



A Framework for Modeling and Optimizing Dynamic Systems Under Uncertainty

Bethany Nicholson

John D. Siirola

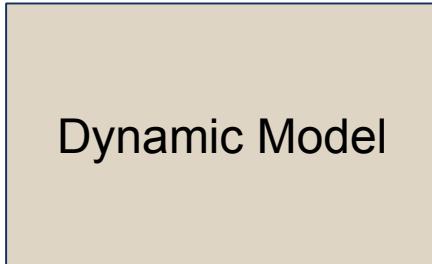
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM

INFORMS Annual Meeting October 22-25, 2017

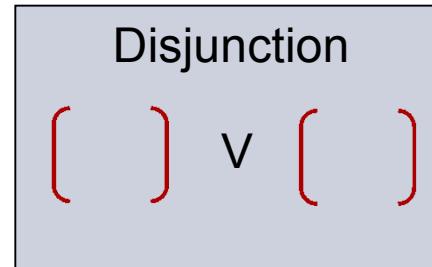


Implement this...

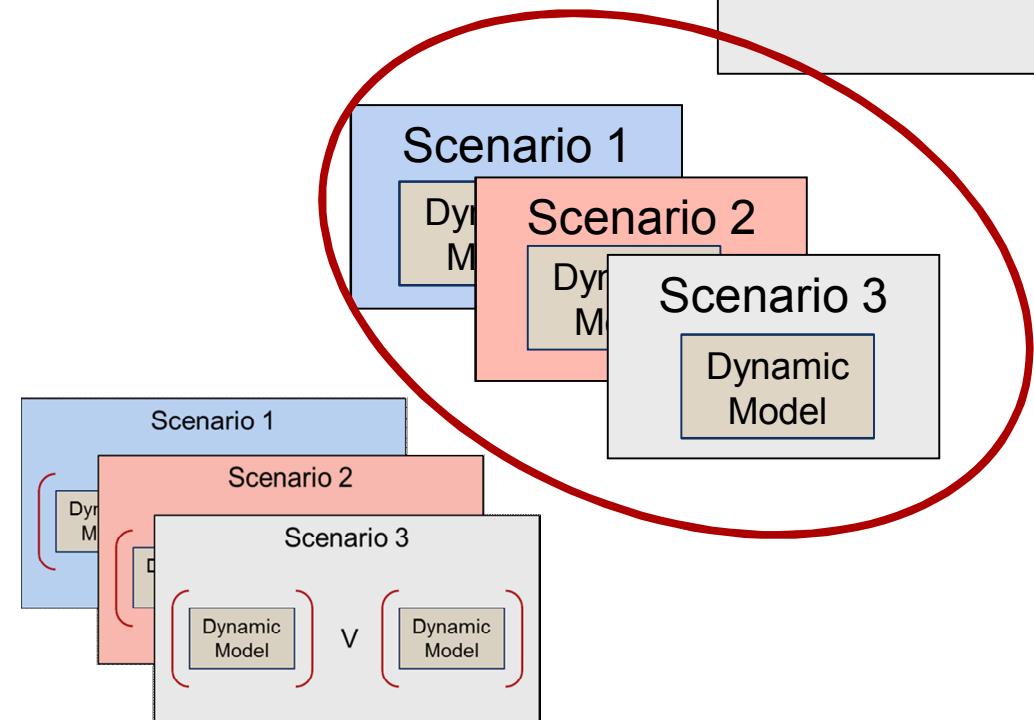
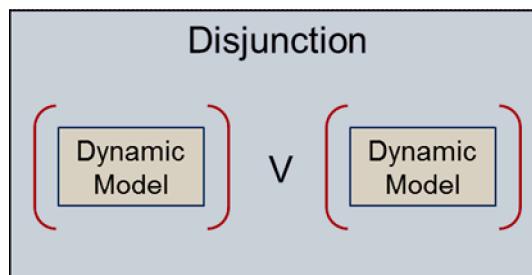
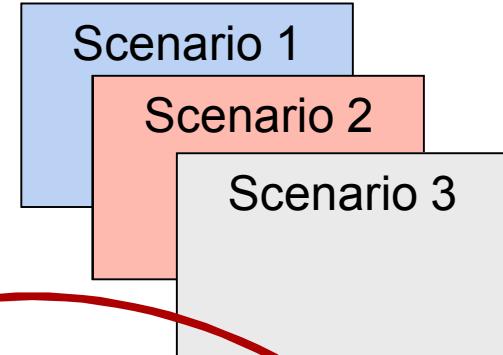
Dynamic Optimization



Generalized Disjunctive Programming



Stochastic Programming



Why capture model structure?

- Challenges with a flat representation
 - manual reformulation is required to write a ‘solvable’ model
 - difficult to reverse engineer the intent or goal of the original problem
 - tedious to experiment with alternative model reformulations
- Benefits to explicitly capturing structure
 - models are formulated in a more natural, intuitive form
 - fewer coding mistakes
 - separates model specification from the solution approach
 - easy to experiment with different model reformulations
 - encourages general implementations of common solution approaches

