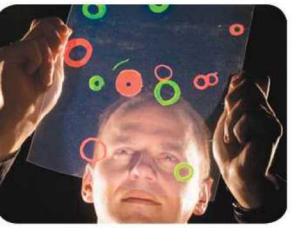
SAND2019-12868C



Applications in Optimizing Dynamic Systems Under Uncertainty

Bethany Nicholson

Center for Computing Research
Sandia National Laboratories
Albuquerque, NM

INFORMS Annual Meeting October 20-23, 2019

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.

“Dynamic system under uncertainty”

Sandia National Laboratories

Dynamic Optimization

Dynamic Model

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

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

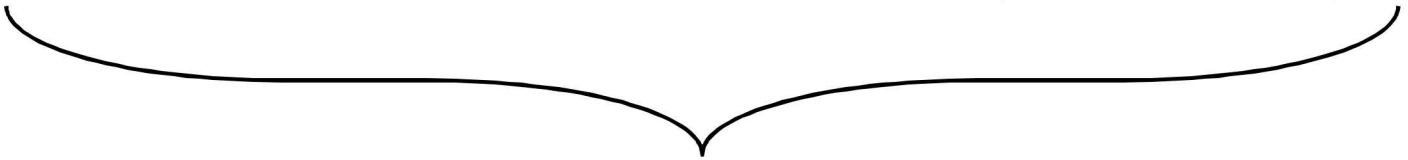
$$x(t_0) = constant$$

Stochastic Programming

Scenario 1

Scenario 2

Scenario 3



Scenario 1

Dyn
M

Scenario 2

Dyn
M

Scenario 3

Dynamic Model

CCR Center for Computing Research

Software platform

Sandia National Laboratories

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

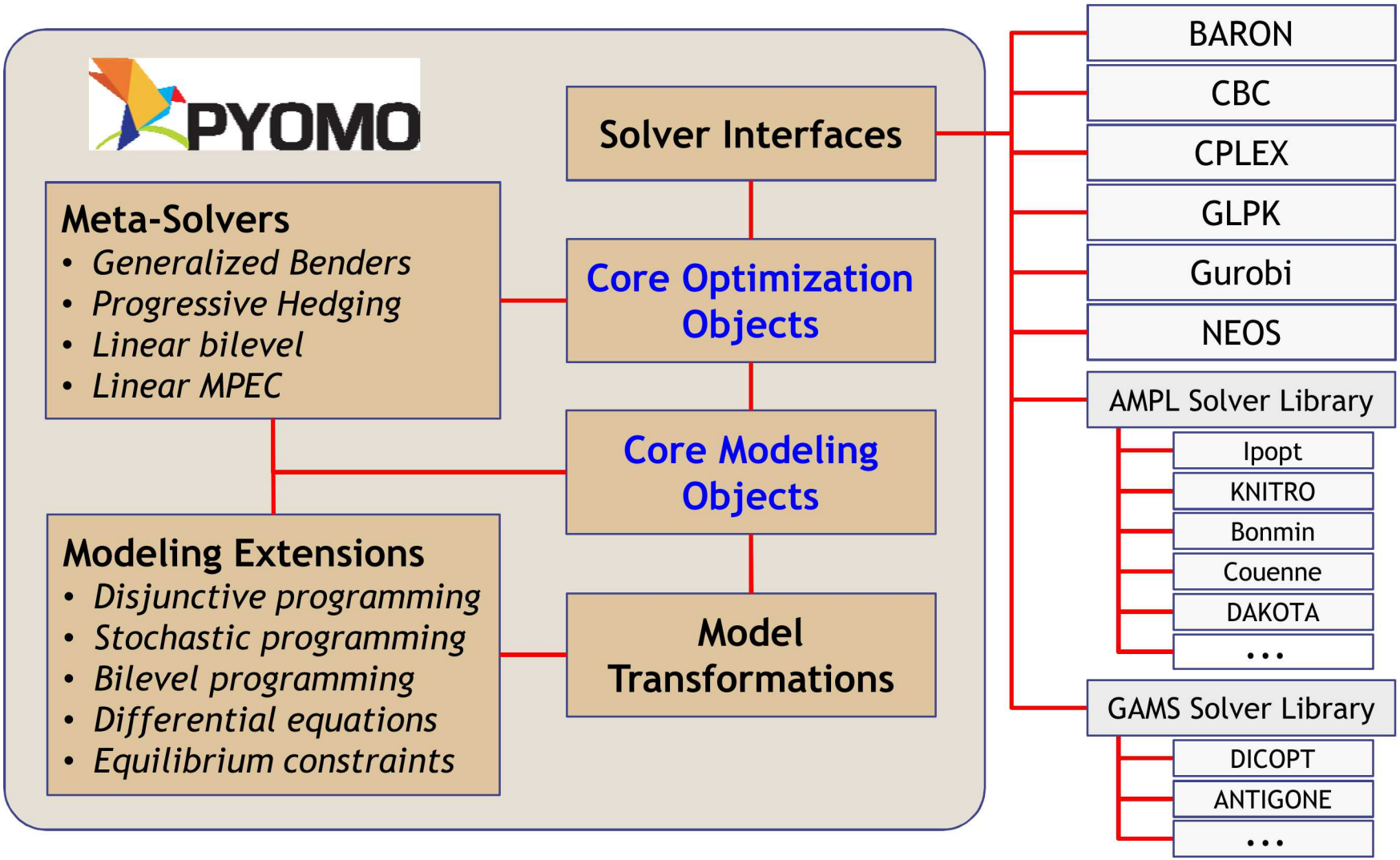

```
from pyomo.environ import *

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

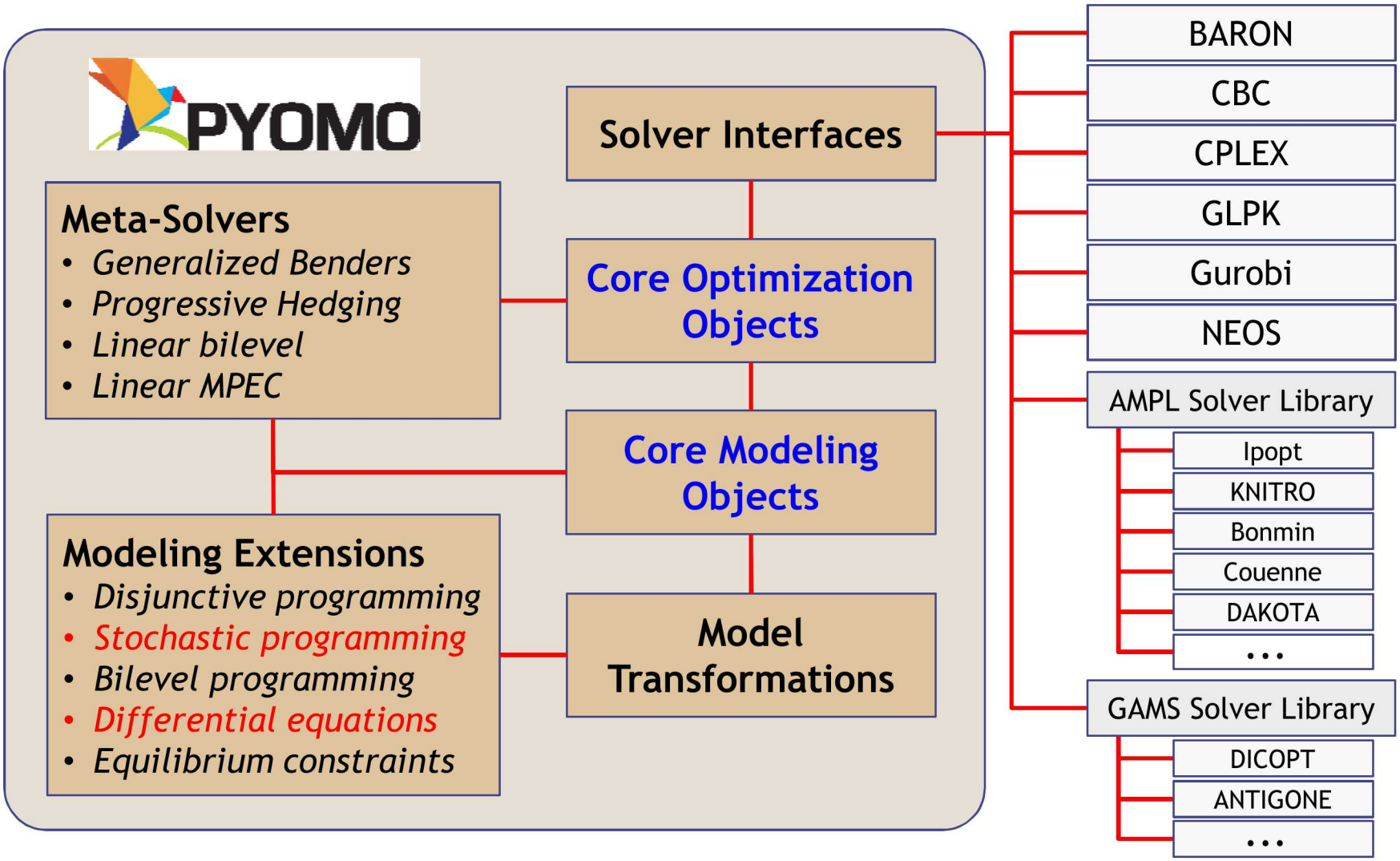
