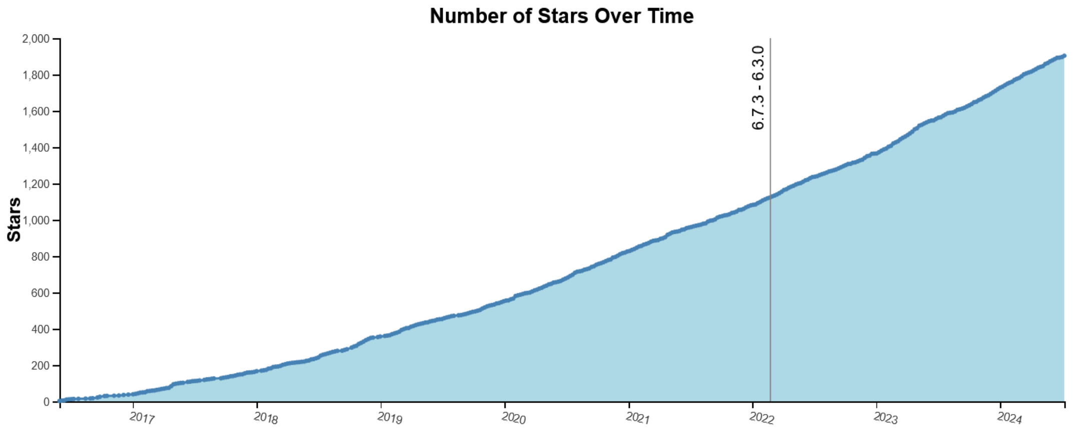
- … and a really cool origami bird

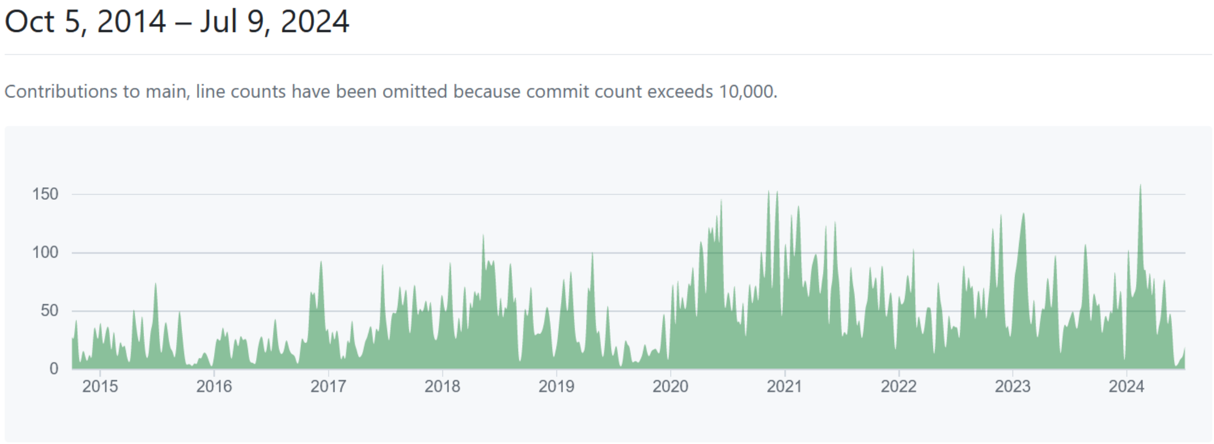
Thanks, Doug Prout!

History of Pyomo

- First released in 2008 as the Coopr software library
- Rebranded as Pyomo around 2011
- Moved to GitHub in mid-2016
- Steady growth in usage and popularity ever since
- In the last year:
  - 5 releases (6.6.2, 6.7.0, 6.7.1, 6.7.2, 6.7.3)
  - 239 merged Pull Requests (from 32 developers)







Pyomo: a growing ecosystem

Sandia National Laboratories

Derived DSMLs

- IDAES – Institute for the Design of Advanced Energy Systems
- PROMMIS – Process Optimization and Modeling for Minerals Sustainability
- DISPATCHES – Design Integration and Synthesis Platform to Advance Tightly Coupled Hybrid Energy Systems
- PARETO – The Produced Water Optimization Initiative
- WaterTAP
- PRESCIENT

External extensions

- *GALINI*
- *mpi-sppy*
- *OMLT*
- *PAO*
- *SUSPECT*

PYOMO

Meta-Solvers

- *Generalized Benders*
- *GDPopt*
- *MindtPy*
- *PyROS*
- *Linear MPEC*

Modeling Extensions

- *Constraint Programming*
- *Disjunctive programming*
- *Stochastic programming*
- *Differential equations*
- *Equilibrium constraints*

Solver Interfaces

Core Optimization Objects

Core Modeling Objects

Model Transformations

Solvers

- BARON
- CBC
- CPLEX
- Cyipopt
- GLPK
- Gurobi
- HiGHS
- Mosek
- Xpress
- SAS
- MAiNGO
- cp-optimizer

AMPL Solver Library

- Bonmin
- Couenne
- DAKOTA
- Ipopt
- KNITRO
- ...

GAMS Solver Library

- ANTIGONE
- DICOPT
- ...

NEOS Server

- Gurobi
- Ipopt
- ...

