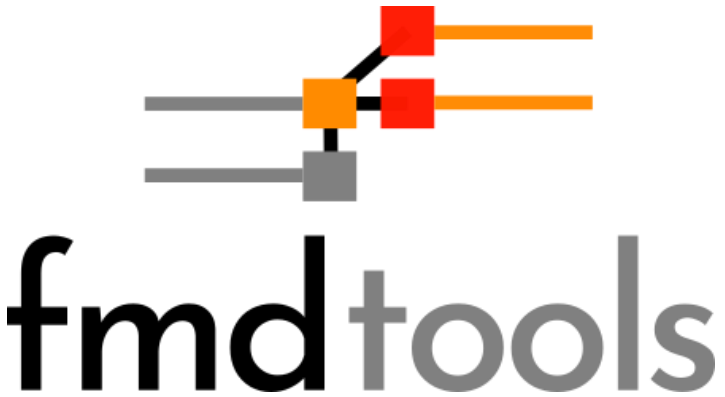


Intro to resilience modelling, simulation, and visualization in Python with fmdtools

Author: Daniel Hulse

Version 1.0.3

Overview/Prerequisites

- Overview of fmdtools
  - Purpose
  - Code structures
  - Methods
  - Syntax
- Coding Activity
  - Example model
  - Workbook
    - Model Instantiation
    - Simulation
    - Visualization/Analysis

Download Slides:

- /docs/Intro to Fmdtools
- Right click "Save link as..."

You need a Python distribution!

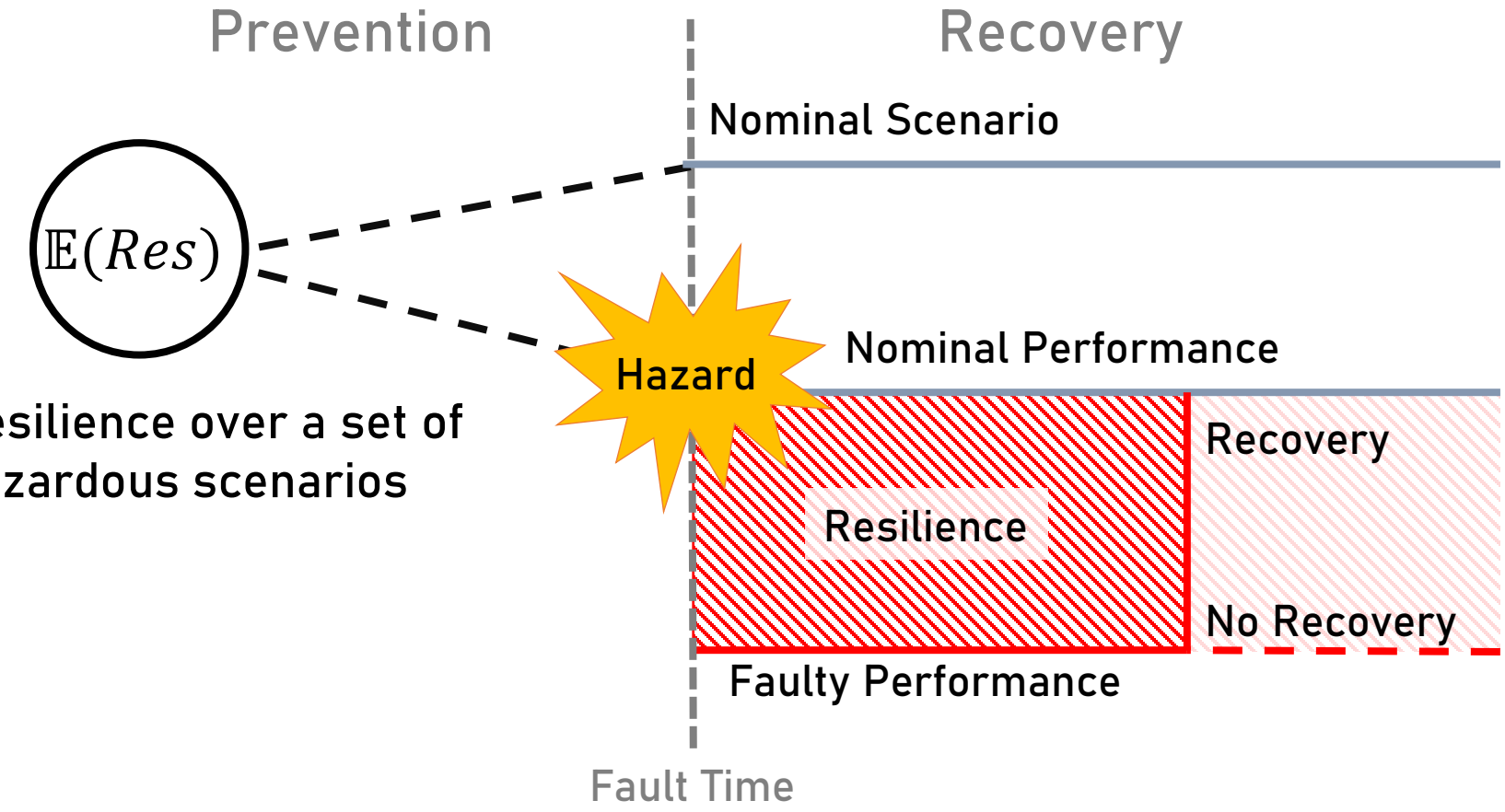
(preferable Anaconda)

https://www.anaconda.com/products/individual

Why model system resilience?

- We want to know how the system will respond to hazardous events, to:
  - Reduce unforeseen maintenance/repairs
  - Ensure system safety
  - Check that system behavior will be robust to unforeseen events
- In a hazardous scenario:
  - Will the system fail to a safe condition?
  - Will it recover?
  - Or, will it wreak havoc on itself and the environment?

What is resilience?

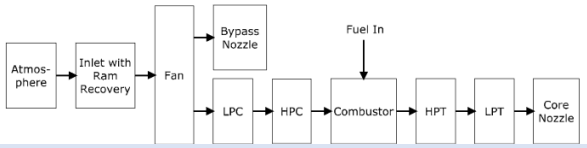


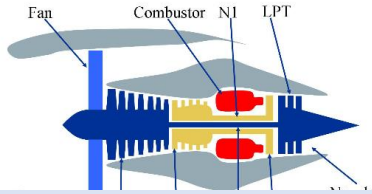
Expected Resilience: Resilience over a set of hazardous scenarios

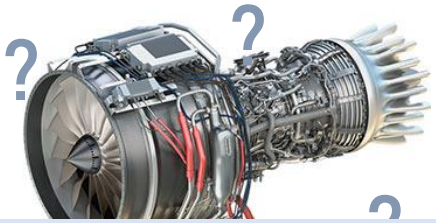
Yodo, N, & Wang, P. (2016).

Why use resilience in design?

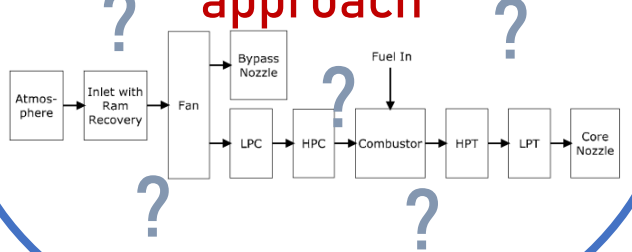
	Concept Design	Embodiment Design	Implementation
Reactive Design	Choose concept	Design system	Retrofit for resilience
Proactive Design	Establish resilience approach	Integrate resilient features in design	Verify resilient function

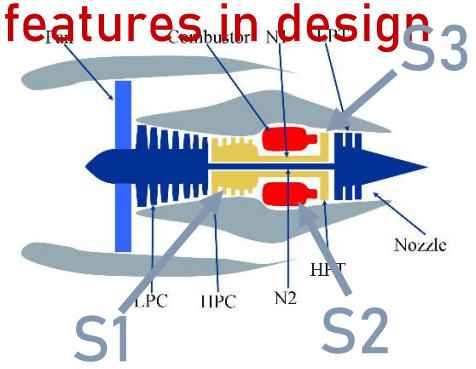






So we can make the system resilient-by-design





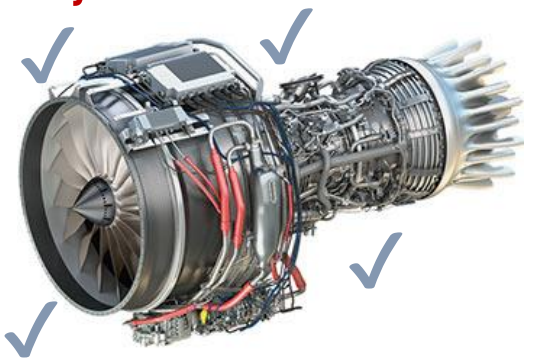


Image 1 & 2 Credit: User's Guide for the Commercial Modular Aero-Propulsion System Simulation (C-MAPSS)

Image 3 Credit: https://en.wikipedia.org/wiki/File:General_Electric_Passport.jpg

Why fmdtools? Why not an existing tool?