  e.g. SciPy, NumPy, MatPlotLib, Pandas

CCR Center for Computing Research

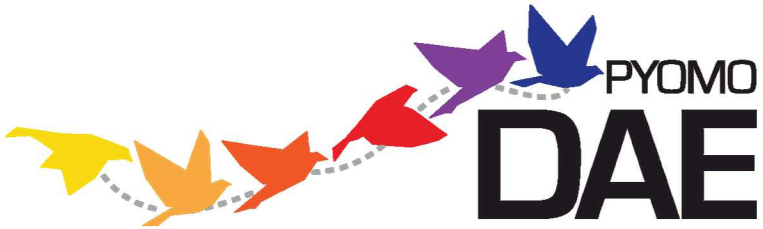
Pyomo at a Glance



Pyomo at a Glance



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

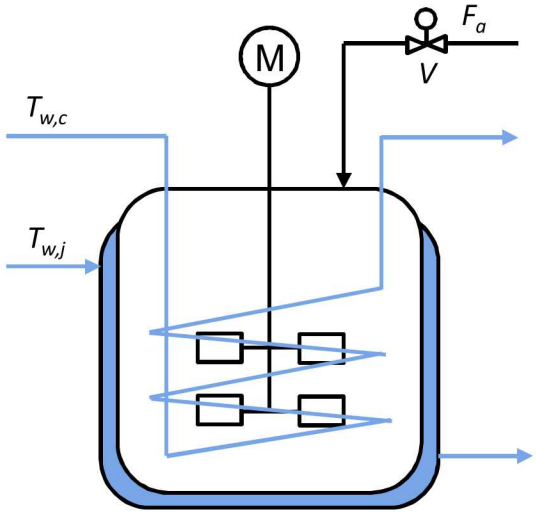
Sandia National Laboratories

- 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]

$$A \rightarrow B \rightarrow C$$



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

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

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

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

$$(\rho_r c_{p_r})\dot{T}_r = \frac{F_a M_{W_a} c_{p_r}}{V_r}(T_f - T_r) - k_1 \exp\left(-\frac{E_1}{RT_r}\right) C_a \Delta H_1 - k_2 \exp\left(-\frac{E_2}{RT_r}\right) C_b \Delta H_2 + \alpha_{w,j}\frac{A_j}{V_{r,0}}(T_{w,j} - T_r) + \alpha_{w,c}\frac{A_c}{V_{r,0}}(T_{w,c} - T_r)$$

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

[2] Abel and Marquardt, AIChE Journal (2000)

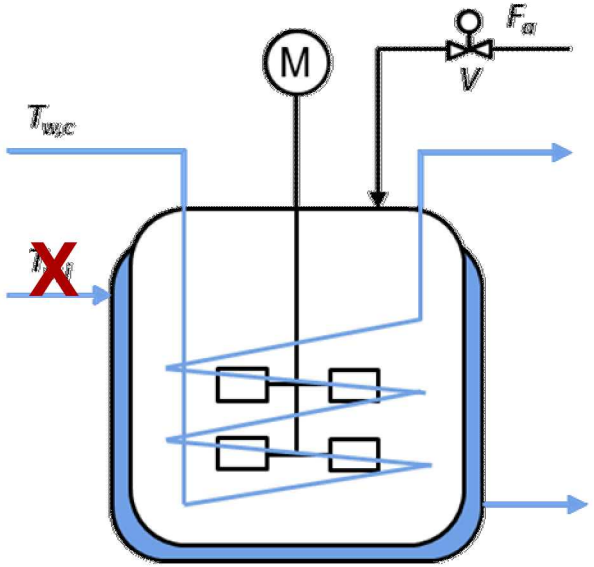
Dynamic model implementation