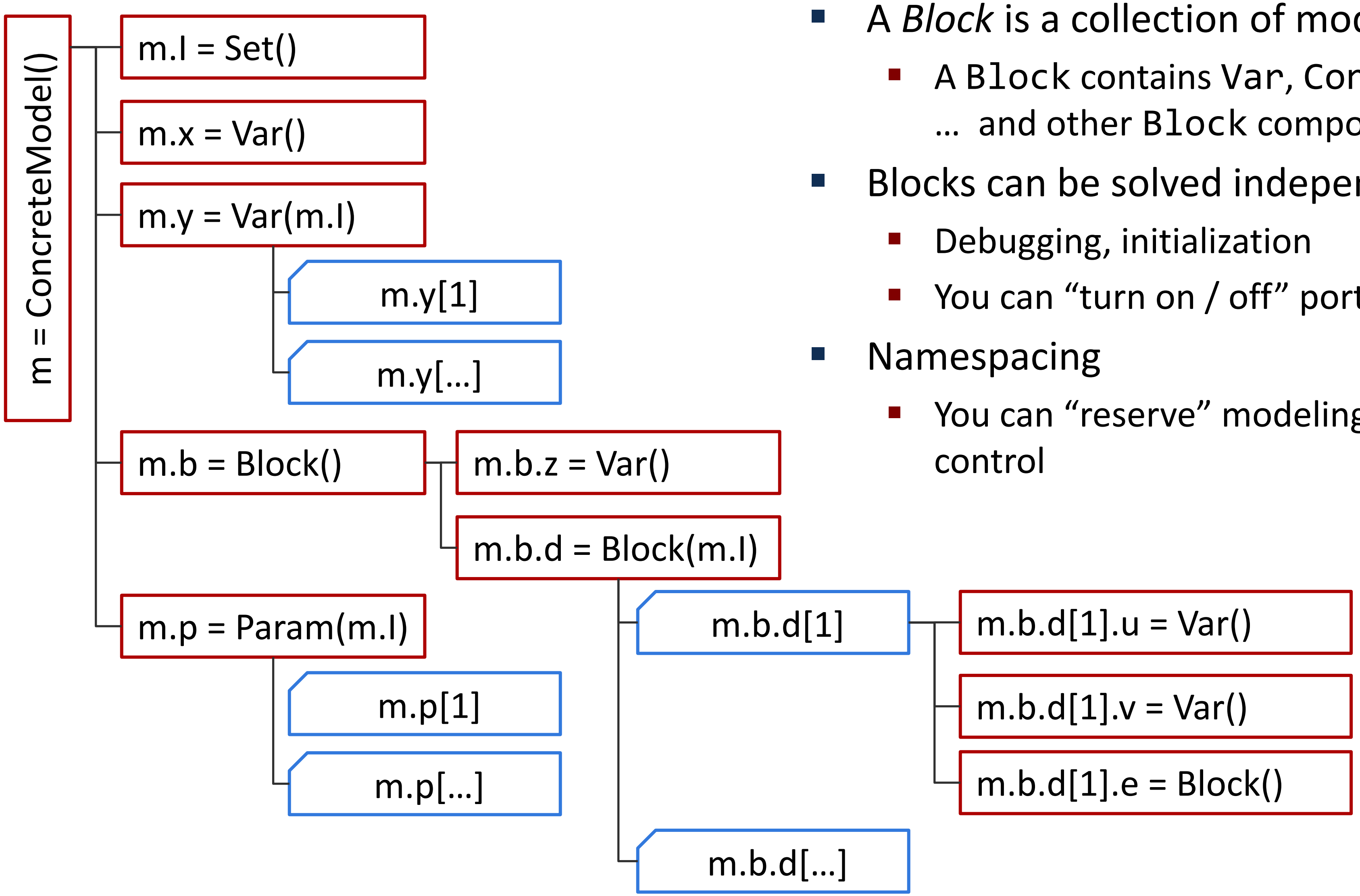
Why am I here?

- Design paradigms we think we "got right"
- Design lessons we've learned
- New features and packages
- What's next

Motivating design principles for Pyomo

- We wanted to express high-level model structure:
  - Use structures and expressions that match our understanding of the system
  - Formulate large models with a concise syntax
  - Composition, logic, dynamics, multi-level optimization
- We wanted to explore new algorithms and approaches:
  - Manage the translation from *what the user said* to *what the solver understands*
  - Decomposition, relaxations, model reformulations, iterative analysis algorithms
- We wanted to build domain-specific optimization libraries
  - Make it easier for researchers to make their innovations available to the community (and us)
  - Electric grid model libraries, process model libraries, specialized tools for asset scheduling

How can we capture *structure* in optimization models?

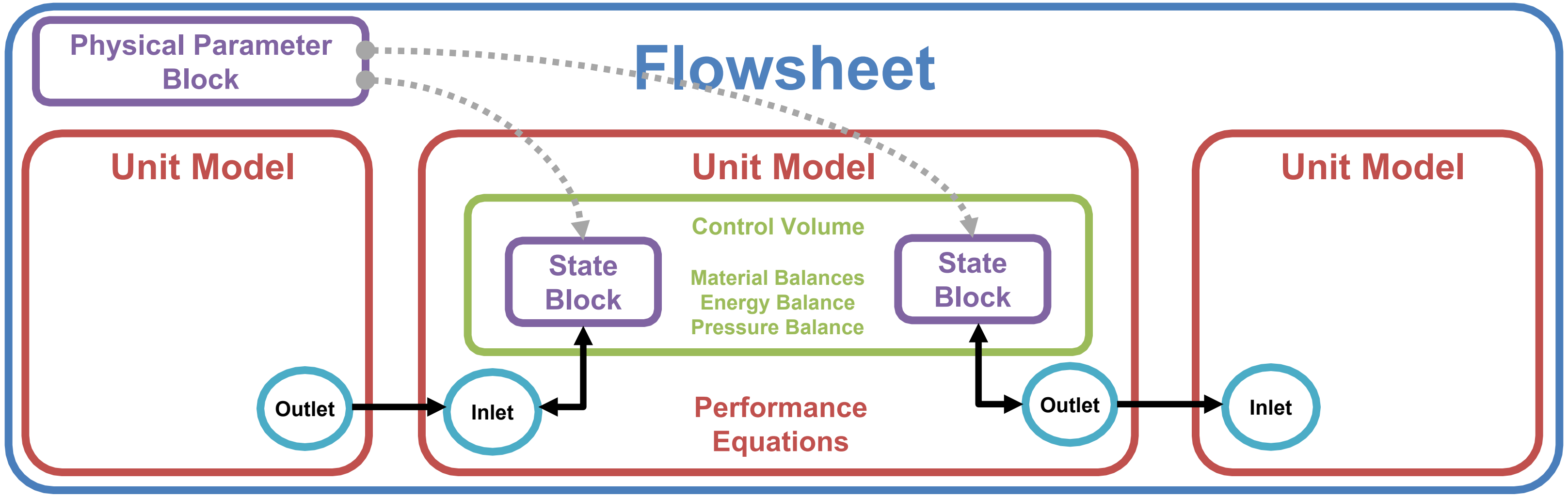


- A *Block* is a collection of modeling components
  - A `Block` contains `Var`, `Constraint`, `Objective`, `Param`, … and other `Block` components
- Blocks can be solved independently of the rest of the model
  - Debugging, initialization
  - You can "turn on / off" portions of the model
- Namespacing
  - You can "reserve" modeling spaces where you have complete control