Software platform

- Pyomo: Python Optimization Modeling Objects
- Formulate optimization models within Python



```
from pyomo.environ import *

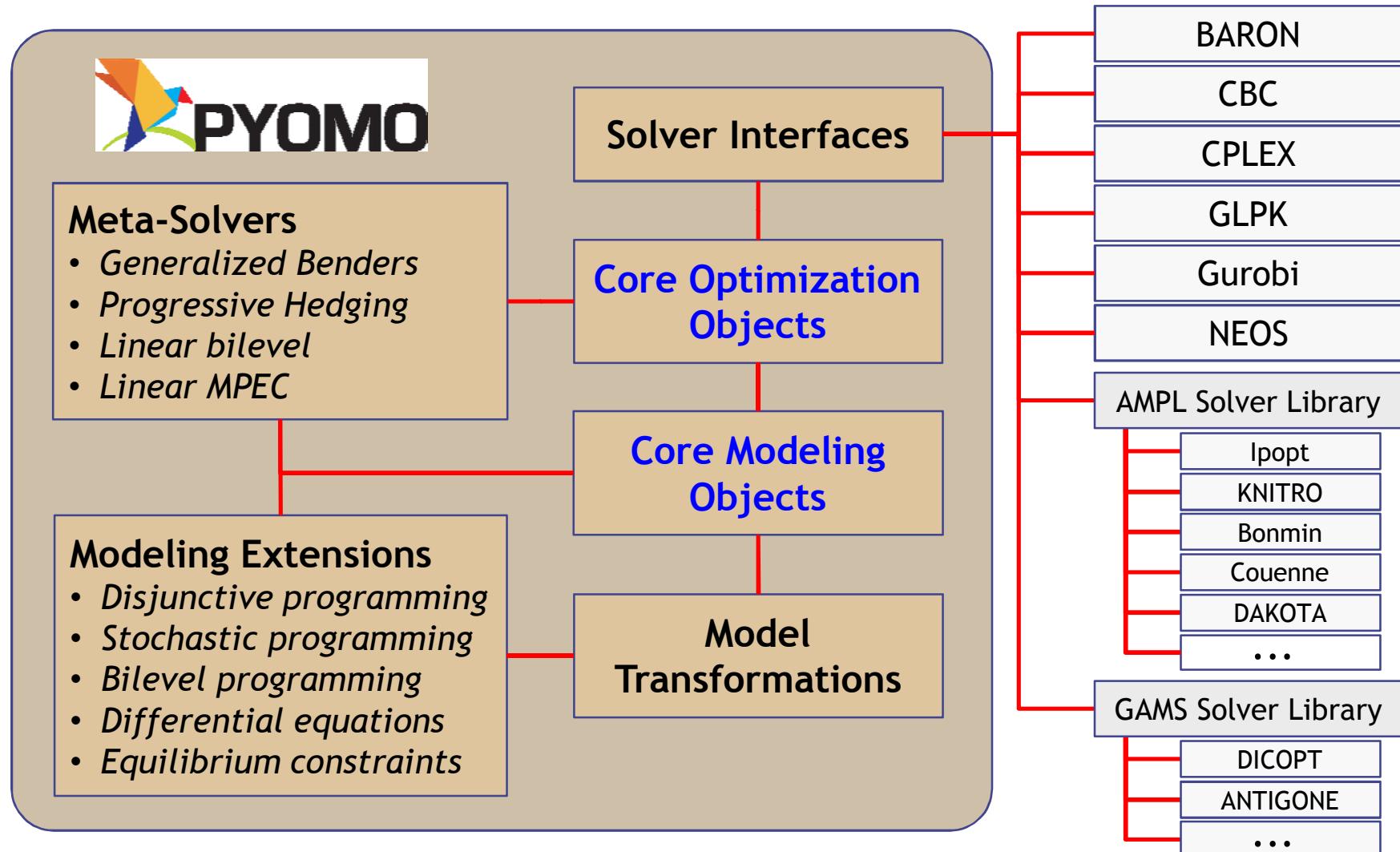
m = ConcreteModel()

m.x1 = Var()
m.x2 = Var(bounds=(-1,1))
m.x3 = Var(bounds=(1,2))

m.obj = Objective(sense = minimize,
    expr = m.x1**2 + (m.x2*m.x3)**4 + m.x1*m.x3
    + m.x2 + m.x2*sin(m.x1+m.x3) )
```

- Utilize high-level programming language to write scripts and manipulate model objects
- Leverage third-party Python libraries
 - e.g. SciPy, NumPy, Matplotlib, Pandas

Pyomo at a Glance



Solving dynamic optimization problems

$$\min \int_{t_0}^{t_F} (\phi(x, u)) dt + \phi(x(t_F))$$

$$\text{s.t. } \begin{aligned} \frac{dx}{dt} &= g(x, u) \\ x(t_0) &= \text{constant} \end{aligned}$$

x: State variables

u: Control variables

Discretize
Model

$$\min \sum_{k=1}^{N-1} (\phi(x_k, u_k)) + \phi(x_N)$$

s.t. $x_{i+1} = f(x_i, u_i), i = 1, \dots, N-1$

$x_1 = \text{constant}$

*Approximate dynamics
using algebraic equations*

$$z^K(t) = \sum_{j=0}^K z_{ij} \ell_j(\tau), \quad \ell_j(\tau) = \prod_{\substack{k=0 \\ k \neq j}}^K \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)}, \quad t_{ij} = t_{i-1} + \tau_j h_i$$

$$\sum_{j=0}^K z_{ij} \dot{\ell}_j(\tau_k) = h_i f(z_{ik}, y_{ik}, u_{ik}, t_{ik}), \quad z_{i+1,0} = \sum_{j=0}^K \ell_j(1) z_{ij}, \quad z_f = \sum_{j=0}^K \ell_j(1) z_{Nj}, z_{1,0} = 0$$

Solving dynamic optimization problems

$$\min \int_{t_0}^{t_F} (\phi(x, u)) dt + \phi(x(t_F))$$

$$s. t. \quad \frac{dx}{dt} = g(x, u) \\ x(t_0) = \text{constant}$$

x: State variables

u: Control variables

Discretize
Model

$$\min \sum_{k=1}^{N-1} (\phi(x_k, u_k)) + \phi(x_N)$$

$$s. t. \quad x_{i+1} = f(x_i, u_i), i = 1, \dots, N-1 \\ x_1 = \text{constant}$$

*Approximate dynamics
using algebraic equations*

Automated with Pyomo.DAE

Pyomo.dae

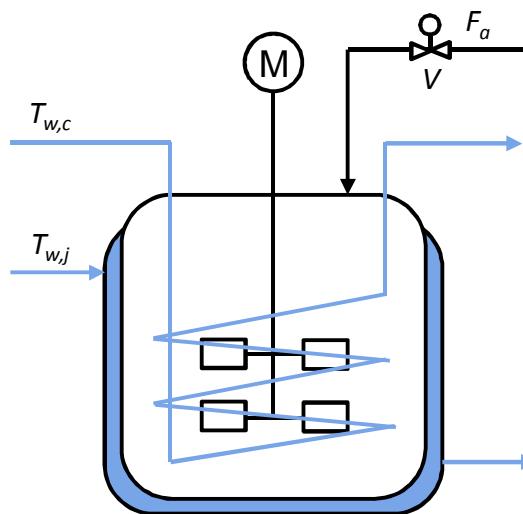
- Extend Pyomo syntax to represent:
 - Continuous domains
 - Ordinary differential equations
 - Partial differential equations
 - Systems of differential algebraic equations
 - Higher order differential equations and mixed partial derivatives
- Available discretization schemes
 - Finite difference methods (Backward/Forward/Central)
 - Collocation (Radau or Legendre roots)
- Extensible framework
 - Write general implementations of custom discretization schemes
 - Build frameworks/meta-algorithms including dynamic optimization
- Interface with numerical simulators
 - Scipy for simulating ODEs
 - CasADi for simulating ODEs and DAEs

PySP

- Framework for simplifying implementation of stochastic programming models, only requiring:
 - deterministic base model
 - scenario tree defining the problem stages and uncertain parameters
- PySP provides two primary solution strategies
 - build and solve the deterministic equivalent (extensive form)
 - Progressive Hedging
 - (plus beta implementations of others, including 2-stage Benders and an interface to DDSIP)
- Parallel infrastructure for generating and solving subproblems on parallel (distributed) computing platforms

Dynamic system under uncertainty

- Semibatch reactor^[2]



Model inputs (control variables)