- Uncertainty Quantification tools: (e.g. OpenCossan)
  - Doesn't incorporate fault modelling/propagation/visualization aspects
- MATLAB/modelica/etc. Fault Simulation tools
  - Rely on pre-existing model/software stack
  - Useful, but often difficult to hack/extend
- Safety Assessment tools: (e.g. Alyrica, Hip-Hops)
  - Focused on quantifying safety, not necessarily resilience
    - (though they are related)
  - Different model formalisms!

Why fmdtools?

Expressive:

- Undirected graph propagation
  - faults from any component can propagate to any other connected component
- General model representation—not a strict formalism
  - Behavioral (equations), fault logic (if-else statements), etc.
- Dynamic simulation needed to quantify resilience

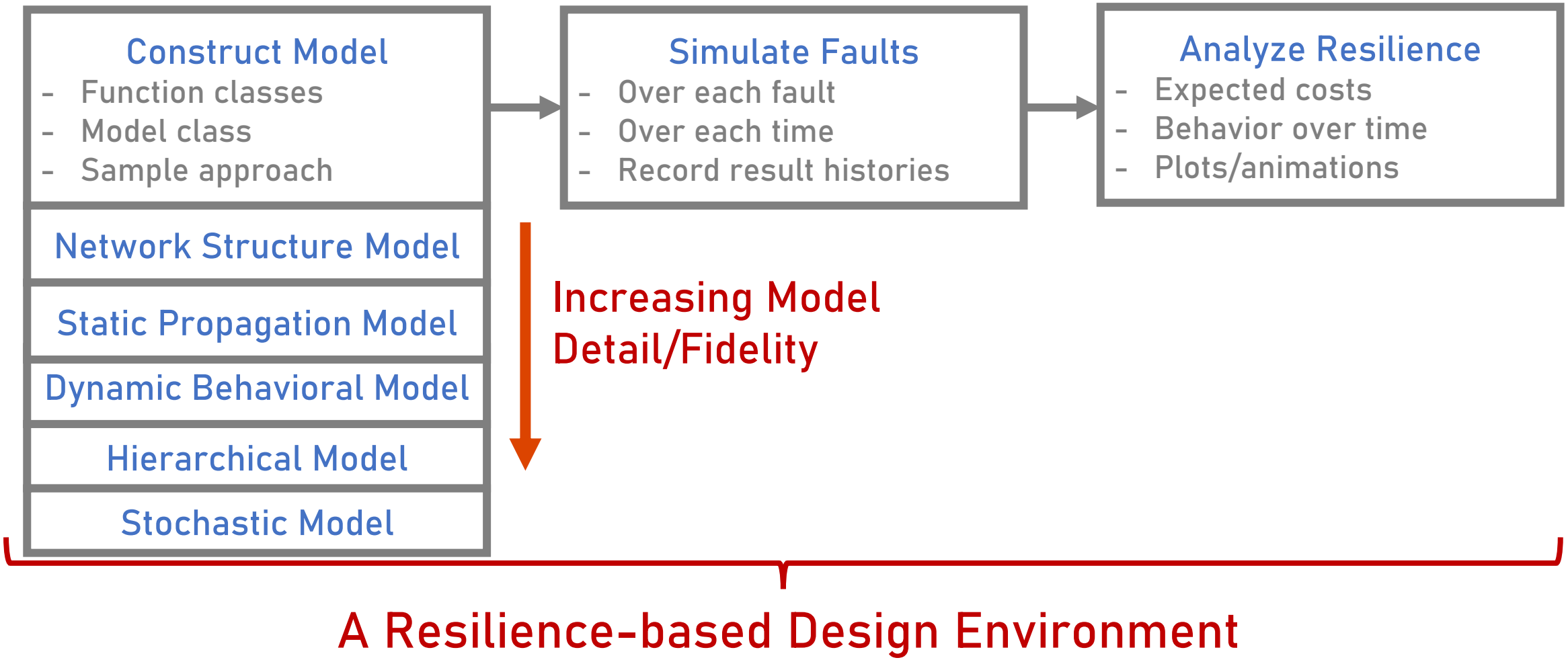
- Research-oriented:
  - Written in/relies on the Python stack
  - Open source/free software
- Enables design:
  - Models can be parameterized an optimized!
  - Provides tools to visualize and quantify simulation results

Why not fmdtools?

Limitations:

- Doesn't work with existing models in different software
  - no integration with Simulink, modelica, etc. system models
- It's research code
  - Not yet fully tested outside of known use-cases
  - Ongoing development may create problems for existing models

What is fmdtools?



Repository

https://github.com/nasa/fmdtools

- /fmdtools – toolkit modules
- /example_XXX – example models
  - Example_pump - Simple model and tutorial/demonstration notebooks
  - Example_multirotor - Example of several model types from "fmdtools: A Fault Propagation Toolkit for Resilience Assessment in Early Design."
  - Example_eps - Example of a static propagation model
  - Example_tank - Example of a human error model implemented with components
- /docs - additional documentation
- /tests - automated software tests

Toolkit modules also available from PyPI (e.g. pip install fmdtools)

Activity: Download and Install fmdtools

- Make sure Python environment is installed
  - e.g. https://www.anaconda.com/products/individual
- Install fmdtools from PyPI:
  - pip install fmdtools
- Download workbook files from Github repository
  - https://github.com/nasa/fmdtools
  - Navigate to /example_pump
  - Right Click Tutorial_unfilled.ipynb and click "Save Link As" to download
  - Right Click ex_pump.py and click "Save Link As" to download to your computer
- Open the notebook and model!

Structure: /fmdtools

Design

modeldef.py

- **Model class**: representation of model functions, flows, behaviors, faults, etc.
  - Function class
  - Flow class, …
- **SampleApproach class**: defines set of fault scenarios to represent the statistical expectation of resilience
- **NominalApproach class**: defines set of nominal scenarios for simulating design/operational envelope

Simulation

/faultsim

networks.py

Quantification and visualization of metrics using a network model representation

propagate.py

Propagation of model behaviors in nominal and faulty scenarios

- nominal(), one_fault()
- singlefaults(), approach()
- nominal_approach()
- nested_approach()