```
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
```

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

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]

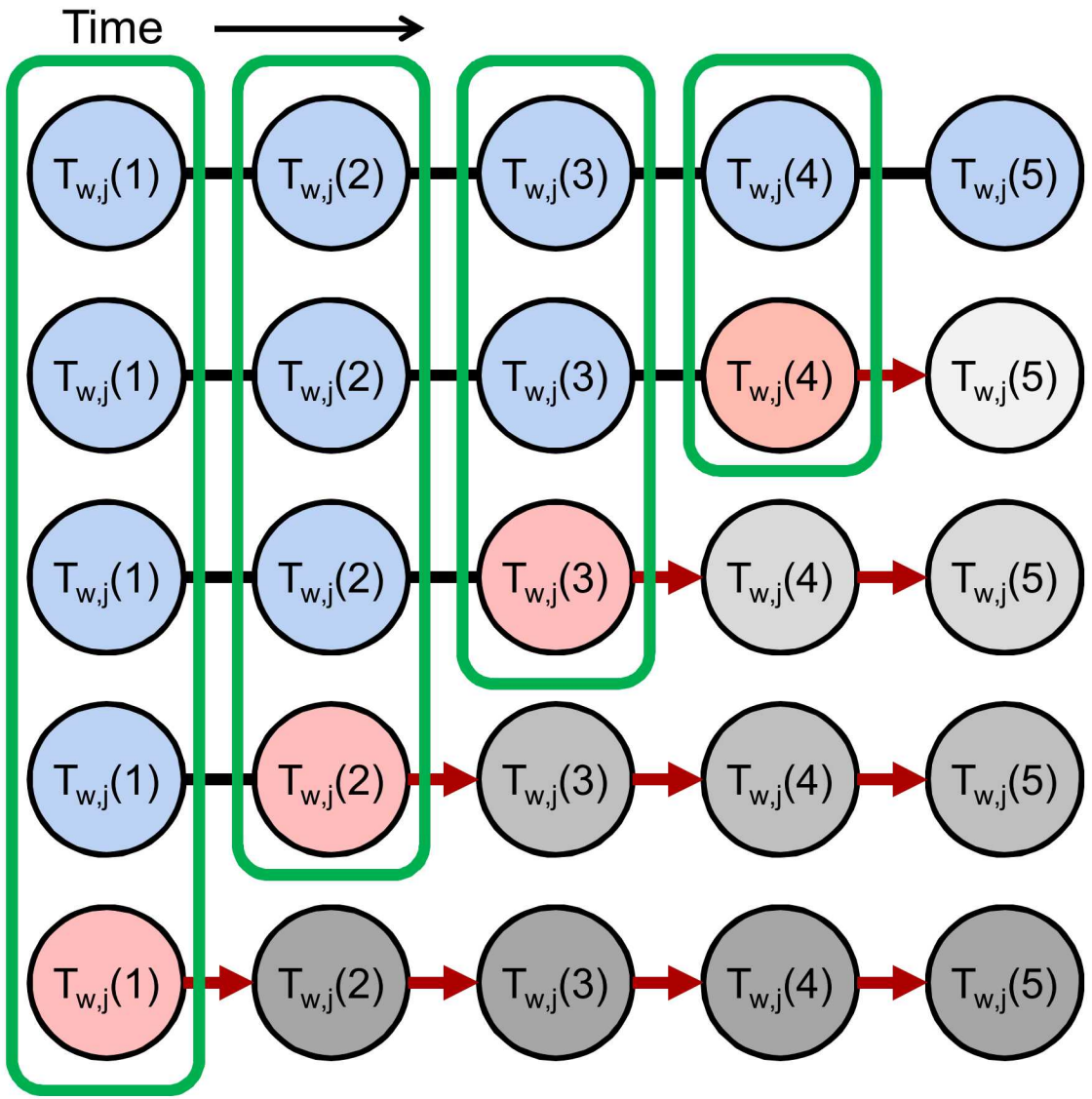


$$\left.\begin{aligned} T_{w,j}(t) &= T_{w,c}(t) && t \le t_{fail} \\ \left(\rho_w C_{p_w} V_j\right)\dot{T}_{w,j} &= \alpha_{w,js} A_j \frac{V_r}{V_{r,0}}\left(T_{w,j} - T_r\right) && t > t_{fail} \end{aligned}\right\}$$

[2] Abel and Marquardt, AIChE Journal (2000)

Optimal control scenarios

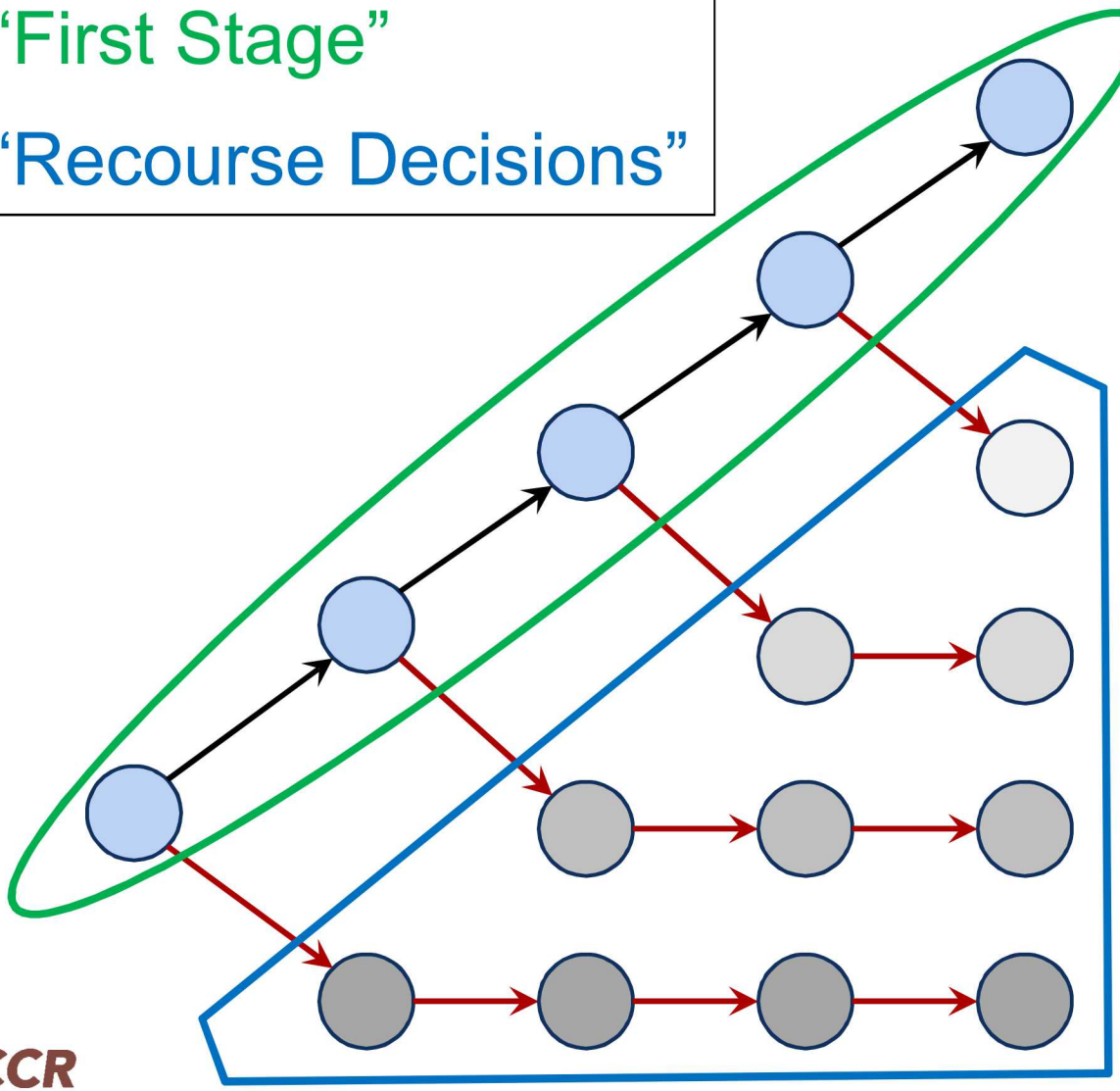
Nonanticipativity Constraints



n-Stage SP as a 2-stage problem

"First Stage"

"Recourse Decisions"



Nominal Scenario

Failure scenarios

Stochastic structure implementation

- Implement callback functions to:
  - Define the scenario tree structure
  - Build a model for each scenario

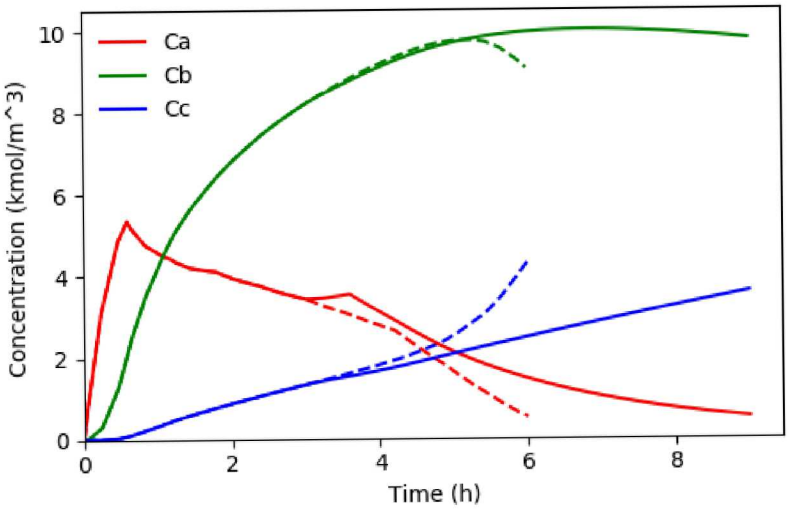
- Create and solve extensive form

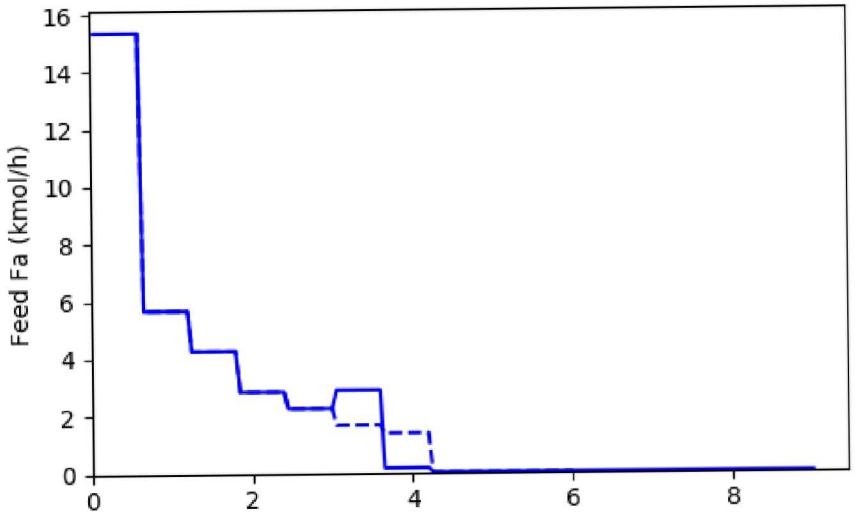
```
runef --solve --solver ipopt --output-solver-log -m semibatch.py
```

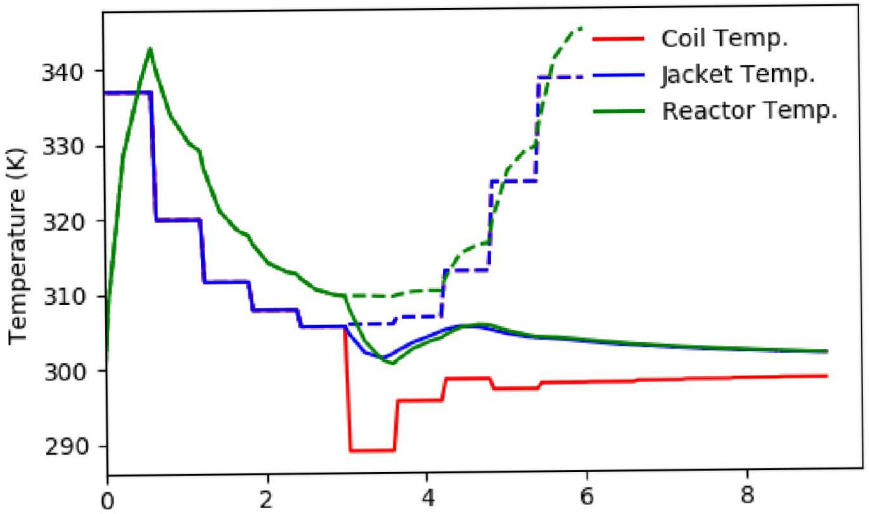
- Solve using progressive hedging

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

Optimal control: t_{fail} = 3 h

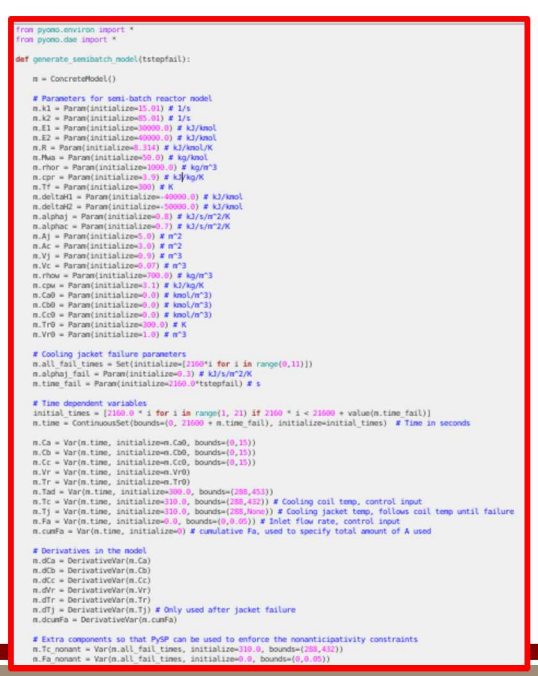






Optimal control implementation

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



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

def generate_semibatch_model(tstepfail):

    m = ConcreteModel()

    # Parameters for semi-batch reactor model
    m.k1 = Param(initialize=15.01) # 1/s
    m.k2 = Param(initialize=85.01) # 1/s
    m.E1 = Param(initialize=30000.0) # kJ/kmol
    m.E2 = Param(initialize=40000.0) # kJ/kmol
    m.R = Param(initialize=8.314) # kJ/kmol/K
    m.Mwa = Param(initialize=50.0) # kg/kmol
    m.rhor = Param(initialize=1000.0) # kg/m^3
    m.cpr = Param(initialize=3.9) # kJ/kg/K
    m.Tf = Param(initialize=300) # K
    m.deltaH1 = Param(initialize=-40000.0) # kJ/kmol
    m.deltaH2 = Param(initialize=-50000.0) # kJ/kmol
    m.alphaj = Param(initialize=0.8) # kJ/s/m^2/K
    m.alphac = Param(initialize=0.7) # kJ/s/m^2/K
    m.Aj = Param(initialize=5.0) # m^2
    m.Ac = Param(initialize=3.0) # m^2
    m.Vj = Param(initialize=0.9) # m^3
    m.Vc = Param(initialize=0.07) # m^3
    m.rhow = Param(initialize=700.0) # kg/m^3
    m.cpw = Param(initialize=3.1) # kJ/kg/K
    m.Ca0 = Param(initialize=5.0) # kmol/m^3)
    m.Cb0 = Param(initialize=0.0) # kmol/m^3)
    m.Cc0 = Param(initialize=0.0) # kmol/m^3)
    m.Tr0 = Param(initialize=300.0) # K
    m.Vr0 = Param(initialize=1.0) # m^3

    # Cooling jacket failure parameters
    m.all_fail_times = Set(initialize=[2160*i for i in range(0,11)])