F_a : Inlet flow rate of A

$T_{w,c}$: Water temperature in cooling coil

$T_{w,j}$: Water temperature in cooling jacket

$$\dot{C}_a = \frac{F_a}{V_r} - k_1 \exp\left(-\frac{E_1}{RT_r}\right) C_a$$

$$\dot{C}_b = k_1 \exp\left(-\frac{E_1}{RT_r}\right) C_a - k_2 \exp\left(-\frac{E_2}{RT_r}\right) C_b$$

$$\dot{C}_c = k_2 \exp\left(-\frac{E_2}{RT_r}\right) C_b$$

$$\dot{V}_r = \frac{F_a M_{W_a}}{\rho_r}$$

$$(\rho_r c_{p_r}) \dot{T}_r = \frac{F_a M_{W_a} c_{p_r}}{V_r} (T_f - T_r)$$

$$- k_1 \exp\left(-\frac{E_1}{RT_r}\right) C_a \Delta H_1 - k_2 \exp\left(-\frac{E_2}{RT_r}\right) C_b \Delta H_2$$

$$+ \alpha_{w,j} \frac{A_j}{V_{r,0}} (T_{w,j} - T_r) + \alpha_{w,c} \frac{A_c}{V_{r,0}} (T_{w,c} - T_r)$$

- Two case studies

- parameter estimation

- optimal control under partial system failure^[2]

Dynamic model implementation

```

from pyomo.environ import *
from pyomo.dae import *

def build_semibatch_model(data):

    m = ConcreteModel()

    # Continuous time domain
    m.t = ContinuousSet(bounds=(0,21600), initialize=m.measT)

    # Other variable/parameter/constraint declarations
    ...

    # Differential Variable
    m.Ca = Var(m.t)
    m.dCa = DerivativeVar(m.Ca)

    # Differential equation
    def _dCacon(m,t):
        if t == 0:
            return Constraint.Skip
        return m.dCa[t] == m.Fa[t]/m.Vr[t] - \
               m.k1*exp(-m.E1/(m.R*m.Tr[t]))*m.Ca[t]
    m.dCa = Constraint(m.t, rule=_dCacon)

    # Automatically apply collocation over finite elements discretization
    discretizer = TransformationFactory('dae.collocation')
    discretizer.apply_to(m, nfe=20, ncp=4)

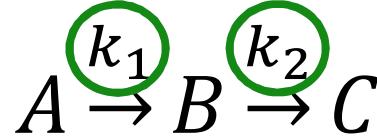
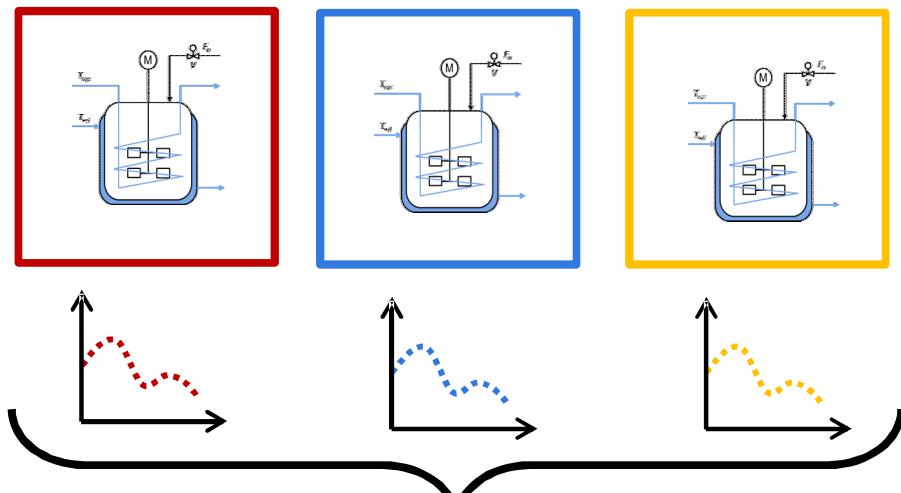
    return m
  