Analysis

/resultdisp

process.py

Processing of simulation results into metrics/statistics

plot.py

Plotting of system behaviors and costs over time

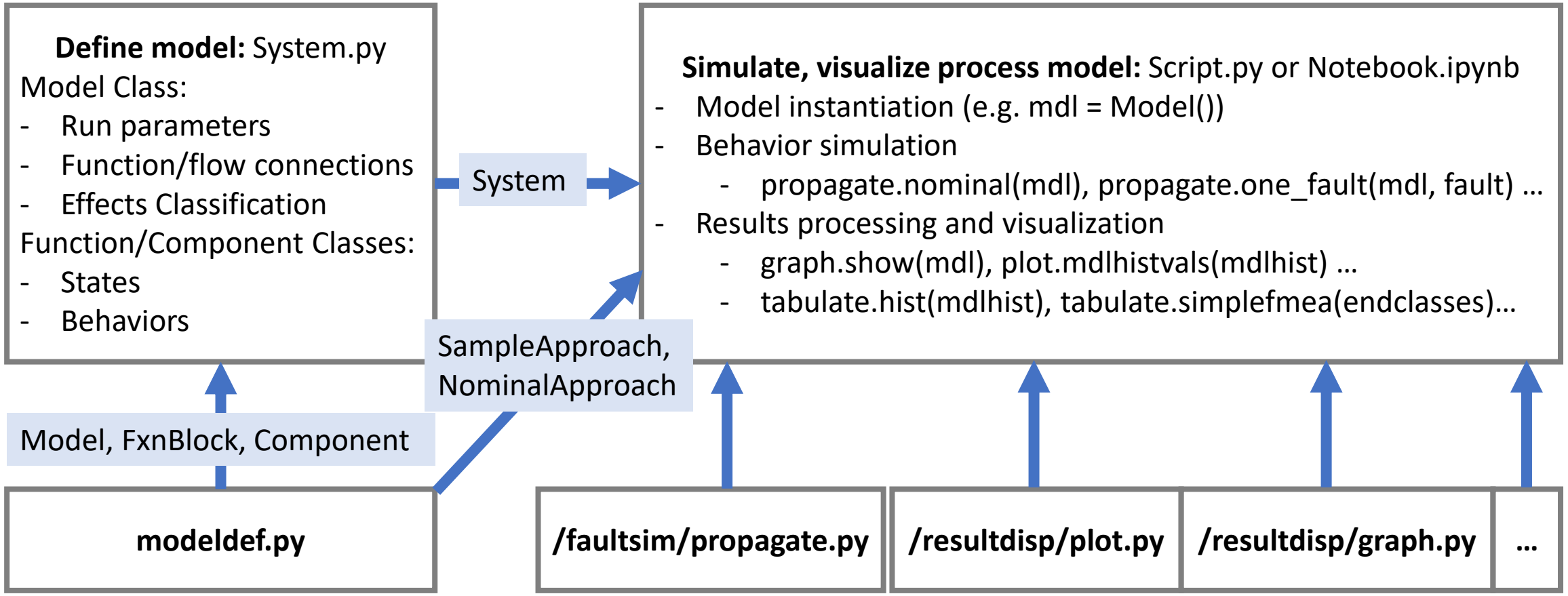
graph.py

Visualization of simulation results on the model graph

tabulate.py

Display and export of simulation results as tables

A typical work-flow



Defining a model

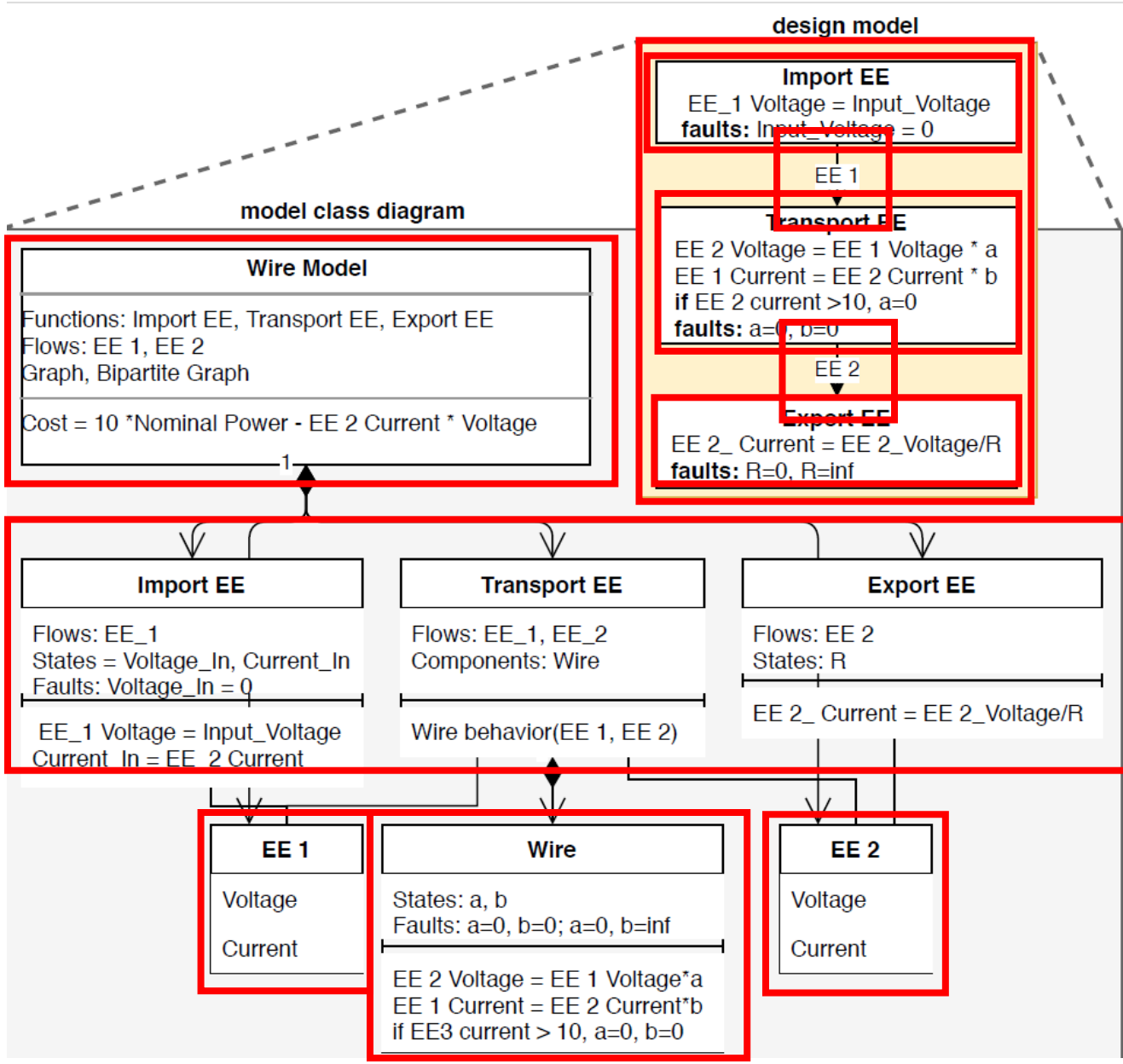
What do we want out of a model?

- What functions/components and how much fidelity do we need?
  - Network or behavioral?
  - Dynamic or static?
  - Functions or component architectures?
  - Behaviors or failure logic?
- What can go wrong? What are the faults/events?
- What model parameters do we need to assess the resilience over?
- What are the possible effects and how bad are they?

Model structure

Flows

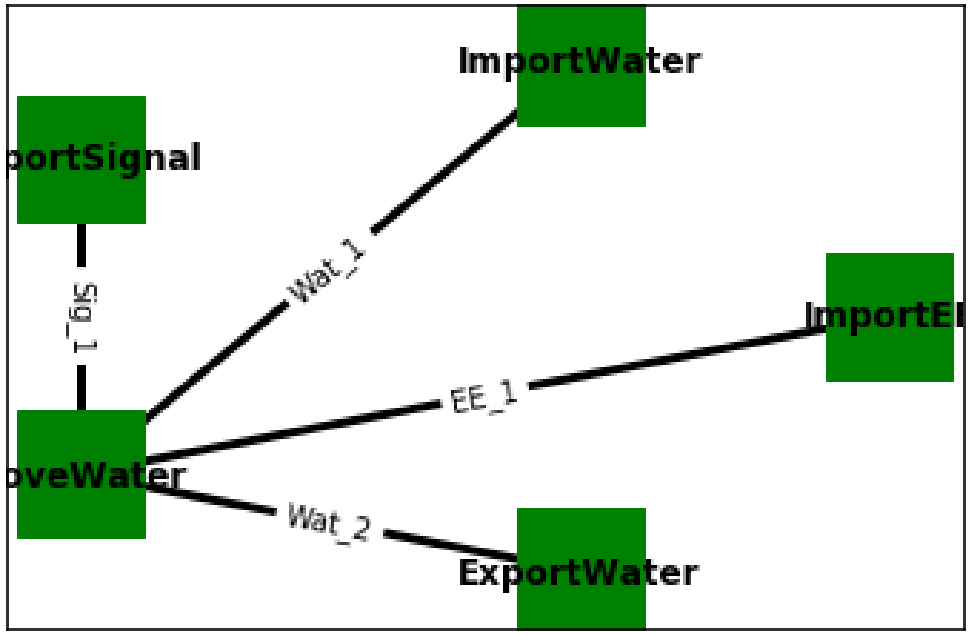
- State Dictionary
- EE = {voltage: 1.0, current:1.0}