    m.alphaj_fail = Param(initialize=0.3) # kJ/s/m^2/K
    m.time_fail = Param(initialize=2160.0*tstepfail) # s

    # Time dependent variables
    initial_times = [2160.0 * i for i in range(1, 21) if 2160 * i < 21600 + value(m.time_fail)]
    m.time = ContinuousSet(bounds=(0, 21600 + m.time_fail), initialize=initial_times)  # Time in seconds

    m.Ca = Var(m.time, initialize=m.Ca0, bounds=(0,15))
    m.Cb = Var(m.time, initialize=m.Cb0, bounds=(0,15))
    m.Cc = Var(m.time, initialize=m.Cc0, bounds=(0,15))
    m.Vr = Var(m.time, initialize=m.Vr0)
    m.Tr = Var(m.time, initialize=m.Tr0)
    m.Tad = Var(m.time, initialize=300.0, bounds=(288,453))
    m.Tc = Var(m.time, initialize=310.0, bounds=(288,432)) # Cooling coil temp, control input
    m.Tj = Var(m.time, initialize=310.0, bounds=(288,None)) # Cooling jacket temp, follows coil temp until failure
    m.Fa = Var(m.time, initialize=0.0, bounds=(0,0.05)) # Inlet flow rate, control input
    m.cumFa = Var(m.time, initialize=0) # cumulative Fa, used to specify total amount of A used

    # Derivatives in the model
    m.dCa = DerivativeVar(m.Ca)
    m.dCb = DerivativeVar(m.Cb)
    m.dCc = DerivativeVar(m.Cc)
    m.dVr = DerivativeVar(m.Vr)
    m.dTr = DerivativeVar(m.Tr)
    m.dTj = DerivativeVar(m.Tj) # Only used after jacket failure
    m.dcumFa = DerivativeVar(m.cumFa)

    # Extra components so that PySP can be used to enforce the nonanticipativity constraints
    m.Tc_nonant = Var(m.all_fail_times, initialize=310.0, bounds=(288,432))
    m.Fa_nonant = Var(m.all_fail_times, initialize=0.0, bounds=(0,0.05))
```

```
    # Differential Equations in the model
    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.dCacon = Constraint(m.time, rule=_dCacon)

    def _dCbcon(m,t):
        # if t == 0:
        #     return Constraint.Skip
        return m.dCb[t] == m.k1*exp(-m.E1/(m.R*m.Tr[t]))*m.Ca[t] - \
                           m.k2*exp(-m.E2/(m.R*m.Tr[t]))*m.Cb[t]
    m.dCbcon = Constraint(m.time, rule=_dCbcon)

    def _dCccon(m,t):
        # if t == 0:
        #     return Constraint.Skip
        return m.dCc[t] == m.k2*exp(-m.E2/(m.R*m.Tr[t]))*m.Cb[t]
    m.dCccon = Constraint(m.time, rule=_dCccon)

    def _dVrcon(m,t):
        # if t == 0:
        #     return Constraint.Skip
        return m.dVr[t] == m.Fa[t]*m.Mwa/m.rhor
    m.dVrcon = Constraint(m.time, rule=_dVrcon)

    def _dTrcon(m,t):
        if t == 0:
            return Constraint.Skip
        return m.rhor*m.cpr*m.dTr[t] == \
                m.Fa[t]*m.Mwa*m.cpr/m.Vr[t]*(m.Tf-m.Tr[t]) - \
                m.k1*exp(-m.E1/(m.R*m.Tr[t]))*m.Ca[t]*m.deltaH1 - \
                m.k2*exp(-m.E2/(m.R*m.Tr[t]))*m.Cb[t]*m.deltaH2 + \