```

$$\dot{C}_a = \frac{F_a}{V_r} - k_1 \exp\left(-\frac{E_1}{RT_r}\right) C_a$$



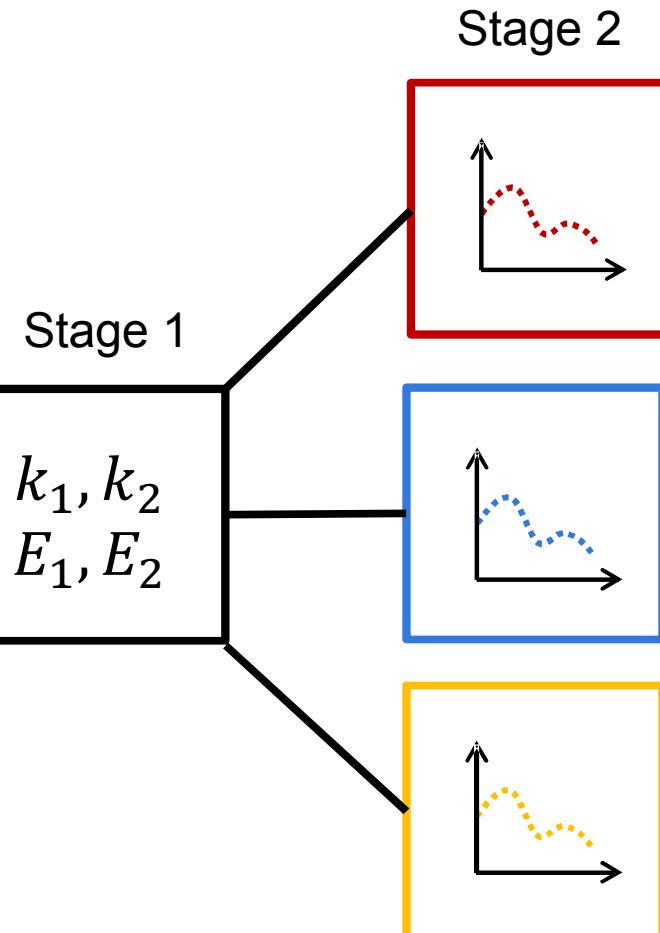
Parameter estimation

Experiment 1 Experiment 2 Experiment 3



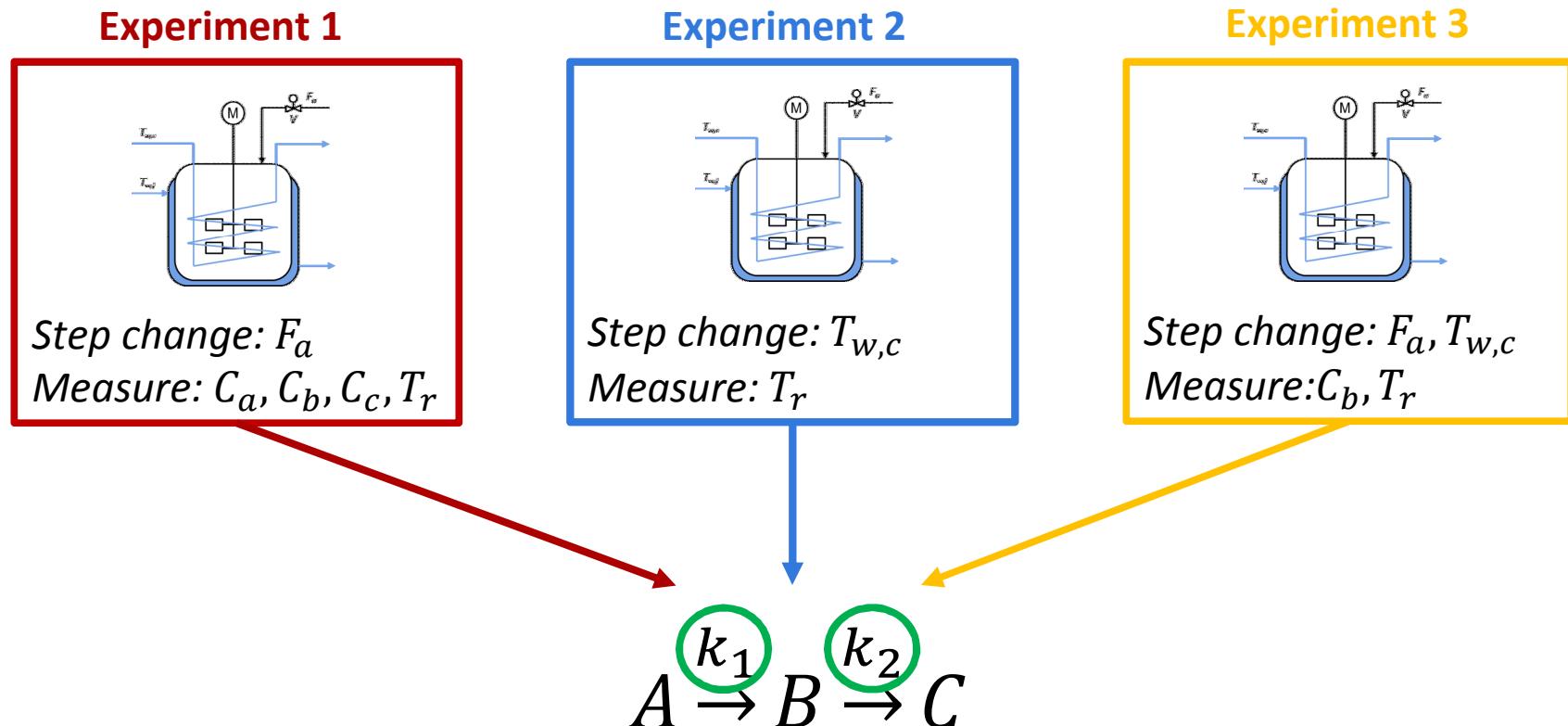
$$\min_{\{k_1, k_2, E_1, E_2\}} \sum_{exp.} (error_{meas})^2$$

s.t. semibatch model equations



Parameter estimation

- Three ***different*** experiments
 - different manipulated variables, different measured data



Stochastic structure implementation (1/2)

```
def pysp_scenario_tree_model_callback():
    from pyomo.pysp.scenariotree.tree_structure_model \
        import CreateConcreteTwoStageScenarioTreeModel

    st_model = CreateConcreteTwoStageScenarioTreeModel(scenarios)

    first_stage = st_model.Stages.first()
    second_stage = st_model.Stages.last()

    # First Stage
    st_model.StageCost[first_stage] = 'FirstStageCost'
    st_model.StageVariables[first_stage].add('k1')
    st_model.StageVariables[first_stage].add('k2')
    st_model.StageVariables[first_stage].add('E1')
    st_model.StageVariables[first_stage].add('E2')

    # Second Stage
    st_model.StageCost[second_stage] = 'SecondStageCost'

    return st_model
```

Stochastic structure implementation (2/2)



```
def pysp_instance_creation_callback(scenario_name, node_names):
    experiment = int(scenario_name.replace('Scenario',''))

    # Experiments with measurement noise
    explist = [1,2,3] # Different step changes in control inputs

    experiment = explist[experiment-1]
    instance = generate_semibatch_model_paramest(experiment)

    return instance
```

- Create and solve extensive form

```
runef --solve --solver ipopt --output-solver-log -m semibatch.py
```

- Solve using progressive hedging

```
runph --solver ipopt --output-solver-log -m semibatch.py --default-rho=.25
```

Semibatch parameter estimation results

- Extensive form results

	k_1 (1/s)	k_2 (1/s)	E_1 (kJ/kmol)	E_2 (kJ/kmol)	Objective
Actual	15.01	85.01	30,000	40,000	-
All Meas.	16.84	81.19	30,322	39,861	2.147
Missing Meas.	20.69	77.42	30,850	39,697	24.976

all: 47 IPOPT iterations, 2674 variables, 2670 constraints, 1.08 s to run

missing: 33 IPOPT iterations, 2674 variables, 2670 constraints, 0.87 s to run

- Progressive Hedging results*

	k_1 (1/s)	k_2 (1/s)	E_1 (kJ/kmol)	E_2 (kJ/kmol)	Objective
Actual	15.01	85.01	30,000	40,000	-
All Meas.	15.72	30.59	30,146	37,017	3.170
Missing Meas.	24.38	69.49	31,302	39,400	25.051

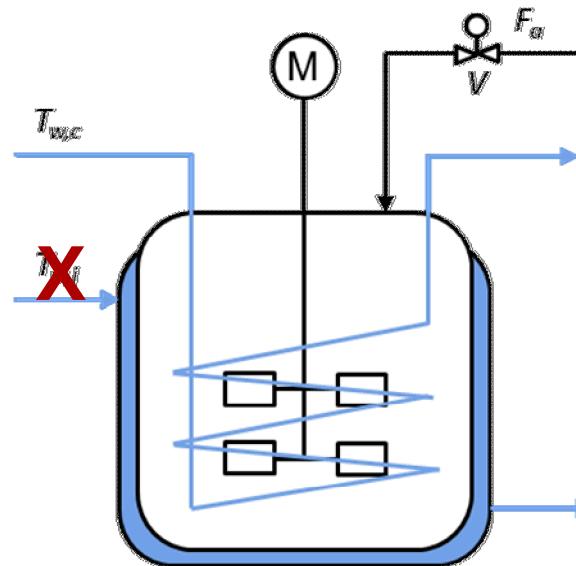
all: 50 PH iterations, 15.08 s to run

missing: 35 PH iterations, 11.05 s to run

IPOPT subproblem size: 890 variables, 886 constraints, ~7 iterations

Optimal control

- Find the nominal control profiles such that the batch can be ‘saved’ given a partial cooling system failure at any point during the batch time^[2]



$$\left. \begin{aligned} T_{w,j}(t) &= T_{w,c}(t) & t \leq t_{fail} \\ (\rho_w C_{p_w} V_j) \dot{T}_{w,j} &= \alpha_{w,js} A_j \frac{V_r}{V_{r,0}} (T_{w,j} - T_r) & t > t_{fail} \end{aligned} \right\}$$