Demonstration Model - Pump

/pump example/ex_pump.py

- Functions: (nodes)
  - Define behaviors, and faults of the of the system
  - e.g. ImportSignal, ImportWater, MoveWater, ExportWater, ImportEE
- Flows: (edges)
  - Define the connections between functions of material, energy, signal, etc.
  - e.g. Sig_1, Wat_1, EE_1, Wat_2



Defining a model – generic code

```
from fmdtools.modeldef import Model, FxnBlock        (imports)
class  System(Model):        (defining model class for system)
        def __init__(self, params={}, modelparams={}, valparams={})        (model initialization method)
                super().__init__(params,modelparams,valparams)        (attach params to model)
                self.add_flow(flowname, {value1:initvalue, value2: initvalue} ...        (define model flows)
                self.add_fxn(functionname, [list of flows], fclass = DoSomething) ...        (add and connect functions)
                self.build_model()        (form model graph and run order, etc.)
        def end_condition(self, time):
                (anything)
                return True/False at end-state
        def find_classification(self,resgraph, endfaults, endflows, scen, mdlhists):
                (anything)        (define how to classify model results)
                return {'rate':rate, 'cost':cost, 'expected cost'=expected cost
```

design params

simulation params (e.g., start/end)

value params (fxns/flows to track)

Defining functions – generic code

```
Class DoSomething(FxnBlock):
    def __init__(self, name, flows):
        super.__init__(name, flows,{state1: initval}, ...)
        self.failrate = failrate
        self.assoc_modes({mode1:properties}, [opermodes])
        self.assoc_randstate(statename, default)
    def behavior(self, time):
        if self.flowname.val1 > threshold: self.add_fault('mode1')
        if self.has_fault('mode1'): self.flowname.val2 = 0
        else: self.flowname.val2 = 1
    def dynamic_behavior(self,time):
        self.state1.inc( = self.state1 + self.flowname.val2
    def static_behavior(self,time):
```

Define class for DoSomething function

initialization method (can also be used to pass parameters)

add/associate model constructs: flows, states, timers, components

Define fault/operational modes

Associate random states (for stochastic simulation)

generic method for all (static and dynamic) behavior

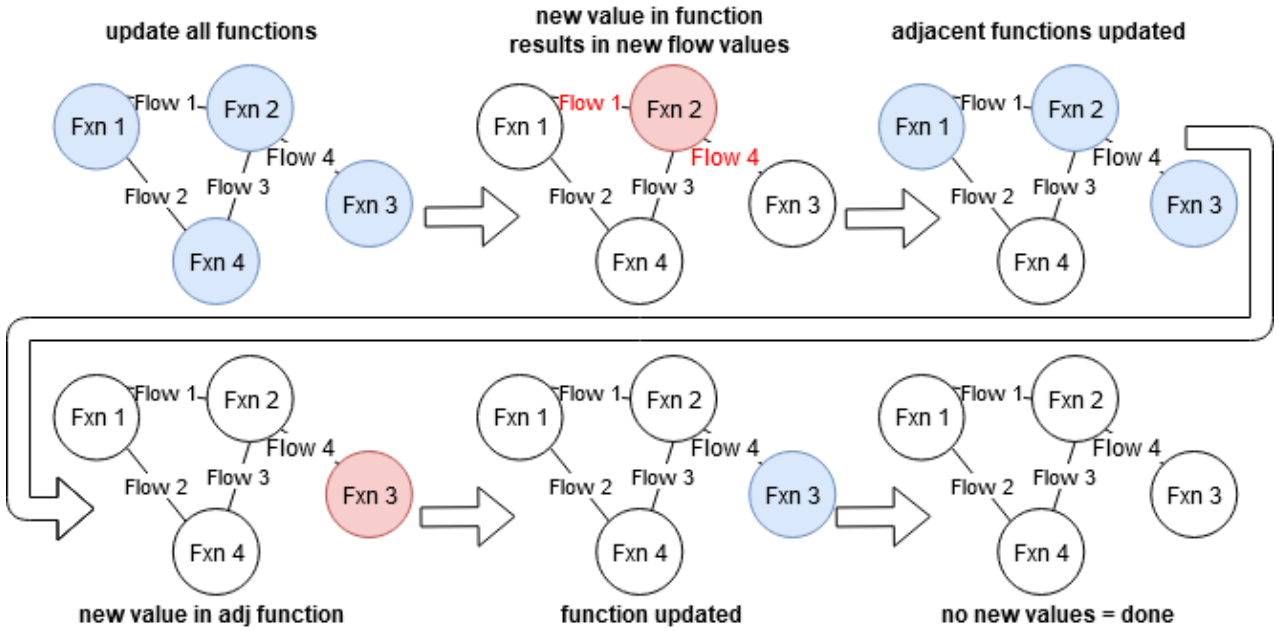
(different modes change behaviors)

method for dynamic (once per timestep) behaviors

(accumulation of a state over time)

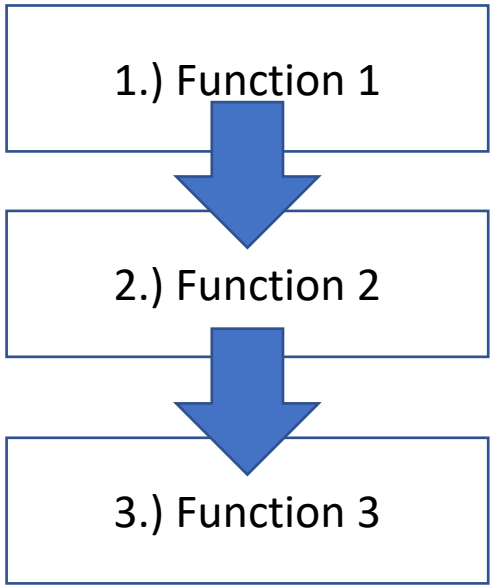
method for solely static behaviors that can propagate between constructs during a single timestep

Static Behaviors



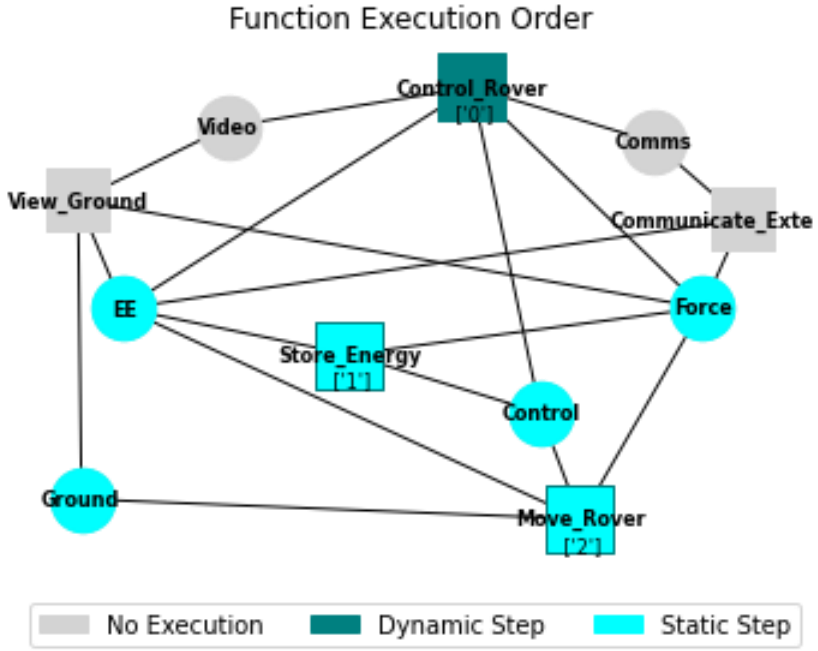
- Behaviors defined in .behavior(), .static_behavior(), and .condfaults()
- Propagates multiple times per timestep until the model is consistent (enables undirected propagation)
- Behavior must be convergent
- Harder to set up, less efficient