                m.alphaj*m.Aj/m.Vr0*(m.Tj[t]-m.Tr[t]) + \
                m.alphac*m.Ac/m.Vr0*(m.Tc[t]-m.Tr[t])
    m.dTrcon = Constraint(m.time, rule=_dTrcon)

    def _singlecooling(m,t):
        # Apply this constraint at time_fail to set initial condition for Tj diff eq
        if value(m.time_fail) == 0 or t <= value(m.time_fail):
            return m.Tc[t] == m.Tj[t]
        else:
            return Constraint.Skip
    m.singlecooling = Constraint(m.time, rule=_singlecooling)

    def _adTempcon(m,t):
        return (m.Tad[t]-m.Tr[t])*(m.rhor*m.cpr*m.Vr[t] +
        m.rhow*m.cpw*(m.Vj+m.Vc)) == \
        m.Vr[t]*((-m.deltaH1-m.deltaH2)*m.Ca[t] + (-m.deltaH2)*m.Cb[t])
    m.adTempcon = Constraint(m.time, rule=_adTempcon)

    def _dTjcon(m,t):
        if value(m.time_fail) == 0 or t <= value(m.time_fail):
            return Constraint.Skip
        else:
            return m.rhow*m.cpw*m.Vj*m.dTj[t] == \
        m.alphaj_fail*m.Aj*m.Vr[t]/m.Vr0*(m.Tj[t]-m.Tr[t])
    m.dTjcon = Constraint(m.time, rule=_dTjcon)

    def _dcumFacon(m,t):
        if t == m.time.first():
            return Constraint.Skip
        return m.dcumFa[t] == m.Fa[t]
    m.dcumFacon = Constraint(m.time, rule=_dcumFacon)

    # Bound on cumulative Fa
    m.cumFalimit = Constraint(expr=m.cumFa[m.time.last()] <= 20)

    # Bound on final Ca
    m.Cafinal = Constraint(expr=m.Ca[m.time.last()] <= 0.5)
```

```
    # Initial Conditions
    def _initcon(m):
        yield m.Ca[m.time.first()] == m.Ca0
        yield m.Cb[m.time.first()] == m.Cb0
        yield m.Cc[m.time.first()] == m.Cc0
        yield m.Vr[m.time.first()] == m.Vr0
        yield m.Tr[m.time.first()] == m.Tr0
        yield m.cumFa[m.time.first()] == 0
    m.initcon = ConstraintList(rule=_initcon)

    # Helper constraints for enforcing nonanticipativity constraints
    def _Tc_nonantcon(m,t):
        if value(m.time_fail) == 0 or t<= value(m.time_fail):
            return m.Tc_nonant[t] == m.Tc[t]
        return Constraint.Skip
    m.Tc_nonantcon = Constraint(m.all_fail_times, rule=_Tc_nonantcon)

    def _Fa_nonantcon(m,t):
        if value(m.time_fail) == 0 or t <= value(m.time_fail):
            return m.Fa_nonant[t] == m.Fa[t]
        return Constraint.Skip
    m.Fa_nonantcon = Constraint(m.all_fail_times, rule=_Fa_nonantcon)

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

    def ComputeSecondStageCost_rule(model):
        return -m.Cb[m.time.last()]
    m.SecondStageCost = Expression(rule=ComputeSecondStageCost_rule)

    def total_cost_rule(model):
        return model.FirstStageCost + model.SecondStageCost
    m.Total_Cost_Objective = Objective(rule=total_cost_rule, sense=minimize)
```