Optimal control scenarios

Nonanticipativity Constraints

Nominal

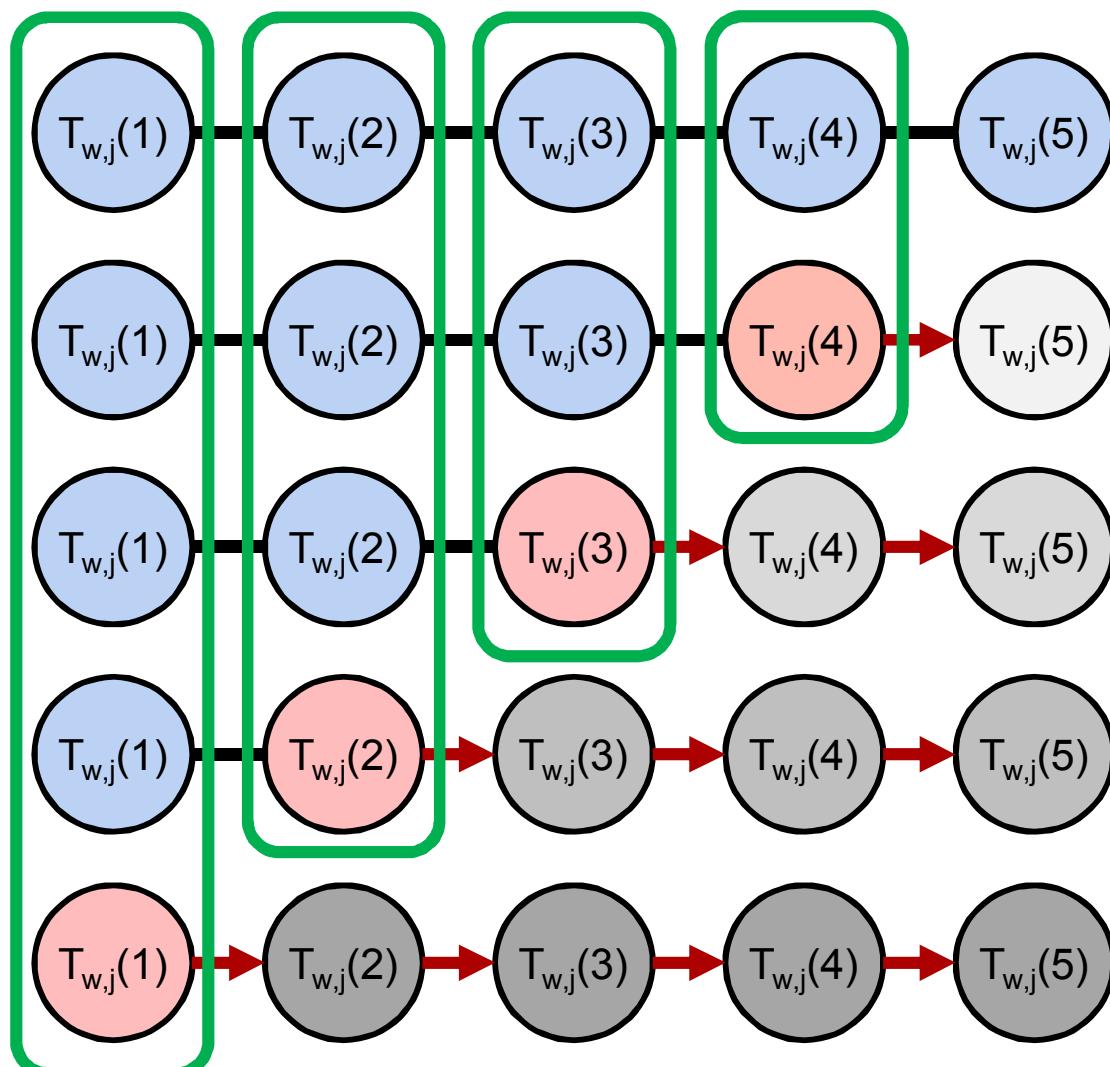
$t_{fail} = 4$

$t_{fail} = 3$

$t_{fail} = 2$

$t_{fail} = 1$

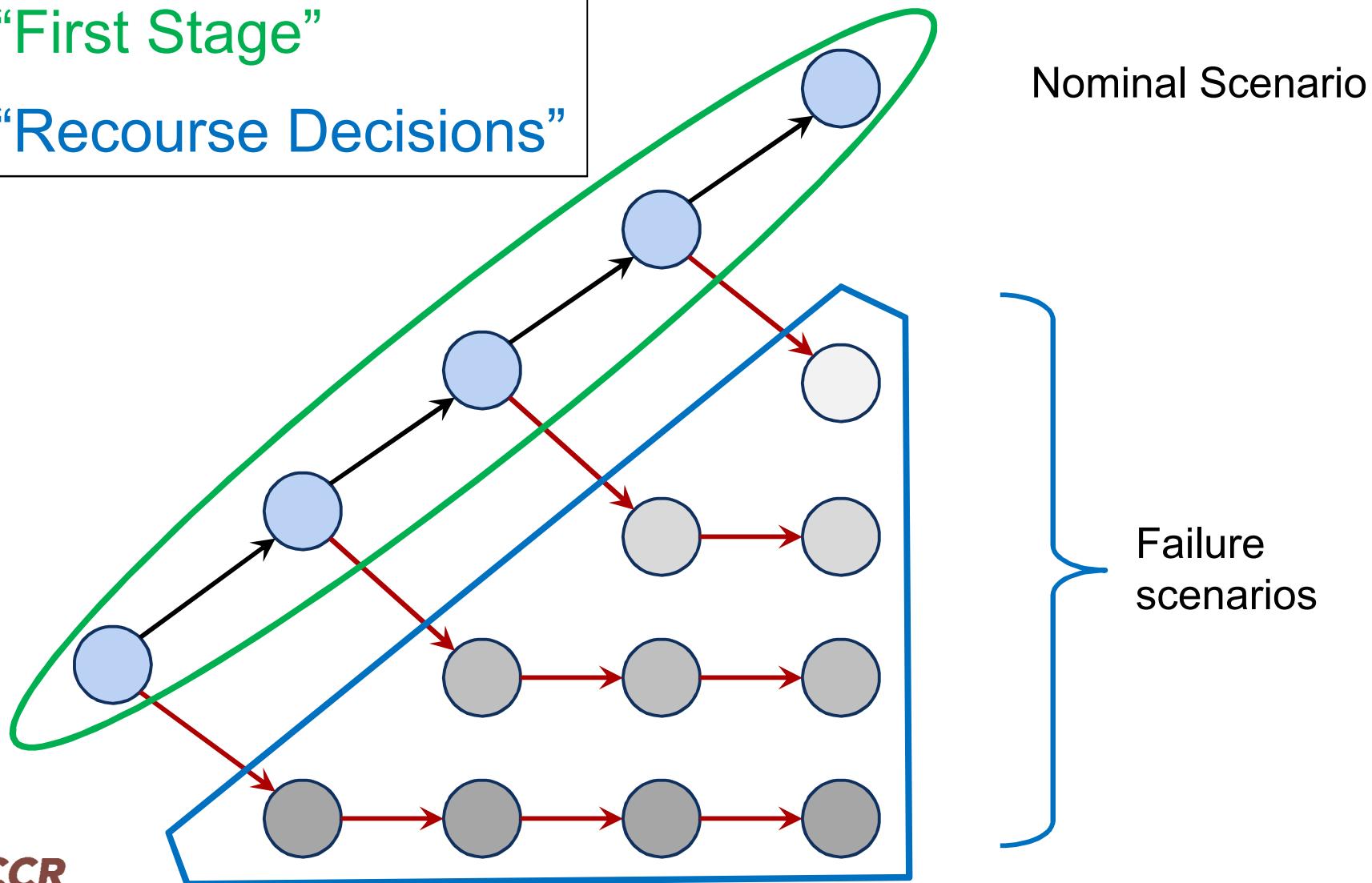