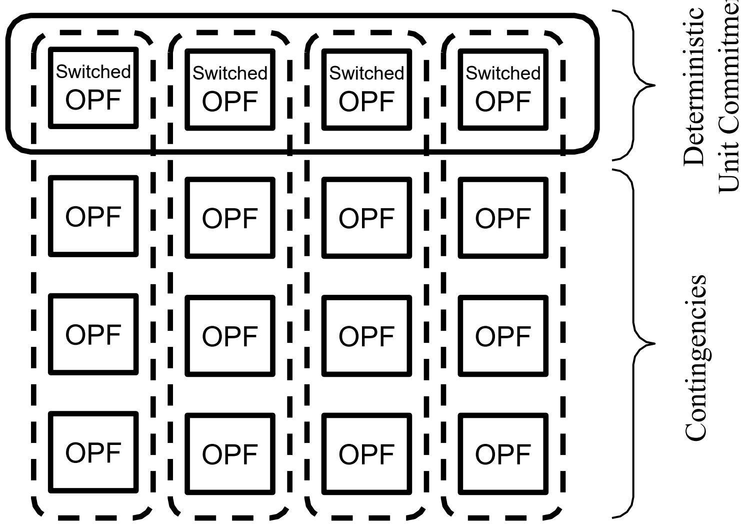
Hierarchical modeling is core to Pyomo

- Blocks facilitate *model composition*

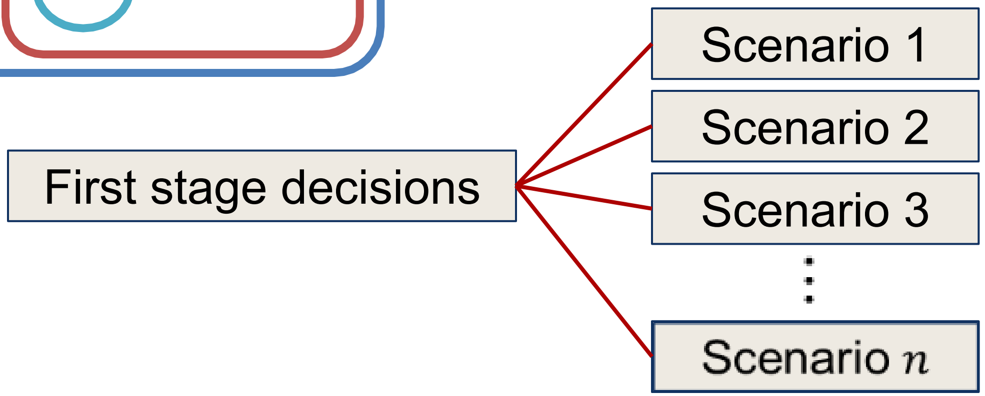
Process Modeling



Grid Modeling

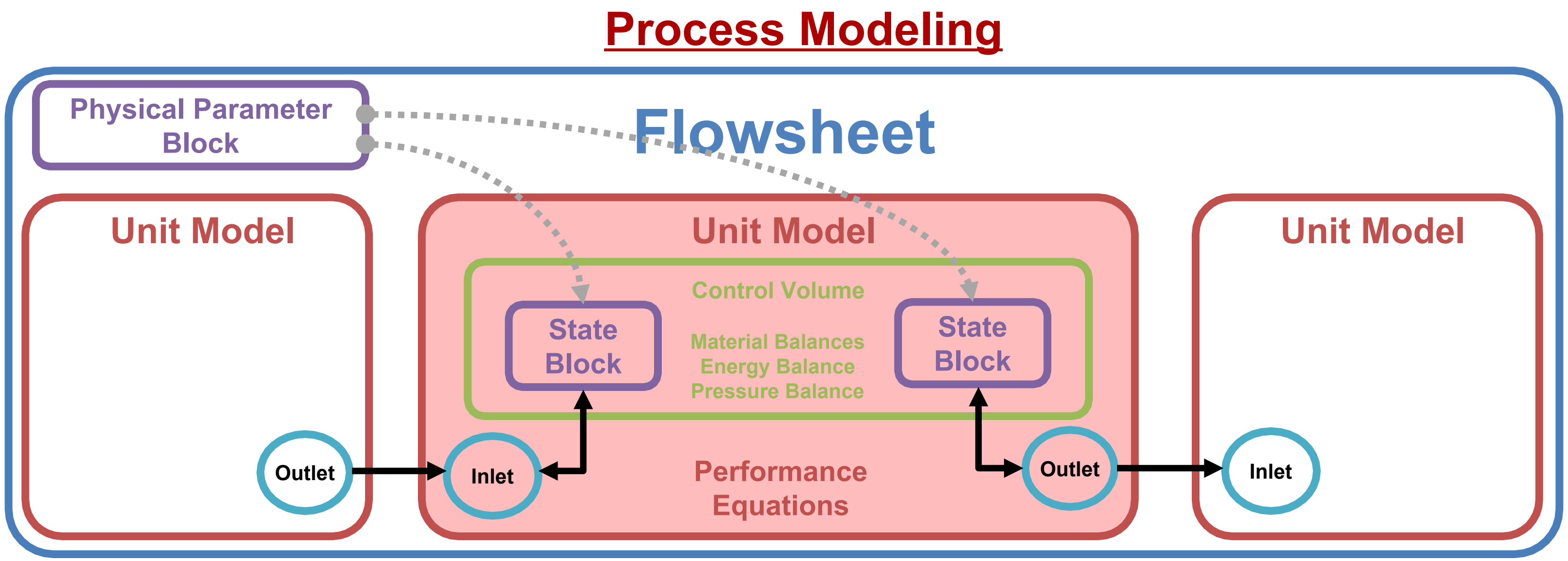


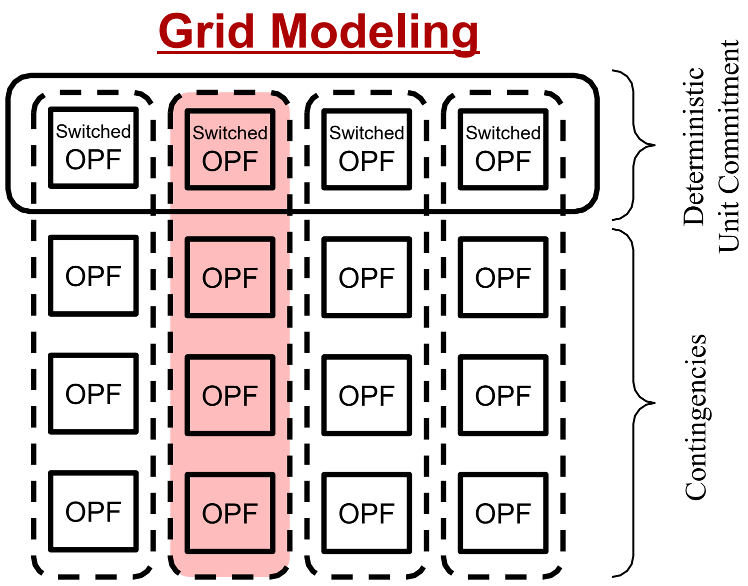
Stochastic Programming

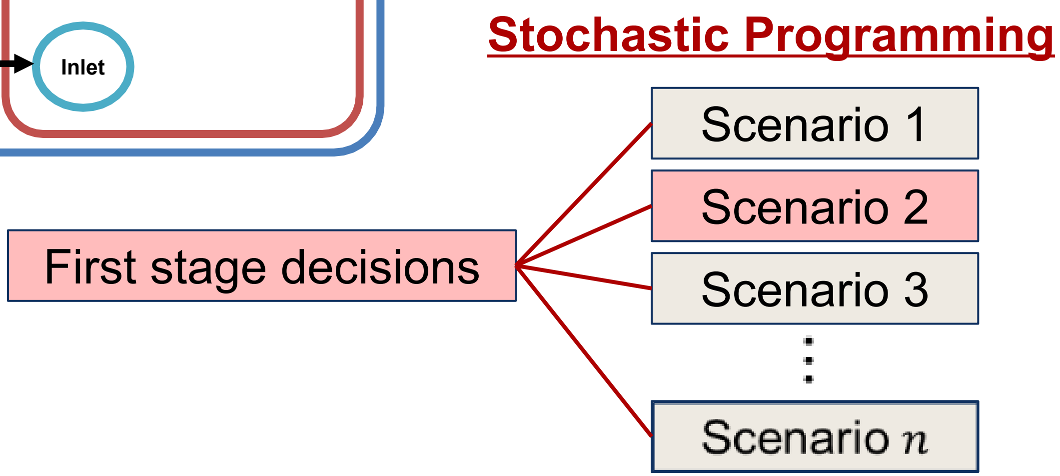


Hierarchical modeling is core to Pyomo

- Blocks facilitate *model composition*
- *and* decomposition

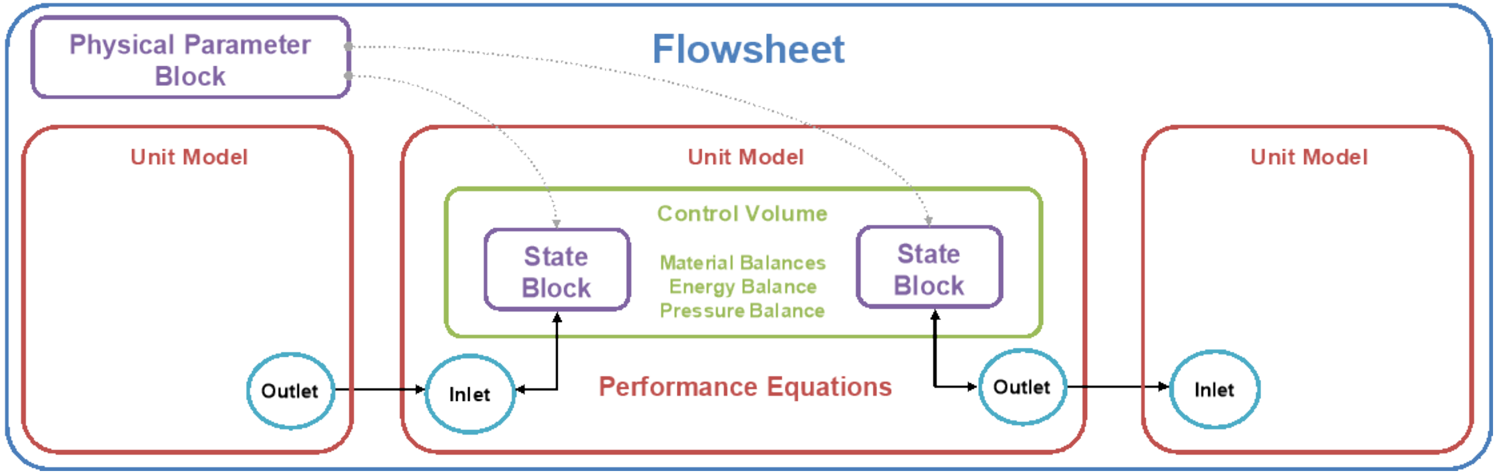






Model decomposition: *what's a model?*

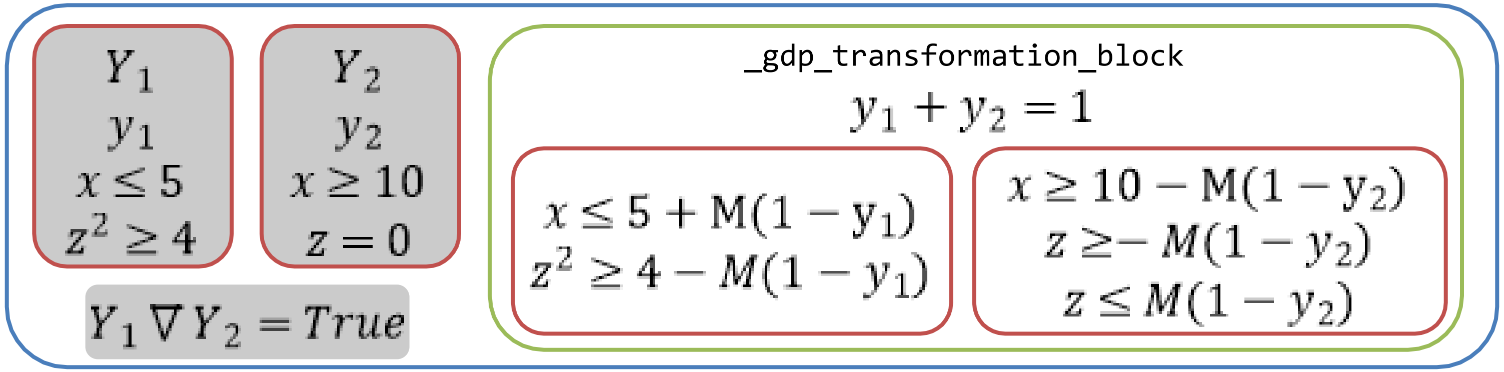
- It is convenient to think of a solvable "model" as a everything contained below a *single block* in the block hierarchy
  - And that (potentially sub-)tree must be self contained (declare all variables, constraints, etc)
    - This is natural for *composition-based modeling*



    - But can break down in the context of *transformations*

$$\begin{bmatrix} Y_1 \\ x \leq 5 \\ z^2 \geq 4 \end{bmatrix} \vee \begin{bmatrix} Y_2 \\ x \geq 10 \\ x = 0 \end{bmatrix}$$

$$Y_1 \veebar Y_2 = True$$



The block hierarchy has influenced "what's a model"

- Pyomo (and the Pyomo development team) has refined the definition of "a model"

  - "The collection of all *active components* reachable by descending through active Blocks starting from a reference Block"