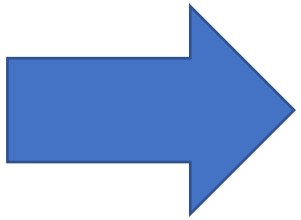
Dynamic Behaviors

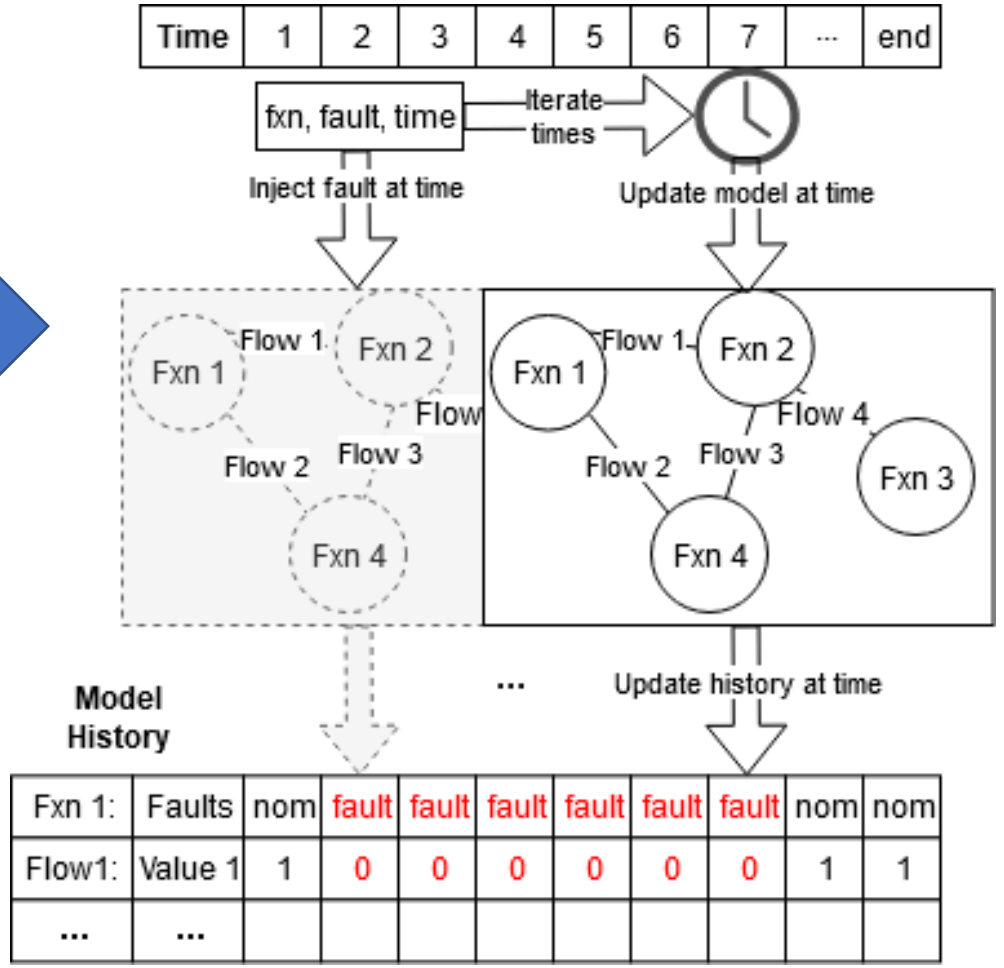


- Behaviors defined in .dynamic_behavior()
- Propagates once per timestep (directed propagation)
  - New parameter values (i.e. flows) only get passed to the next function on the same timestep
  - Otherwise, previous functions must wait until the next timestep
- Useful for quantities that accumulate (e.g., water in a tank, position of a rover)
- Over 2x more efficient

Model Simulation







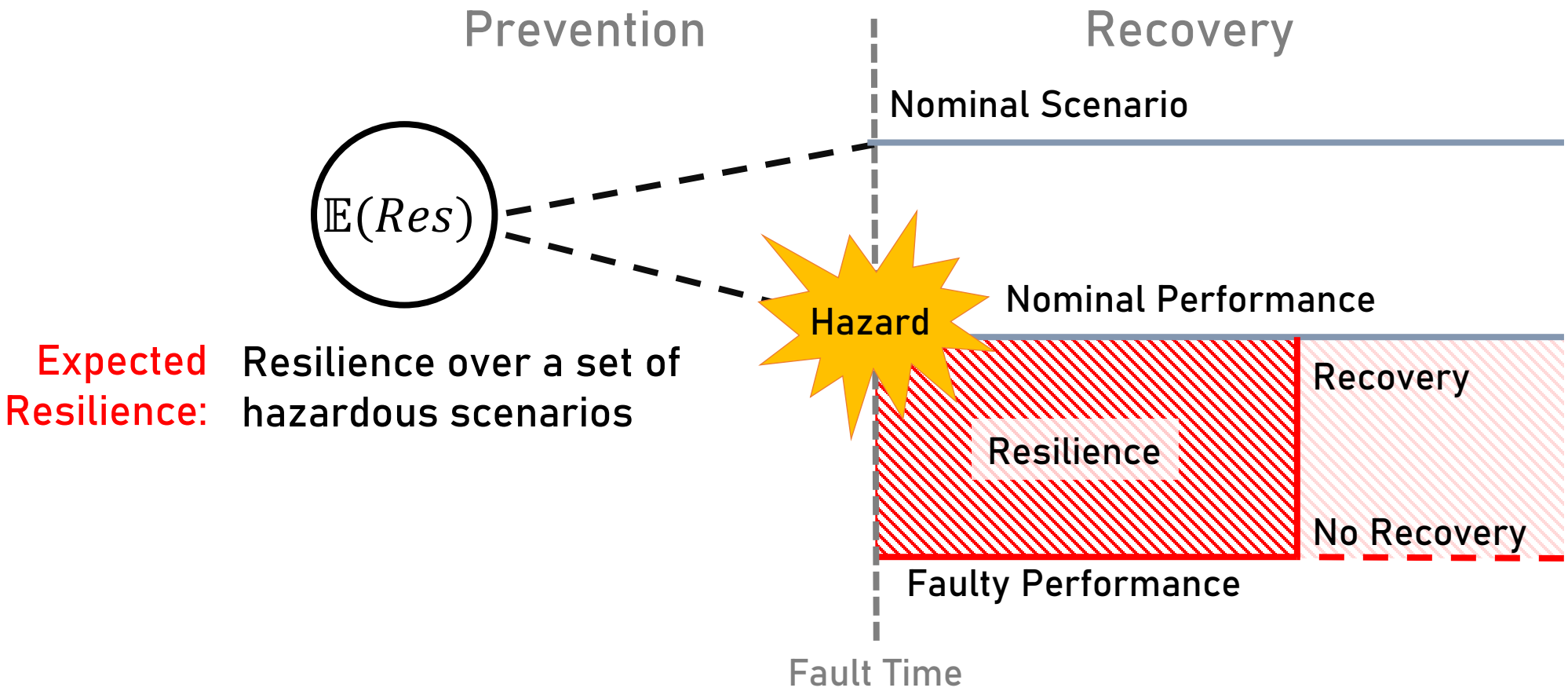
Each simulation is run over a set of time-steps defined in a model. At each timestep:

1. Dynamic behaviors are run in order (can be user-defined)
2. Static behaviors are run until converged

Activity: Open and instantiate pump model

- Review code:
  - See pump code defining the Pump model class
  - See pump code defining MoveWater function class
- Open tutorial notebook
  - Tutorial_unfilled.ipynb
- Instantiate the model
  - mdl = Pump()
- Explore
  - Try different parameters! Change things!
  - What does the model directory look like? dir(mdl)

Reminder: Goal is to Simulate Resilience



Expected Resilience: Resilience over a set of hazardous scenarios

Yodo, N, & Wang, P. (2016).