Time



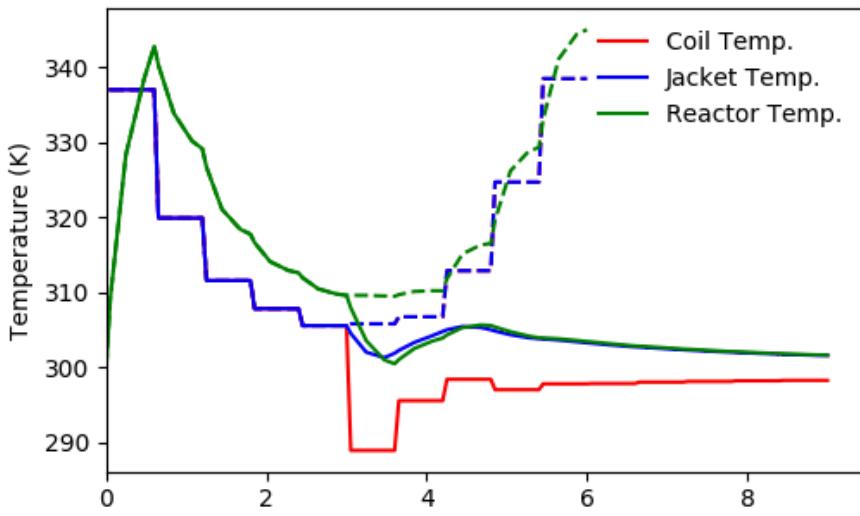
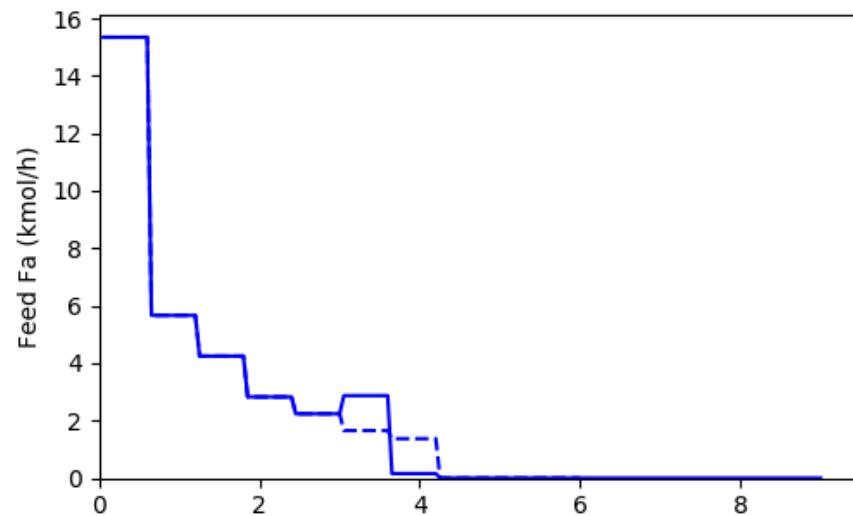
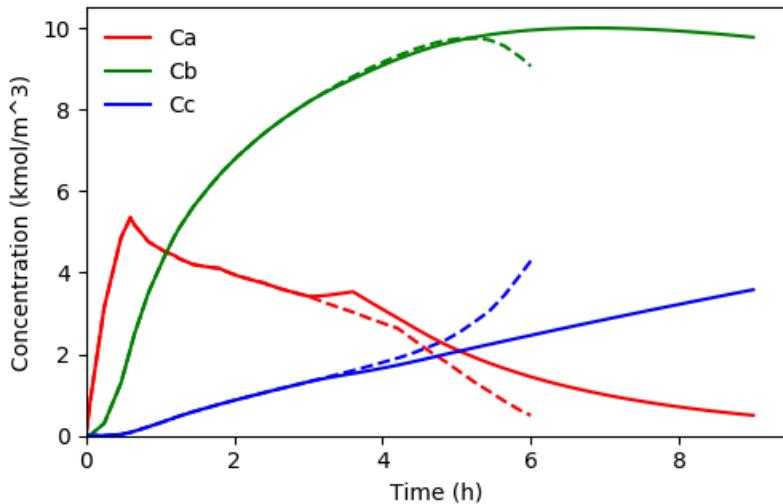
n-Stage SP as a 2-stage problem

“First Stage”

“Recourse Decisions”