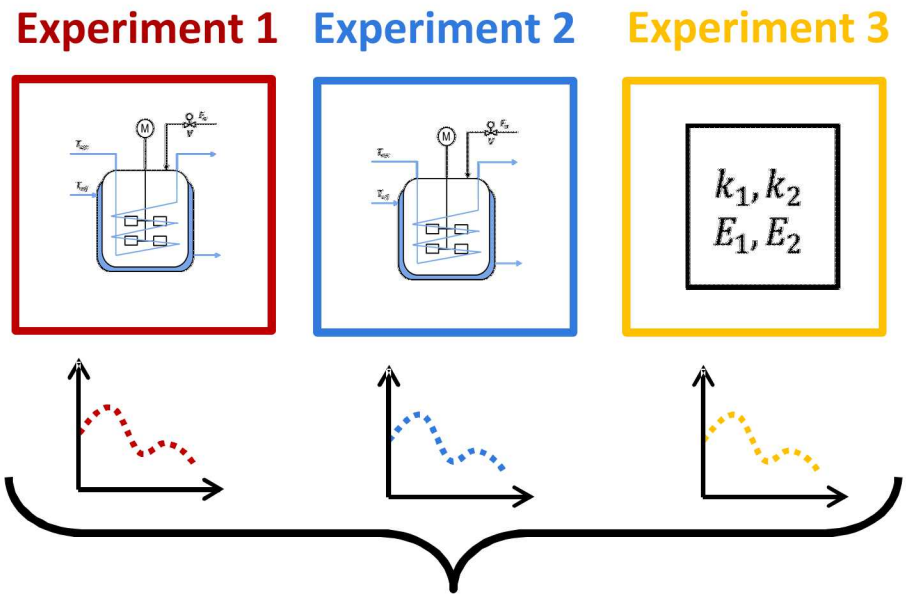
```
    # Discretize model. Number of finite elements depends on the cooling failure time
    numfe = m.time.last()/2160.0
    disc = TransformationFactory('dae.collocation')
    disc.apply_to(m, nfe=numfe, ncp=4)
    disc.reduce_collocation_points(m, var=m.Fa, ncp=1, contset=m.time)
    disc.reduce_collocation_points(m, var=m.Tc, ncp=1, contset=m.time)
    return m
```

```
# Number of scenarios
scenarios = 10

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('Fa_nonant[*]')
    st_model.StageVariables[first_stage].add('Tc_nonant[*]')

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

    return st_model

def pysp_instance_creation_callback(scenario_name, node_names):
    failtimestep = int(scenario_name.replace('Scenario',''))
    instance = generate_semibatch_model(failtimestep-1)

    return instance
```

```
import os
from pyomo.environ import *
from pyomo.pysp.scenariotree.manager import \
    ScenarioTreeManagerClientSerial
from pyomo.pysp.ef import create_ef_instance

thisdir = os.path.dirname(os.path.abspath(__file__))

options = ScenarioTreeManagerClientSerial.register_options()

options.model_location = os.path.join(thisdir, 'semibatch.py')

manager = ScenarioTreeManagerClientSerial(options)
manager.initialize()

ef_instance = create_ef_instance(manager.scenario_tree, verbose_output=True)
print('Created EF instance')
solver = SolverFactory('ipopt')

solver.solve(ef_instance, tee=True)
```

```
m = ef_instance.Scenario6

nomtime = [i/3600.0 for i in nom.time]
time = [i/3600.0 for i in m.time]

import matplotlib.pyplot as plt

grey1 = '0.7'
grey2 = '0.4'
grey3 = '0'

plt.subplot(313)
plt.plot(nomtime,[value(nom.Ca[t]) for t in nom.time],'--',color=grey1)
plt.plot(nomtime,[value(nom.Cb[t]) for t in nom.time],'--',color=grey2)
plt.plot(nomtime,[value(nom.Cc[t]) for t in nom.time],'--',color=grey3)
plt.plot(time,[value(m.Ca[t]) for t in m.time],label='Ca',color=grey1)
plt.plot(time,[value(m.Cb[t]) for t in m.time],label='Cb',color=grey2)
plt.plot(time,[value(m.Cc[t]) for t in m.time],label='Cc',color=grey3)
plt.legend(loc='best')
plt.xlabel('Time (h)')
plt.ylabel('Concentration (kmol/m^3)')
plt.xlim(xmin=0)
plt.ylim(ymin=0)

plt.subplot(311)
plt.plot(time[1:],[value(m.Fa[t])*3600 for t in m.time][1:],color=grey2)
plt.plot(nomtime[1:],[value(nom.Fa[t])*3600 for t in nom.time][1:],'--',color=grey2)
plt.legend(loc='best')
plt.ylabel('Feed Fa (kmol/h)')
plt.xlim(xmin=0)
plt.ylim(ymin=0)

plt.subplot(312)
plt.plot(nomtime[1:],[value(nom.Tc[t]) for t in nom.time][1:],'--',color=grey1)
plt.plot(nomtime[1:],[value(nom.Tj[t]) for t in nom.time][1:],'--',color=grey2)
plt.plot(nomtime,[value(nom.Tr[t]) for t in nom.time],'--',color=grey3)
plt.plot(time[1:],[value(m.Tc[t]) for t in m.time][1:],label='Coil Temp.',color=grey1)
plt.plot(time[1:],[value(m.Tj[t]) for t in m.time][1:],label='Jacket Temp.',color=grey2)
plt.plot(time,[value(m.Tr[t]) for t in m.time],label='Reactor Temp.',color=grey3)
plt.legend(loc='best')
plt.ylabel('Temperature (K)')
plt.xlim(xmin=0)
plt.show()
```