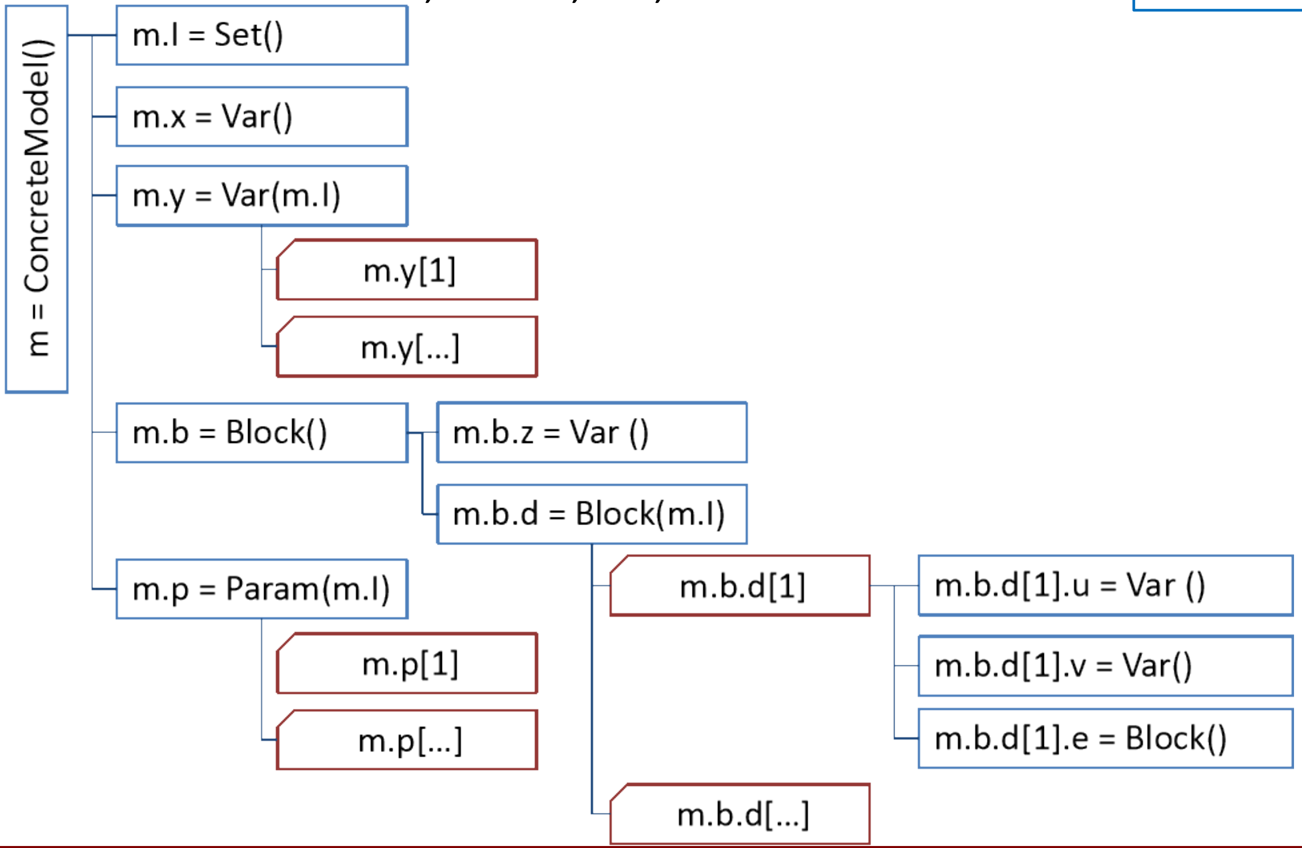
  - "Active Components": e.g., Block, Constraint, Disjunct, Suffix
    - What's *not* an active component: Param, Set, Var

  - This is a relaxation of the previous definition, which required *all* components used in the model be reachable by walking the block hierarchy
    - Better supports solving individual blocks within a larger model
      - constraints in the block can reference variables outside the subtree defined by the block
    - Cleaner handling of implicit sets: dynamically created indexing sets are no longer explicitly attached to the model (and no longer need to be named)

  - Currently promulgating this change through the writers
    - LP, NL, APPSI complete; BAR and GMS in progress

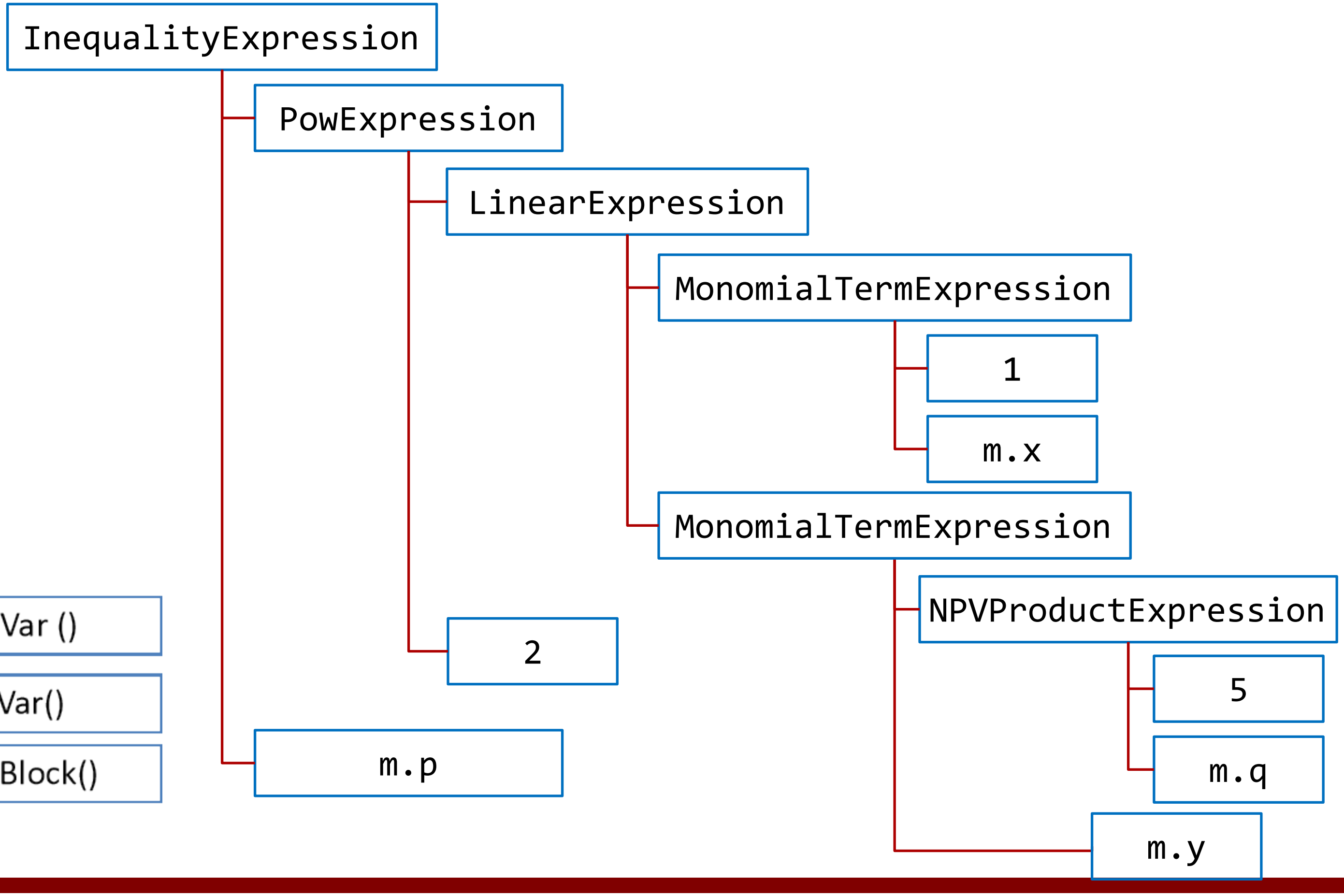
Pyomo models are trees!