Simulation—things to consider

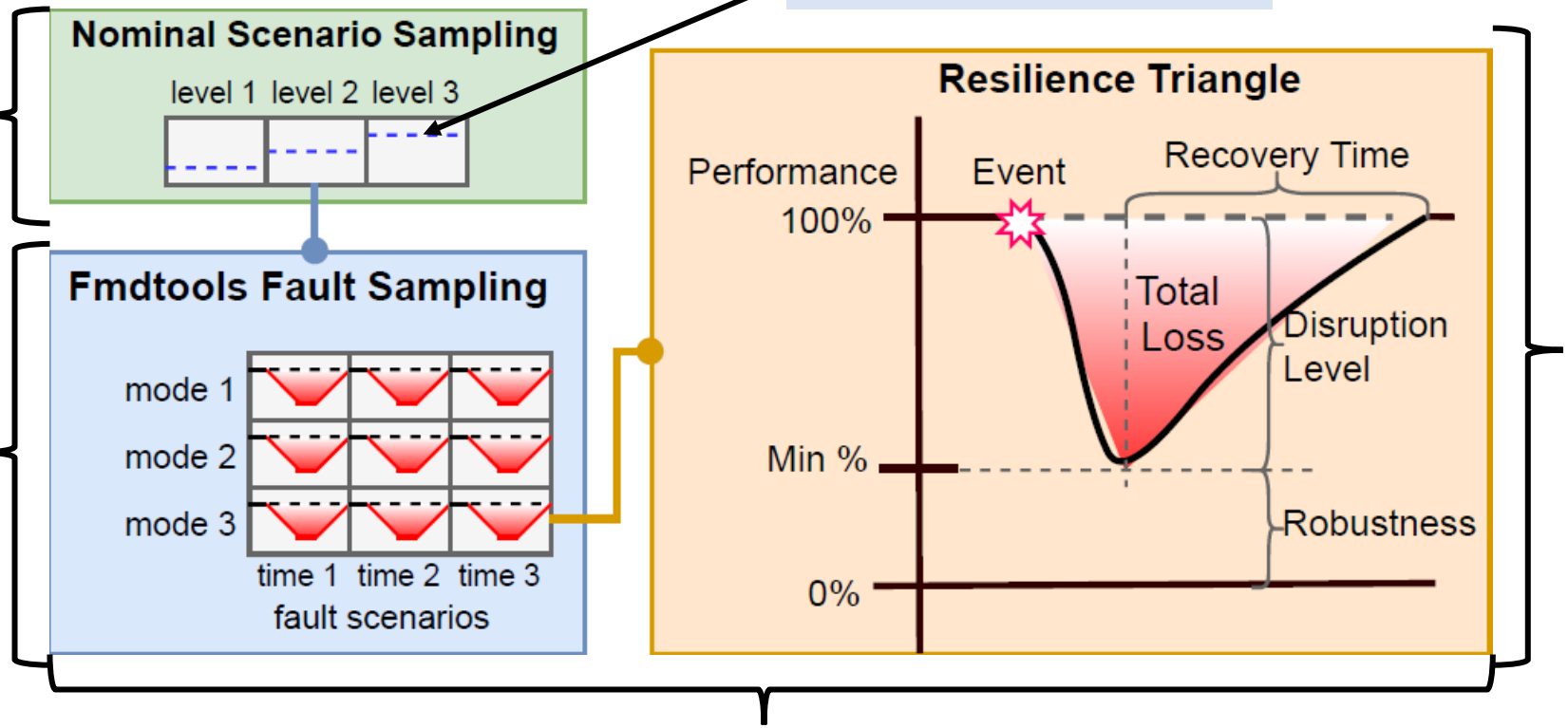
What are we trying to model:

- The nominal performance of a system?
- The performance of a system over a set of operational paremeters
- The performance in a fault scenario?
- The performance over a set of fault scenarios?
    - What kind of set?

Simulation types

Performance over a set of possible operational parameters

Resilience over a set of fault modes



Nominal performance

Resilience to a single fault or hazard

Resilience to a set of fault modes over a set of operational parameters

Behavioral Simulation Methods

- propagate.nominal(mdl)
  - Simulates system once with no fault—good for model setup
- propagate.one_fault(mdl, function, fault, time=faulttime), .propagate.mult_fault(mdl, faultseq)
  - Simulates system once with and without a fault
  - Good for verifying/visualizing model behavior under a given fault
- propagate.single_faults(mdl)
  - Simulates the list of single fault modes
  - Only use for static (single time-step) models & to gauge fault severity
- propagate.approach(mdl, app)
  - Simulates a set of fault modes over a set of times
  - Useful for expected resilience quantification
  - app defined in SampleApproach class
- propagate.nominal_approach(mdl, nomapp), propagate.nested_approach(mdl,nomapp)
  - Used for sampling the model over a range of parameters defined in a NominalApproach in nominal (nominal_approach) and faulty (nested_approach) scenarios.

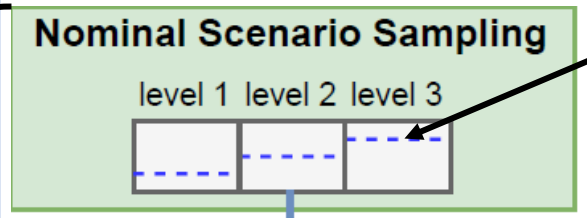
Fault propagation aspects/options

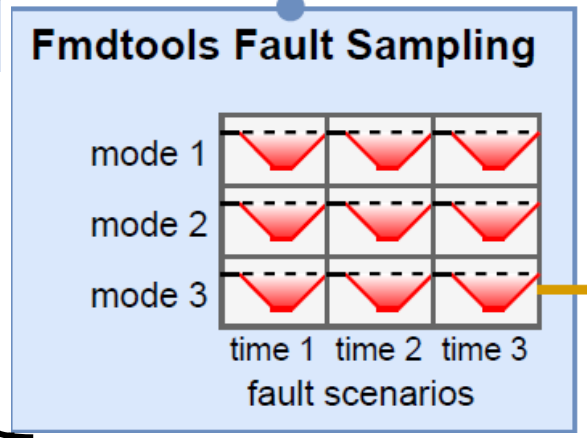
- Static/Dynamic propagation: How function states propagate to each other in a single time-step and multiple time-steps
  - Undirected graph representation—states can effect all other connected states, and vice versa, in any order
- Stochastic Propagation: Whether and how stochastic states are instantiated over time
  - e.g. do we run with the "default" values of parameters, or do we sample from a random number generator?
- Breadth of Scenarios: How hazards are represented as discrete scenarios to simulate
  - What set of joint faults do we use? How many times are sampled?
  - Operational scenarios and Joint operational/fault scenarios

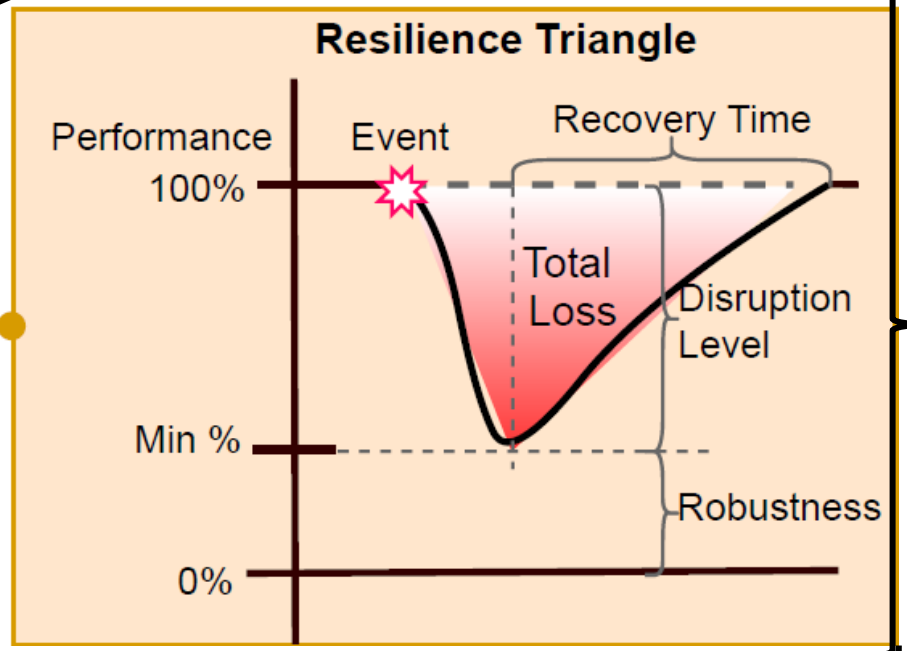
Simulation types

Performance over a set of possible operational parameters
propagate.nominal_approach()
Requires NominalApproach class