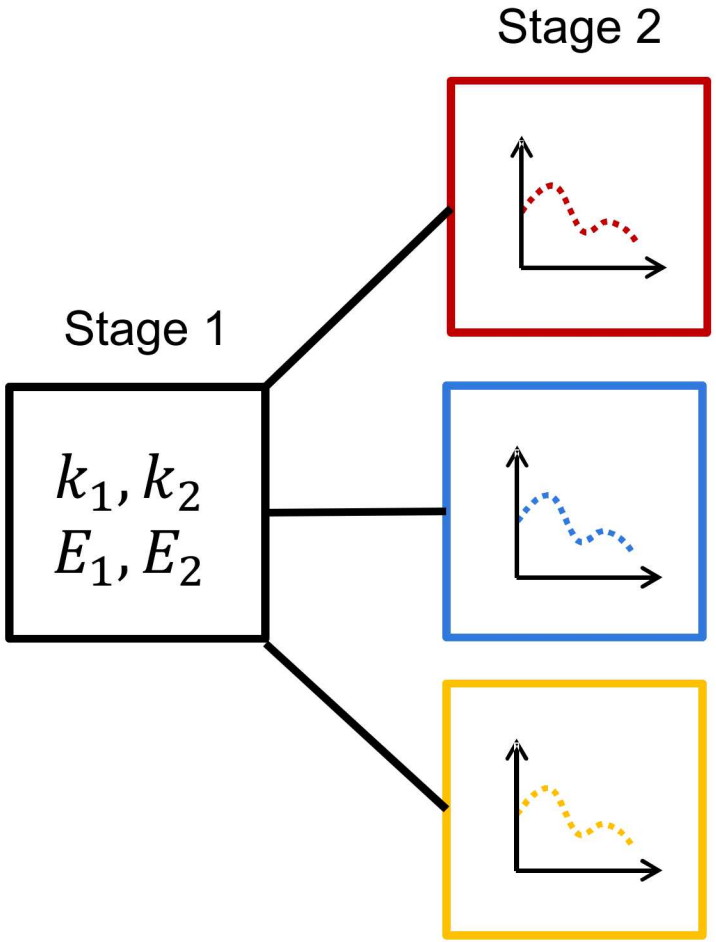
Parameter estimation



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

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

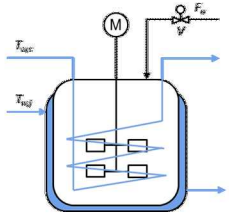
$$s.t. \quad semibatch\ model\ equations$$



Parameter estimation

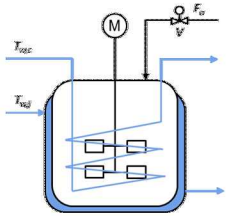
- Three ***different*** experiments
  - different manipulated variables, different measured data

Experiment 1



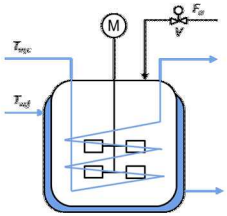
Step change: F_a
Measure: C_a, C_b, C_c, T_r