- Pyomo models are trees
  - Internal nodes are *Blocks*
  - Leaf nodes are *Component containers*
    - Set, Param, Var, Constraint



- Pyomo expressions are also trees

```
Constraint(expr=(m.x + 5*m.y*m.q)**2 <= m.p)
```

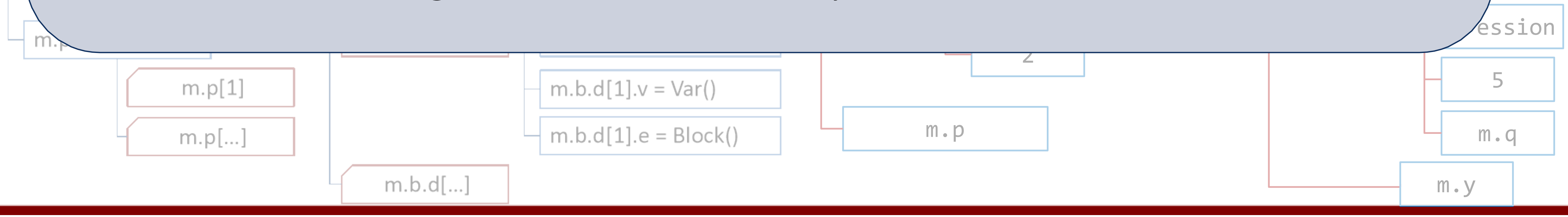


Pyomo models are trees!

- Pyomo models are trees
  - Internal nodes are *Blocks*
  - Leaf nodes are *Component containers*
- Pyomo expressions are also trees

```
Constraint(expr=(m.x + 5*m.y*m.q)**2 <= m.p)
```

- Pyomo's fundamental data structure is an expression tree … even for things like linear expressions
  - Can be slower than custom representations for special structures like (MI)LP
  - Makes model manipulation easier with fewer special cases
- The fundamental "operation" on a model is an expression walker
  - Focus on making them efficient and easy to write/extend



Standardizing expression generation/manipulation

- Generating and “walking” expression trees core to Pyomo
  - Shift to leveraging *multiple dispatch* for extensible expression generation and processing
  - Not native to the Python language, but efficiently implementable using `dict` vtables and dynamic registration

- Multiple dispatch has been integrated into
  - Numeric expression generation
  - Linear / Quadratic / AMPL expression compilers
  - LP, NL writers

- Will be included as part of upcoming refactors of
  - Logical expression generation
  - BAR, GMS writers

What do these have in common?

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

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

$$a = |x - 3|$$

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

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

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

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

Models 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.

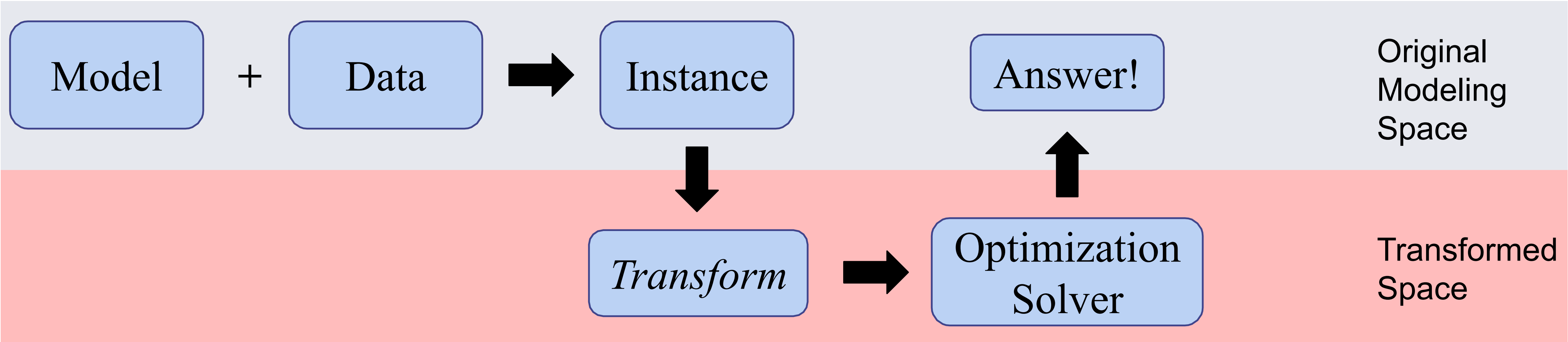
Do you speak solver?

Model + Data → Instance → Optimization Solver → Answer!

- 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?
- 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

Transformations enable more intuitive modeling



- Transformations separate the 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 Pyomo 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

A fresh take on solver interfaces

- The original solver interfaces were designed for “more than just Pyomo models”
  - Leverage an internal “meet in the middle” approach for mapping the model to the solver
  - Designed exclusively for “once through” paradigms
- 2021: introduced APPSI (the *Automatic Persistent Pyomo Solver Interface*)
  - Redesigned to efficiently support repeated (related) solves of the same model
  - Heavily leveraged compiled extensions for key operations (like model compilation)
  - Proposed several fundamental (backwards incompatible) changes to the solver API
- 2023: introduced updated problem compilers (writers)
  - Significant change to the information that needs to be passed between compilers and solvers
    - Presolve information, scaling factors, variable ordering, etc.
- 2024: took lessons learned from APPSI and new writers and developed new standard solver interface
  - Still under development, preview available in pyomo.contrib.solver
  - New solvers are available in existing API / infrastructure through a “Legacy interface wrapper”
  - Many new writer features (e.g., presolve and model scaling) are only available via the new interfaces

Revisiting model compilation

- We recently rewrote Pyomo's NL writer
- New (NL) features
  - Linear presolve:
    - Detect implicitly fixed variables
    - Variable aggregation with no fill-in
  - Model scaling
    - Efficiently scale variables / constraints after model compilation and before writing
    - Complements the "scaling transformation"
      - Same parameterization, but solver agnostic and avoids the cost of duplicating the model

Impact of presolve on DAE optimal control problem

```
This is Ipopt version 3.14.11, running with linear solver ma27.

                                                      with presolve    without presolve
Number of nonzeros in equality constraint Jacobian...:      5499      6052
Number of nonzeros in inequality constraint Jacobian.:         0         0
Number of nonzeros in Lagrangian Hessian.............:      2660      2666

Total number of variables............................:      1533      1760
Total number of equality constraints.................:      1324      1551

Number of Iterations....:                                     90       319
Objective...............:                  8.5411094197678061e-02    2.0874479958555342e-01
Total seconds in IPOPT                                     0.163     2.186

                                EXIT: Optimal Solution Found.    EXIT: Restoration Failed!
```

- New compiled representation
  - "linear standard form": $\min c^T x$ $s.t.\ Ax \leq b$ (where A, c are `scipy.sparse` arrays and b is a `numpy.ndarray`)
    - Optionally, add slack variables and compile to "$\min c^T x\ \ s.t.\ Ax = b$"
    - Optionally, convert all variables to nonnegative domains