Optimal control: $t_{fail} = 3$ h



Optimal control implementation



- 15 hours (including debugging)
- 300 lines of code
 - (60%) Deterministic dynamic model specification
 - (2%) Discretization
 - (18%) Stochastic problem formulation
 - (20%) Result plotting

```

# Differential Equations in the model
def _dcacon(t):
    # t if t == 0:
    # return Constraint.Skip
    return n._dcat[t] == n._fa1(t).R.Vr[t] - n._k1*exp(-n._El/(n.Rm.Tr(t)))*n._ca[t]
n._dcacon = Constraint(n.time, rule=_dcacon)

def _dccon(t):
    # t if t == 0:
    # return Constraint.Skip
    return n._dcat[t] == n._k2*exp(-n._El/(n.Rm.Tr(t)))*n._ca[t] - \
        n._k2*exp(-n._El/(n.Rm.Tr(t)))*n._cb[t]
n._dccon = Constraint(n.time, rule=_dccon)

def _drccon(t):
    # t if t == 0:
    # return Constraint.Skip
    return n._dcat[t] == n._k2*exp(-n._El/(n.Rm.Tr(t)))*n._cb[t]
n._drccon = Constraint(n.time, rule=_drccon)

def _drcon(t):
    # t if t == 0:
    # return Constraint.Skip
    return n._dcat[t] == n._fa1(t).R.mn.rhor
n._drcon = Constraint(n.time, rule=_drcon)

def _drfcon(t):
    # t if t == 0:
    # return Constraint.Skip
    return n._dcat[t] == n._fa1(t).R.mn.rhor
n._drfcon = Constraint(n.time, rule=_drfcon)

def _drcfcon(t):
    # t if t == 0:
    # return Constraint.Skip
    return n._dcat[t] == n._fa1(t).R.mn.rhor
n._drcfcon = Constraint(n.time, rule=_drcfcon)

def _singelecooling(t):
    # Apply this constraint at time fail to set initial condition for Tj diff eq
    if value(n.time.fail) == 0 or t <= value(n.time.fail):
        return n.Tct[t] == n.j1[t]
    else:
        return Constraint.Skip
n._singelecooling = Constraint(n.time, rule=_singelecooling)

def _addTempcon(n,t):
    return (n.Tad[t] - n.Tct[t]) * n.Rm.ca.prm.R.Vr[t] + \
        n.Rm.ca.prm.R.Vr[t] * n.Tad[t] == 1
    n.Vr[t] * (1 - (n.deltaH1 * n.deltaH2) * n.ca[t] + (n.deltaH2) * n.cb[t])
n._addTempcon = Constraint(n.time, rule=_addTempcon)

def _djcon(t):
    if value(n.time.fail) == 0 or t <= value(n.time.fail):
        return Constraint.Skip
    else:
        return n.Rm.ca.prm.R.Vr[n.Dj[t]] == \
            n.alpha[n.fail]*n.Aj[n.Vr[t]]*n.Vr[n.Dj[t].m.Tr[t]]
n._djcon = Constraint(n.time, rule=_djcon)

def _dcufacon(n,t):
    if t == n.time.last:
        return Constraint.Skip
    return n._dcufat[t] == n._fa1(t)
n._dcufacon = Constraint(n.time, rule=_dcufacon)

# Bound on cumulative Ra
n._cumRaInit = Constraint(expr=expone.cufFa(n.time.last)) == 20

# Bound on final Ca
B = 2500000.0/(2017*expone.Ca(n.time.last)) < 0.5

```

```

# Initial Conditions
def _initial_conditions():
    n = Cm(n.time, firstt) == n.C0
    yield n.Cm(n.time, firstt) == n.C0
    yield n.Cm(n.time, firstt) == n.C0
    yield n.Vr(n.time, firstt) == n.V0
    yield n.Vr(n.time, firstt) == n.V0
    yield n.cumFa(n.time, firstt) == 0
    n.initon = ConstraintList(rule=_initon)

# Helper constraints for enforcing nonanticipativity constraints
def Tc_nonanticon(n):
    if n.m.all fail times == 0 or t <= value(n.time_fail):
        return n.Tc_noncon[1] == n.Tc[1]
    return Constraint.Skip
n.Tc_nonanticon = Constraint(n.m.all fail times, rule=Tc_nonanticon)

def Fa_nonanticon(n):
    if n.m.all fail times == 0 or t <= value(n.time_fail):
        return n.Fa_noncon[1] == n.Fa[1]
    return Constraint.Skip
n.Fa_nonanticon = Constraint(n.m.all fail times, rule=Fa_nonanticon)

# Stage specific cost computations
def ComputeFirstStageCost_rule(model):
    return 0
n.FirstStageCost = Expression(rule=ComputeFirstStageCost_rule)

def ComputeSecondStageCost_rule(model):
    return -n.Cb(t=TimeList.lastt)
n.SecondStageCost = Expression(rule=ComputeSecondStageCost_rule)

def total_cost_rule():
    model.TotalCost = model.FirstStageCost + model.SecondStageCost
    n.Total_Cost_Ojective = Objective(rule=total_cost_rule, sense=minimize)

# Discretize model. Number of finite elements depends on the cooling failure t
nunite = n.time.lastt/1000
disc = TransformationFactory("dae.collocation")
disc.apply_to(model, n.unite, ncp=2)
disc.set_max_pointpoints(1, varname=Fa, ncp=1, contset=n.time)
disc.reduce_collocation_pointpoints(varname=Tc, ncp=1, contset=n.time)
return n

```

```

# Number of scenarios
scenarios = 10

def pypm_scenario_tree(model, callback):
    from pyomo.pysp.scenarios.tree.structure_tree import CreateConcreteTreeWithStageScenarioTreeModel
    import CreateConcreteTreeWithStageScenarioTreeModel

    st_model = CreateConcreteTreeWithStageScenarioTreeModel(scenarios)

    first_stage = st_model.Stages.First()
    second_stage = st_model.Stages.Last()

    # First Stage
    st_model.StageCost[first_stage] = "FirstStageCost"
    st_model.StageVariables[first_stage].add("Fa_renant[1]")
    st_model.StageVariables[first_stage].add("Tc_renant[1]")

    # Second Stage
    st_model.StageCost[second_stage] = "SecondStageCost"

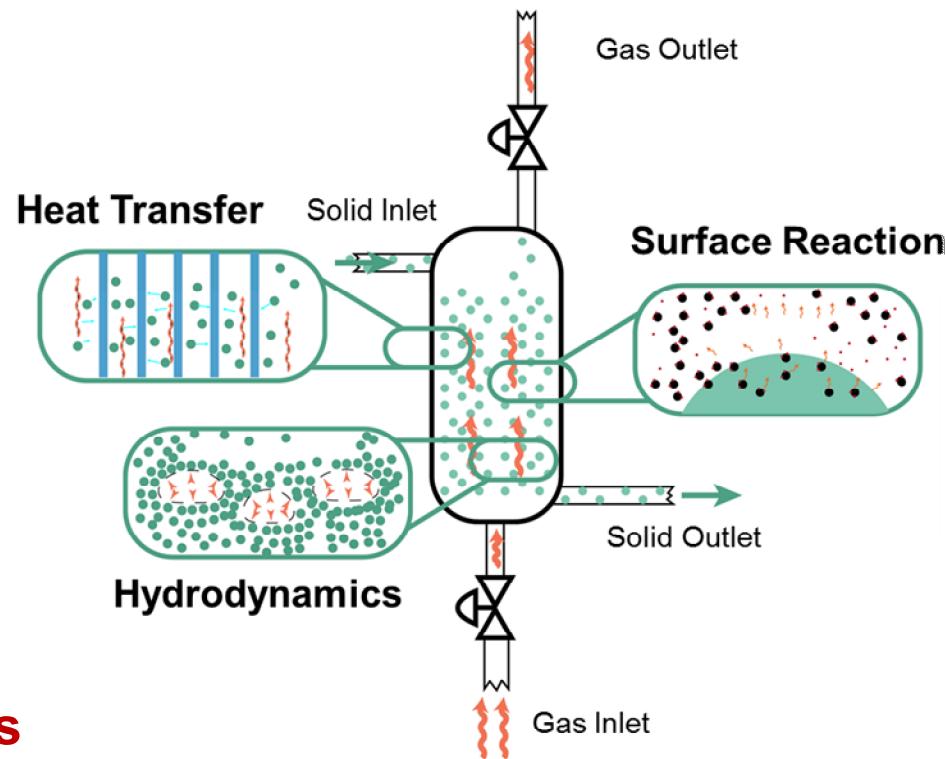
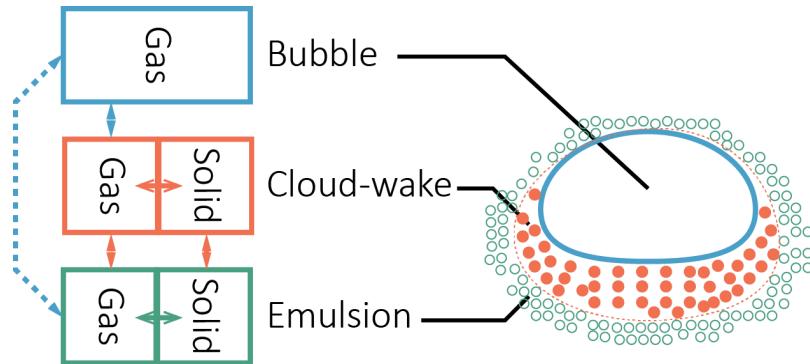
    return st_model

def pypm_instance_creation_callback(scenario_name, node_index):
    fail timestep = int(scenario_name.replace("stage", ""))
    instance = generate_seniorbath_model(fail timestep, 1)