Nominal performance
propagate.nominal()







Resilience to a single fault or hazard
propagate.one_fault()
propagate.mult_fault()

Resilience over a set of fault modes
propagate.approach()
Requires SampleApproach class

Resilience to a set of fault modes over a set of operational parameters
propagate.nested_approach()
Requires NominalApproach class and options for SampleApproach

Definition of a fault sampling approach

```
app = SampleApproach(mdl,
        faults = 'all'/[list of faults]
        phases = 'all'/{fxn:phase:[s,e]}
        modephases = {fxn:mode:{phases}}
        jointfaults = {'faults':#,
                       'jointfuncs':True/False
                       'pcond': 0.##}
        defaultsamp: = {'samp': 'quad'/'fullint'/'evenspacing'...
                       'numpts': #
                       'quad': {'nodes': [] 'weights': []}
        sampparams = {(fault, mode): sampparam}
```

(faults to include in the list)

(phases of each function to include (if model has phases))

(how function modes correlate to phases)

(Number of joint-fault modes to simulate (e.g. None, 2,3,..))

(Whether function modes are mutually exclusive)

(Conditional probability for joint faults (ind. by default))

(how to represent phases)

(Number of samples to take in each phase)

(If a quadrature is used, nodes/weights

(If a different approach is to be used for each fault, sampparam follows the defaultsamp structure)

Definition of a nominal sampling approach

nomapp = NominalApproach()

Instantiate approach

(a) nomapp.add_seed_replicates(raingeid, seeds)

Construct approach as a set of random seeds to instantiate in the model

(b) nomapp.add_param_replicates(paramfunc, rangeid, replicates)

Construct approach as the output of parameters in a function over x replicates

(c) nomapp.add_param_ranges(paramfunc, rangeid, *args, **kwargs

Construct approach as the output of a parameter function a range of arguments *args, and **kwargs which may either be fixed or varied as a tuple defining the range (start, end, step)

(d) nomapp.add_rand_params(paramfunc, rangid, *fixedargs, **randvars)

Construct approach as the output of a parameter function a range fixed arguments *fixedargs and stochastic variables generated in **kwargs as (randfunc, funcparam1, funcparam2)

Activity: Propagate faults in the model

Run fault propagation methods:

- Nominal
- Fault(s)
- Approach

What do the results look like?

- endresults, resultshist, etc

Explore:

- What happens when you change SampleApproach parameters?
- What happens when you change model parameters?
- How do these methods compare in terms of computational time?

Visualization, Quantification, Processing

- Looking at a model
- Looking at modelled quantities
- Quantifying resilience
- Saving/exporting results, etc

/resultdisp

process.py

Processing of simulation results into metrics/statistics

-**hists(mdlhists)/hist(mdlhist):** processes result history for fmea/visualization

-**heatmaps(reshist),** etc: creates heatmap of respective quantities over time/runs

-**totalcost(endclasses):** calculates total expected cost

-end/overall_diff(endclasses): Difference between nominal and faulty scenarios

-percent/average/rate/… Various statistics for endclasses

plot.py

Behavioral/Statistical Plotting

-mdlhist/s/vals(mdlhist): Plots behavior of given states over time in a set of scenarios

-**samplecost(app, endclasses):** Cost and rates of a set of faults

-**costovertime(endclasses,app)**: Cost of fault modes over time

-phases(mdlphases) Function modes over time.

-nominal_vals_1d/2d/3d(): Simulation metric(s) in terms of input parameters

-nom/res_factor_comparison() Comparison of simulation statistics over given factors



graph.py

Visualization of simulation results on the model graph

-set_pos(graph): Manually positions of functions/flows in graph

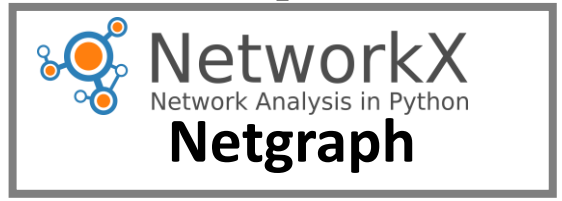
-show(mdl/graph): Show graph at end-state

-exec_order(mdl/graph) Show dynamic/static order

-history(graphhist): Shows set of graphs over history

-result(s)_from(mdl, reshist, t) Shows model state at a particular time(s)

-animation_from(mdl, reshist,t) Animates model states over given times



tabulate.py

Display and export of simulation results as tables

-**hist(mdlhist):** Gives table of the history of model states.

-**deghist(mdlhist, withstats):** Gives table of all degraded functions and flows over time

-**heatmaps(heatmap):** Displays heatmap dictionary

-**simple/phase/sum/fullfmea()**: FMEA-like assessment with faults, probabilities, and costs.

-**nominal/nested_stats():** Table of simulation statistics

-nom/res_factor_comparison() Table of simulation statistics over given factors



plot.py

plot.py

Behavioral/Statistical Plotting

-mdlhist/s/vals(mdlhist):
Plots behavior of given states over time in a set of scenarios

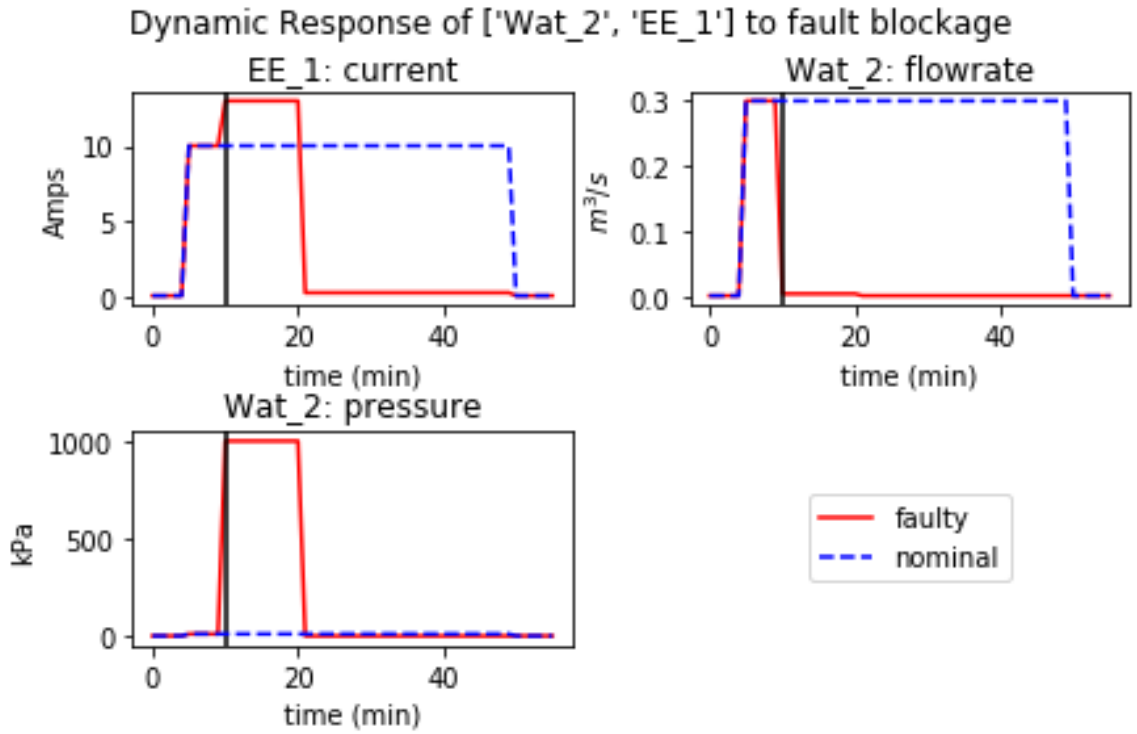
-**samplecost(app, endclasses):**
Cost and rates of a set of faults

-**costovertime(endclasses,app)**:
Cost of fault modes over time

-phases(mdlphases)
Function modes over time.