Exploring new AML ideas with a focus on performance

- Coek: A C++ Optimization Expression Kernel
  - Express optimization problems in C++
  - Integrates CppAD and ASL to compute derivatives for nonlinear problems
  - Development is being driven by targeted experiments and demonstrations, often with runtime performance as a major driver
- Poek: A performant Python library used to formulate and solve optimization problems
  - A light-weight Python wrapper for Coek
  - Can express large optimization problems in Python with modest overhead
- Pyomo_coek: Pyomo hybrids that leverage Coek to accelerate common operations
- Smoek: A new Python-based modeling language that explicitly exploits compact expressions
  - Designed to support different backends (e.g. code generation for Coek or Pyomo models)

Some of the Pyomo extensions under active development

- CP (E. Johnson)
  - Constraint programming abstractions and solver interfaces
- DoE (J. Liu, A. Dowling)
  - Model-based design of experiments
  - Workshop material from ESCAPE/PSE 2024: https://dowlinglab.github.io/pyomo-doe/Readme.html
- Incidence analysis (R. Parker)
  - Structural / numeric analysis of nonlinear programs
  - Core part of IDAES Diagnostics: https://idaes-pse.readthedocs.io/en/stable/explanations/model_diagnostics/index.html
- Latex Printer (C. Karcher)
  - Print Pyomo models to a LaTeX compatible format
- MindtPy (Z. Peng, D. Bernal)
  - Decomposition strategies for MINLPs, including Duran & Grossmann outer approximation algorithm
- Piecewise (E. Johnson)
  - Modeling with and reformulating multivariate piecewise linear functions
- PyROS (J. Sherman, N. Isenberg, C. Gounaris)
  - Robust Optimization Solver (generalized robust cutting set algorithm)

Pyomo Param component

- Pyomo supports a parameter component (`Param`)
  - Keeps data documented on the model
  - Allows for validation of data, default values, and changes in data **without needing to rebuild the 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,
                               mutable = True )
```

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

- 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*

Units handling in Pyomo

- Units can be assigned to Var, Param, and ExternalFunction Pyomo components
- Units can also be used directly in expressions (e.g., defining constraints)
- Implemented using the pint Python package

```
import pyomo.environ as pyo
from pyomo.environ import units as u
from pyomo.util.check_units import assert_units_consistent, identify_inconsistent_units

model = pyo.ConcreteModel()
model.acc = pyo.Var(initialize=5.0, units=u.m/u.s**2)
model.obj = pyo.Objective(expr=(model.acc - 9.81*u.m/u.s**2)**2)

assert_units_consistent(model.obj) # raise exception if units invalid on obj
assert_units_consistent(model) # raise exception if units invalid anywhere on the model
print(u.get_units(model.obj.expr)) # print the units on the objective, m**2/s**4
```

Wrapping up

- Things we learned from JuMP
  - Multiple dispatch to accelerate operator overloading
  - Direct memory solver interfaces
  - Templatization and working with an “abstract expression tree”
  - Consistent dual convention in the modeling language
- Where are we going?
  - A significant rework of the online documentation
    - (targeting late summer release)
  - Complete redesign of `parmest` and `pyomo.DoE`
    - Move both tools to common abstractions and interfaces
  - Porting advancements from LP, NL writers to GAMS, BAR writers
    - (10-50% faster)
  - Template-aware writers
    - Avoid expanding most constraint expressions (speed + memory improvements)

Comparing Pyomo 6.5.0 and 6.7.1	
Component	Improvement
Model creation	3%
LP writer	28%
NL writer	18%
BAR writer	19%
GAMS writer	4%

Thank you!

- For more information:
  - www.pyomo.org
  - http://github.com/Pyomo/pyomo
  - pyomo-forum@googlegroups.com
- Acknowledgements

  - This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, 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-NA0003525
  - IDAES gratefully acknowledges support from the U.S. Department of Energy, Office of Fossil Energy and Carbon Management through the Simulation-Based Engineering/Crosscutting Research Program
  - PROMMIS gratefully acknowledges support from the U.S. DOE's Fossil Energy and Carbon Management Office of Resource Sustainability.
  - DISPATCHES gratefully acknowledges support from the Grid Modernization Laboratory Consortium through FECM, NE, & EERE/HFTO

Backup

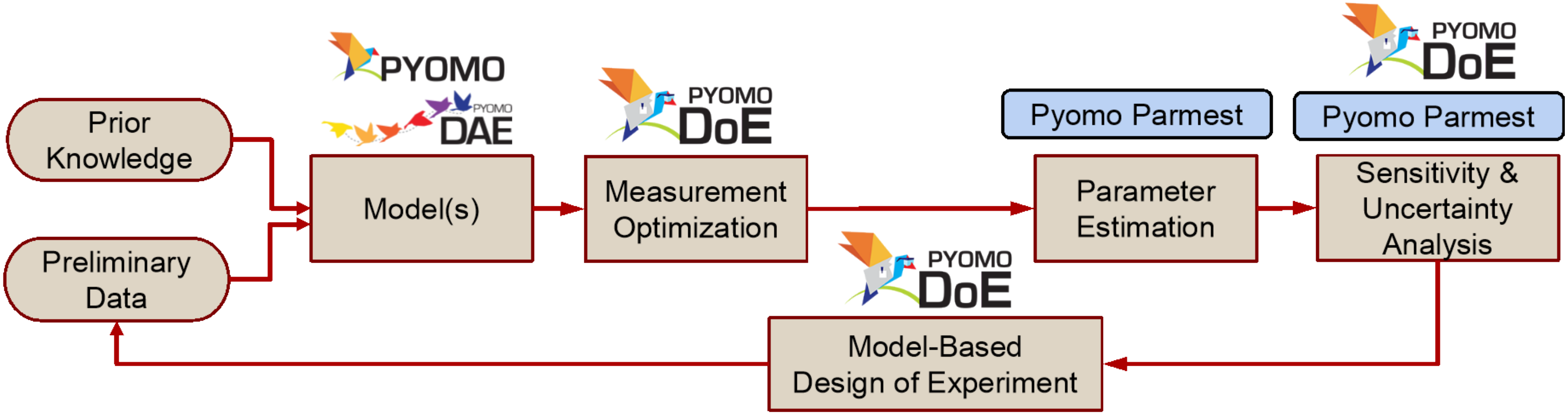
CP: Solving Logical / Constraint Programming models

- 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

DoE: Model-based Design of Experiments

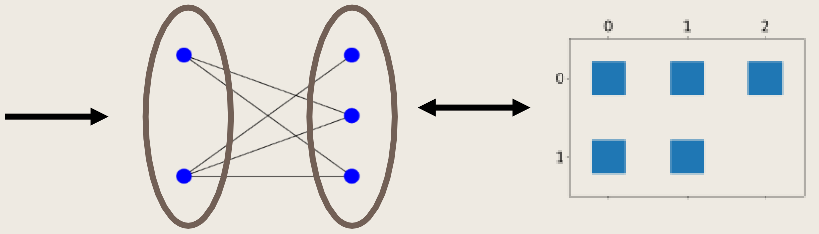
- Model-based Design of Experiments in Pyomo (J. Wang, A. Dowling)
  - Given:
    - Pyomo model, nominal parameter values, experimental design variables, covariance matrix
  - Compute Fisher information matrix
  - Perform exploratory analysis (enumeration)
  - Compute A- or D-optimal experimental design (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

$$x^2 + y^2 + z^2 = 1$$
$$x + 2y = 3$$

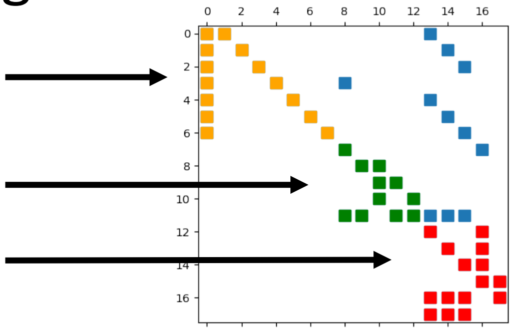


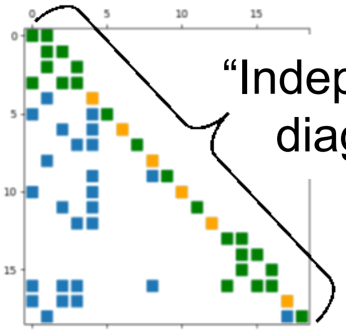
- Result: Block triangularization and Dulmage-Mendelsohn tell us whether and why systems are singular

Underconstrained

Well-constrained

Overconstrained





"Independent" diagonal blocks

[1] Tasseff, Coffrin, Wächter, and Laird. https://arxiv.org/pdf/1909.08104.pdf

Piecewise-linear approximations of multivariate functions

- Vielma et al. [2015] presents a collection of formulations for multivariate piecewise linear representations
  - `pyomo.contrib.piecewise` generalizes these formulations through four GDP representations and subsequent application of standard GDP → MIP transformations.

	BigM	Multiple BigM	Hull	Ad-hoc
Inner Representation GDP			Disaggregated Convex Combination Model	
Reduced-Space Inner Representation GDP		Convex Combination Model		
Outer Representation 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)

Generation of LaTeX from Pyomo models

- One of the most oft-requested features is the ability convert a Pyomo model into (reasonable) LaTeX

[3]:
```
from pyomo.contrib.latex_printer import latex_printer
tex = latex_printer(model)
print(tex)
```