Experiment 2



Step change: $T_{w,c}$
Measure: T_r

Experiment 3



Step change: $F_a, T_{w,c}$
Measure: C_b, T_r

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

Semibatch parameter estimation results

Sandia National Laboratories

- 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

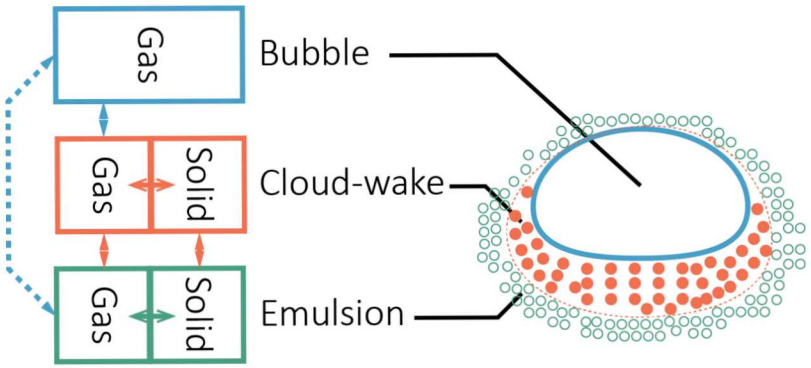
CCR Center for Computing Research

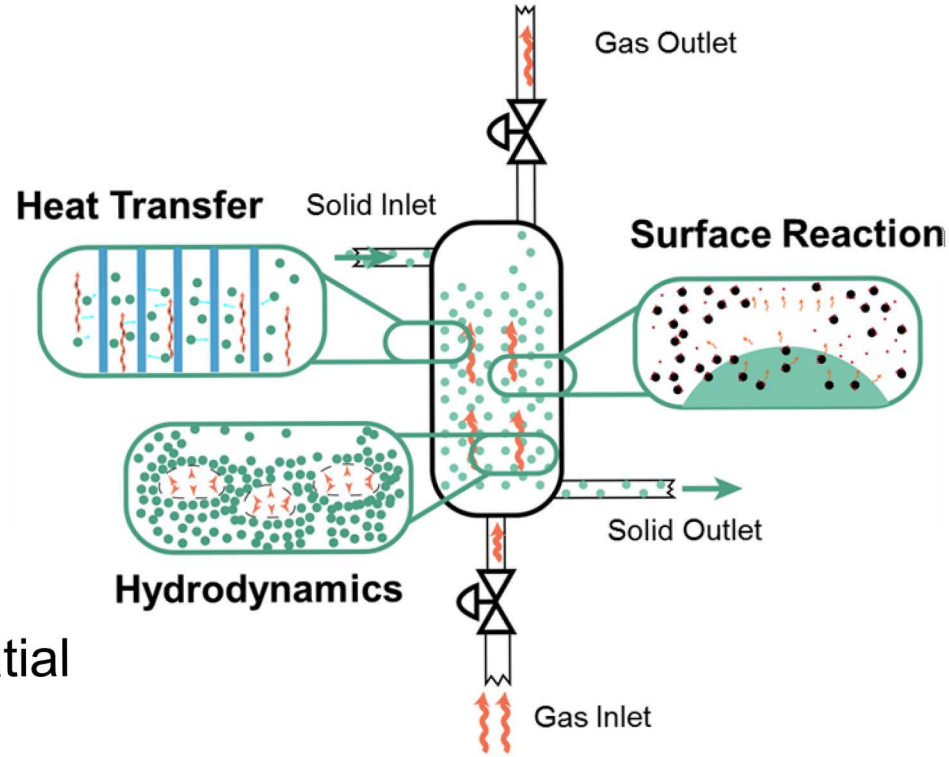
** PH is not guaranteed to converge on nonconvex problems*

Bubbling Fluidized Bed (BFB) Model

- Gas-solid, 3 region model

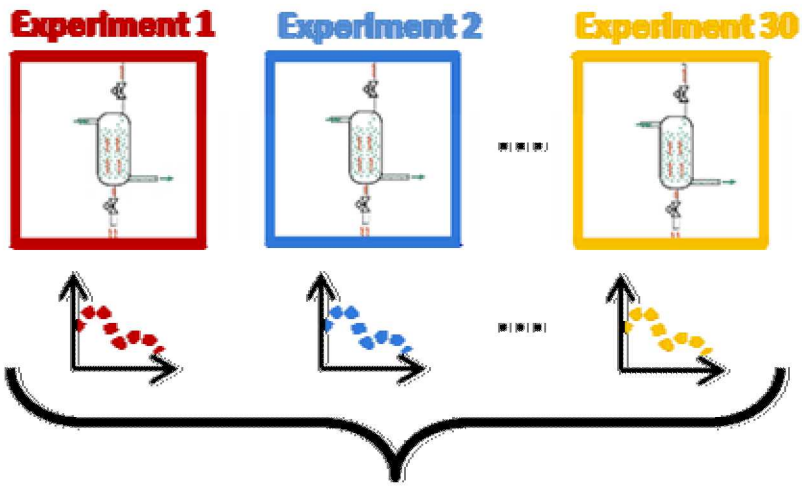
(Lee and Miller, 2013, Ind. Eng. Chem. Res.)





- Steady state model with 1-D spatial variation

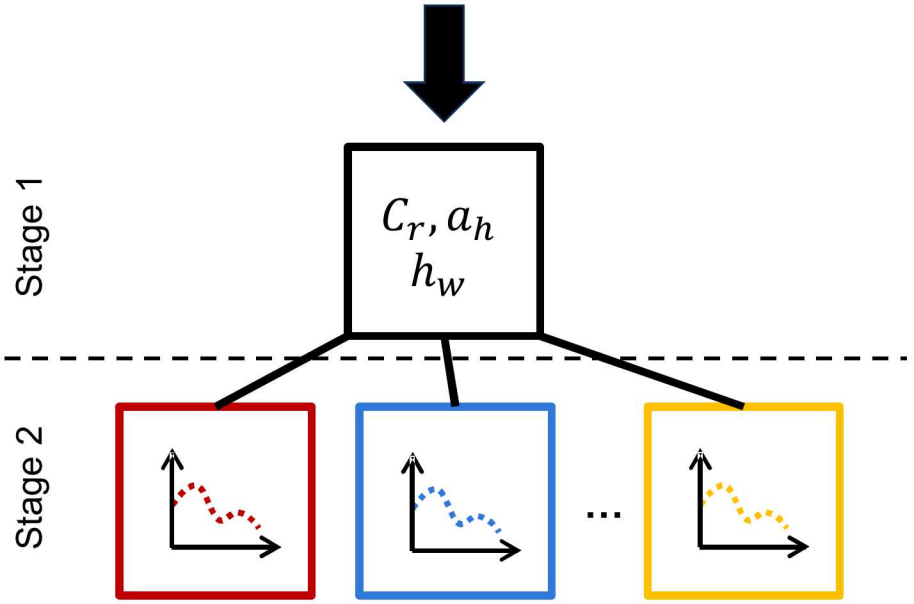
BFB Parameter Estimation (1/2)



Heat Exchanger Model Parameters	
C_r	Average correction factor for tube model
a_h	Empirical factor for tube model
h_w	Heat transfer coefficient of tube walls

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

$$s.t. \quad BFB\ model\ equations$$



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 processors)	0.9824	0.7850	1501.74	610.98
Progressive Hedging (30 processors)	0.9824	0.7850	1501.74	459.10

Extensive form problem size ~400,000 variables and constraints
PH subproblem size ~13,000 variables and constraints

Summary

- Explicitly capturing both uncertainty and system dynamics can be important for many real-world applications
- Pyomo provides high-level modeling constructs that can be easily combined to solve complex, structured optimization problems. (www.pyomo.org)

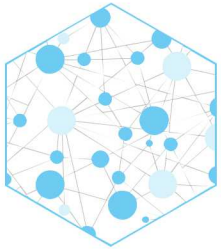
New application areas

- Dynamics-informed optimization for resilient energy
- Trajectory planning under uncertainty

Thursday morning sessions
TA33. Pyomo I
TB33. Pyomo II

Questions?

- Acknowledgements
  - John Siirola, Carl Laird, Jean-Paul Watson, Pyomo team
  - 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.

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

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
```