-nominal_vals_1d/2d/3d():
Simulation metric(s) in terms of input parameters

-nom/res_factor_comparison()
Comparison of simulation statistics over given factors



Example Syntax:

```
rd.plot.mdlhistvals(mdlhist, 'faultname',
                    time=faulttime,
                    fxnflowvals={'Flow1':['val1', 'val2'],'flow2':'val1'},
                    cols=2, timelabel="time (min)")
```

graph.py

plot.py

Behavioral/Statistical Plotting

-mdlhist/s/vals(mdlhist):
Plots behavior of given states over time in a set of scenarios

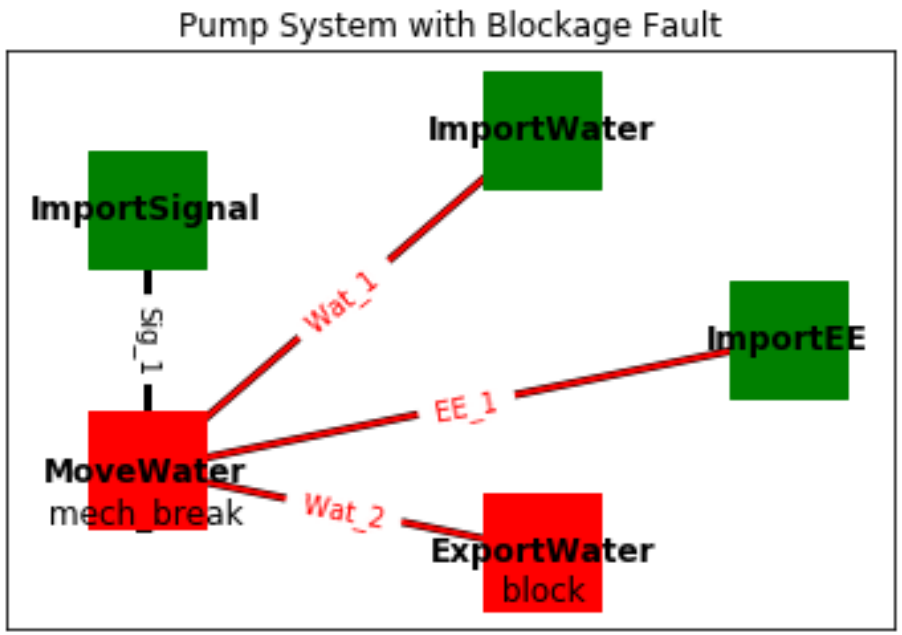
-**samplecost(app, endclasses):**
Cost and rates of a set of faults

-**costovertime(endclasses,app)**:
Cost of fault modes over time

-phases(mdlphases)
Function modes over time.

-nominal_vals_1d/2d/3d():
Simulation metric(s) in terms of input parameters

-nom/res_factor_comparison()
Comparison of simulation statistics over given factors



Example Syntaxes:

```
rd.graph.show(mdl.graph)
rd.graph.show(resgraph)
rd.graph.result_from(mdl, reshist, time,
        gtype = 'normal'/'bipartite',
        scale=2, pos=pos)
```

tabulate.py

tabulate.py

Display and export of simulation results as tables

-**hist(mdlhist):** Gives table of the history of model states.

-**deghist(mdlhist, withstats):** Gives table of all degraded functions and flows over time

-**heatmaps(heatmap):** Displays heatmap dictionary

-**simple/phase/sum/fullfmea()**: FMEA-like assessment with faults, probabilities, and costs.

-**nominal/nested_stats():** Table of simulation statistics

-nom/res_factor_comparison() Table of simulation statistics over given factors

	degraded functions	degraded flows	rate	cost	expected cost
ImportEE no_v, t=0	[ImportEE]	[EE_1, Wat_1, Wat_2]	0.000448	20125	901600
ImportEE inf_v, t=0	[ImportEE]	[EE_1, Wat_1, Wat_2]	0.000112	25125	281400
ImportWater no_wat, t=0	[ImportWater]	[EE_1, Wat_1, Wat_2]	0.00056	11125	623000
ImportSignal no_sig, t=0	[ImportSignal]	[EE_1, Sig_1, Wat_1, Wat_2]	5.6e-05	20125	112700
MoveWater mech_break, t=0	[MoveWater]	[EE_1, Wat_1, Wat_2]	0.000336	15125	508200
MoveWater short, t=0	[ImportEE, MoveWater]	[EE_1, Wat_1, Wat_2]	0.00056	30125	1.687e+06
ExportWater block, t=0	[MoveWater, ExportWater]	[EE_1, Wat_1, Wat_2]	0.00056	20100.3	1.12561e+06
ImportEE no_v, t=20	[ImportEE]	[EE_1, Wat_1, Wat_2]	0.000448	16750	750400
ImportEE inf_v, t=20	[ImportEE]	[EE_1, Wat_1, Wat_2]	0.000112	21750	243600
ImportWater no_wat, t=20	[ImportWater]	[EE_1, Wat_1, Wat_2]	0.00056	7750	434000

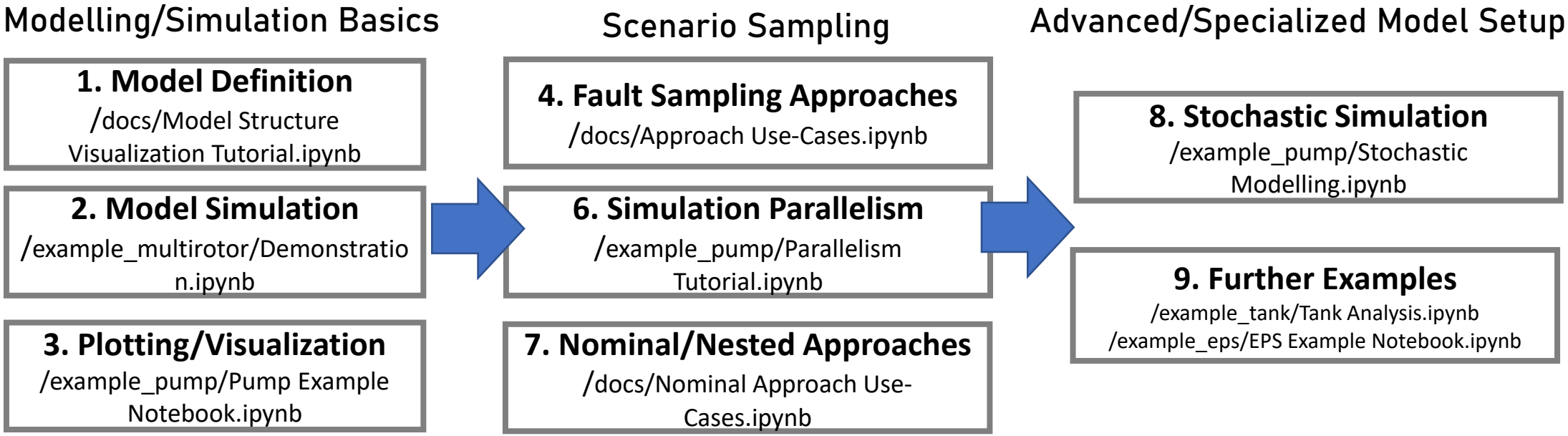
Example Syntaxes:

```
reshists, diffs, summaries = rd.process.hists(mdlhists)
fullfmea = rd.tabulate.fullfmea(endclasses, summaries)
phasefmea = rd.tabulate.phasefmea(endclasses, app)
```

Activity: Visualize the results

- Visualize the results
  - Show model graph
  - Show nominal performances
  - Show performances in a nominal scenario
  - Make a scenario-based fmea
- Explore
  - How can you show only the parameters you want? Or change the formatting?
  - What does the behavior under other faults look like?
  - What other analyses can you perform with these results?

Further Reading



Explanatory Papers

- D. Hulse, H. Walsh, A. Dong, C. Hoyle, I.Y. Tumer, C. Kulkarni, K. Goebel, "fmdtools: A Fault Propagation Toolkit for Resilience Assessment in Early Design" IJPHM. Submitted
- D. Hulse, C. Hoyle, I.Y. Tumer, K. Goebel, C. Kulkarni, "Temporal Fault Injection Considerations in Resilience Quantification." ASME IDETC/CIE 2020, Design Automation Conference. IDETC2020-19287.