```
\begin{align}
    & \min
    & & \sum_{ i \in Locations  } \sum_{ j \in Customers  } cost_{i,j} serve\_customer\_from\_locat
ion_{i,j} & \label{obj:M1_obj} \\
    & \text{s.t.}
    & & \sum_{ i \in Locations  } serve\_customer\_from\_location_{i,j} = 1 &  \qquad \forall j \in
Customers \label{con:M1_single_x} \\
    &&& serve\_customer\_from\_location_{i,j} \leq select\_location_{i} &  \qquad \forall i,j \in L
ocations \times Customers \label{con:M1_bound_y} \\
    &&& \sum_{ i \in Locations  } select\_location_{i} = P & \label{con:M1_num_facilities} \\
    & \text{w.b.}
    & & 0.0 \leq serve\_customer\_from\_location \leq 1.0 & \qquad \in \mathbb{R} \label{con:M1_ser
ve_customer_from_location_bound} \\
    &&& select\_location & \qquad \in \left\{ 0 , 1 \right \} \label{con:M1_select_location_bound}
\end{align}
```

[4]:
```
import IPython
IPython.display.Math(tex)
```

[4]:

$$\begin{align}
& \min & & \sum_{i \in Locations} \sum_{j \in Customers} cost_{i,j} serve_customer_from_location_{i,j} & \\
& \text{s.t.} & & \sum_{i \in Locations} serve_customer_from_location_{i,j} = 1 & \qquad \forall j \in Customers \\
&&& serve_customer_from_location_{i,j} \leq select_location_{i} & \qquad \forall i,j \in Locations \times Customers \\
&&& \sum_{i \in Locations} select_location_{i} = P & \\
& \text{w.b.} & & 0.0 \leq serve_customer_from_location \leq 1.0 & \qquad \in \mathbb{R} \\
&&& select_location & \qquad \in \left\{ 0 , 1 \right\}
\end{align}$$

```
import random
from pyomo.environ import *

model = ConcreteModel(name="M1")
model.N = Param(initialize=6, within=PositiveIntegers)
model.M = Param(initialize=6, within=PositiveIntegers)
model.P = Param(initialize=3, within=RangeSet(1, model.N), mutable=True)

model.Locations = RangeSet(1, model.N)
model.Customers = RangeSet(1, model.M)

model.cost = Param(
    model.Locations, model.Customers,
    initialize=lambda n, m, model: random.uniform(1.0, 2.0), within=Reals,
)
model.serve_customer_from_location = Var(
    model.Locations, model.Customers, bounds=(0.0, 1.0)
)
model.select_location = Var(model.Locations, within=Binary)

@model.Objective()
def obj(model):
    return sum(
        model.cost[n, m] * model.serve_customer_from_location[n, m]
        for n in model.Locations for m in model.Customers
    )

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

@model.Constraint(model.Locations, model.Customers)
def bound_y(model, n, m):
    return model.serve_customer_from_location[n, m] <= model.select_location[n]

@model.Constraint()
def num_facilities(model):
    return sum(model.select_location[n] for n in model.Locations) == model.P
```

Generation of LaTeX from Pyomo models

- One of the most oft-requested features is the ability to convert a Pyomo model into (reasonable) LaTeX
- We provide a level of customization
  - E.g., reducing meaningful variable names into "journal-friendly notation"
- Disclaimers
  - This is *experimental* and under development
    - You are likely to run into bugs
  - "Compact" model representation requires that your model constraints be "templatizable"
    - No logic / conditions within rules
  - Not all of Pyomo is supported yet
    - Blocks, DAE, and GDP are still in progress

[5]:
```
from pyomo.contrib.latex_printer import latex_printer
lcm = ComponentMap()
lcm[model.Locations] = ['L', ['n']]
lcm[model.Customers] = ['C', ['m']]
lcm[model.cost] = 'd'
lcm[model.serve_customer_from_location] = 'x'
lcm[model.select_location] = 'y'
tex = latex_printer(model, latex_component_map=lcm)
```

[6]:
```
import IPython
IPython.display.Math(tex)
```

[6]:

$$
\begin{aligned}
& \min & & \sum_{n \in L} \sum_{m \in C} d_{n,m} x_{n,m} & & \\
& \text{s.t.} & & \sum_{n \in L} x_{n,m} = 1 & & \forall m \in C \\
& & & x_{n,m} \leq y_n & & \forall n,m \in L \times C \\
& & & \sum_{n \in L} y_n = P & & \\
& \text{w.b.} & & 0.0 \leq x \leq 1.0 & & \in \mathbb{R} \\
& & & y & & \in \{0,1\}
\end{aligned}
$$