```

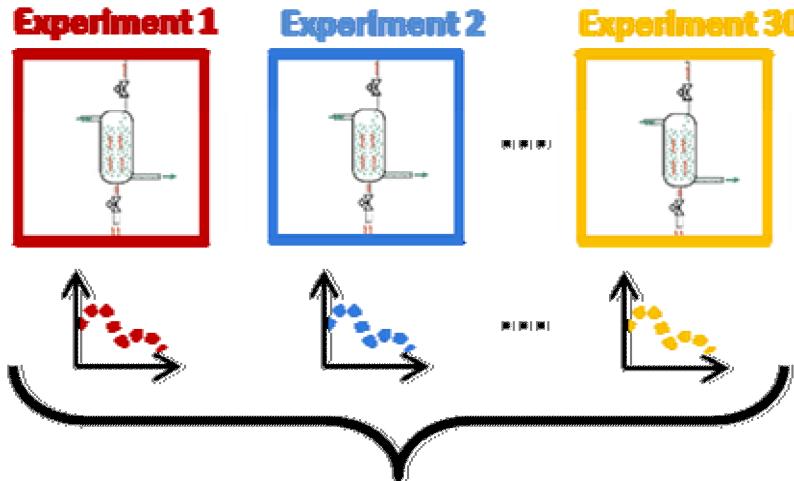
Bubbling Fluidized Bed (BFB) Model

- Gas-solid, 3 region model
(Lee and Miller, 2013, Ind. Eng. Chem. Res.)



- Modeled using system of **partial differential algebraic equations** (PDAEs)

BFB Parameter Estimation (1/2)

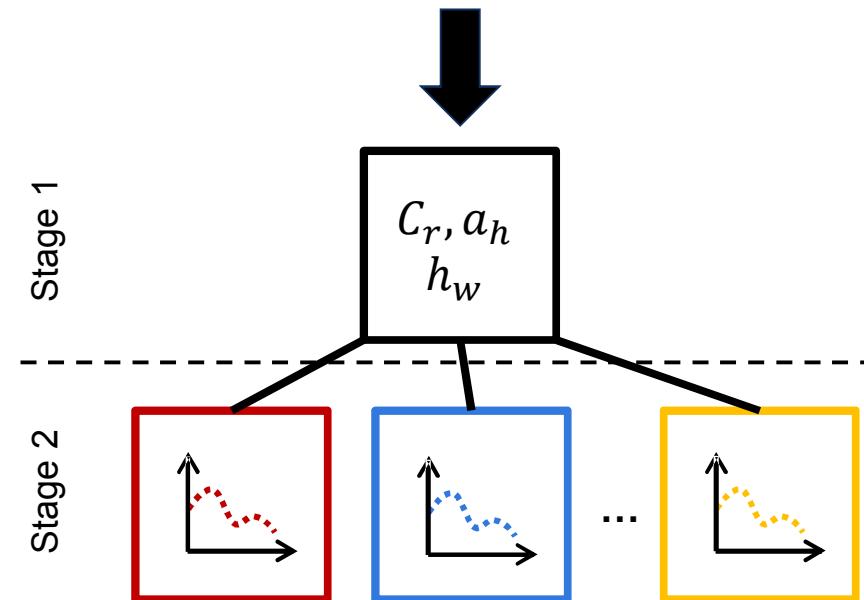


Heat Exchanger Model Parameters

a_h Empirical factor for tube model

$$\min_{\{C_r, a_h, h_w\}} \sum_{exp.} (error_{meas})^2$$

s.t. *BFB model equations*



BFB Parameter Estimation (2/2)

- Solve using progressive hedging in parallel

```
mpirun -np 1 pyomo_ns : -np 1 dispatch_srvr : -np 30 phsolverserver : \
  -np 1 runph --solver-manager=phpyro --shutdown-pyro \
  -m bfb_paramest.py --solver=ipopt --default-rho=0.25
```

	C_r	a_h	h_w	Solve Time (s)
Actual	1.0	0.8	1500.0	-
Extensive Form	1.016	0.51	1450.35	604.45
Progressive Hedging (15 proc)	0.9824	0.7850	1501.74	610.98
Progressive Hedging (30 proc)	0.9824	0.7850	1501.74	459.10

Summary

- Explicitly capturing high-level structure leads to significantly easier, faster, and more flexible implementations
- Pyomo provides high-level modeling constructs that can be easily combined to solve complex, structured optimization problems. (www.pyomo.org)

On-going pyomo.dae work

- Interface to DAE simulators
- Shooting methods for dynamic optimization

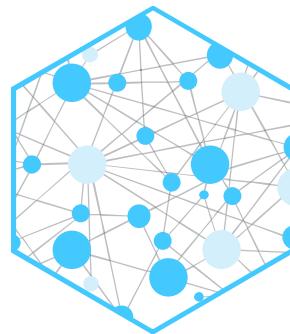
New challenges and open questions

- general implementations of meta-solvers to exploit layered/nested structure
- scalability of these techniques

Questions?

■ Acknowledgements

- This work was conducted as part of the Institute for the Design of Advanced Energy Systems (IDAES) with funding from the Office of Fossil Energy, Cross-Cutting Research, U.S. Department of Energy



IDAES
Institute for the Design of
Advanced Energy Systems



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.

Disclaimer This presentation was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.