Intro to resilience modeling, simulation, and visualization in Python with fmdtools.

Author: Daniel Hulse

Version: 2.2.0

Overview

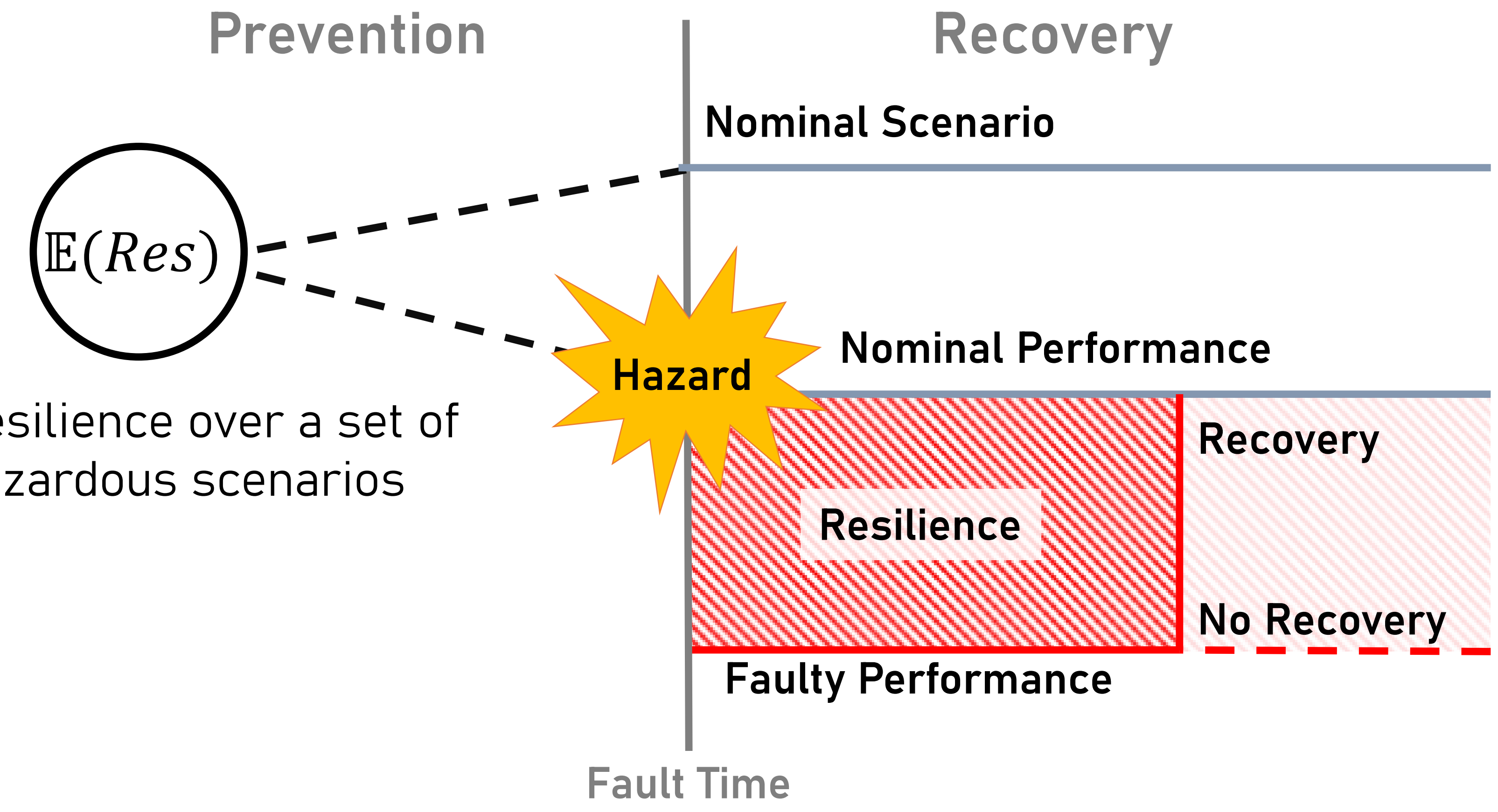
- **Overview of fmdtools**
  - Purpose
  - Project Structure
  - Common Classes/Functions
  - Basic Syntax
- **Coding Activity**
  - Example model: `examples/pump/ex_pump.py`
  - Workbook: `examples/pump/Tutorial_unfilled.ipynb`
    - Model Instantiation
    - Simulation
    - Visualization/Analysis

Prerequisites

- Ideally, some pre-existing Python and Git knowledge
- Python distribution (anaconda or uv)
  - Ideally this is already set up!
  - Download/install from:
    - https://www.anaconda.com/products/individual
    - https://github.com/astral-sh/uv
- A git interface
  - Github Desktop (graphical git environment)
  - git-scm (stand-alone CLI)

Motivation: Modeling System Resilience

Resilience means taking a **dynamic understanding of risk and safety**

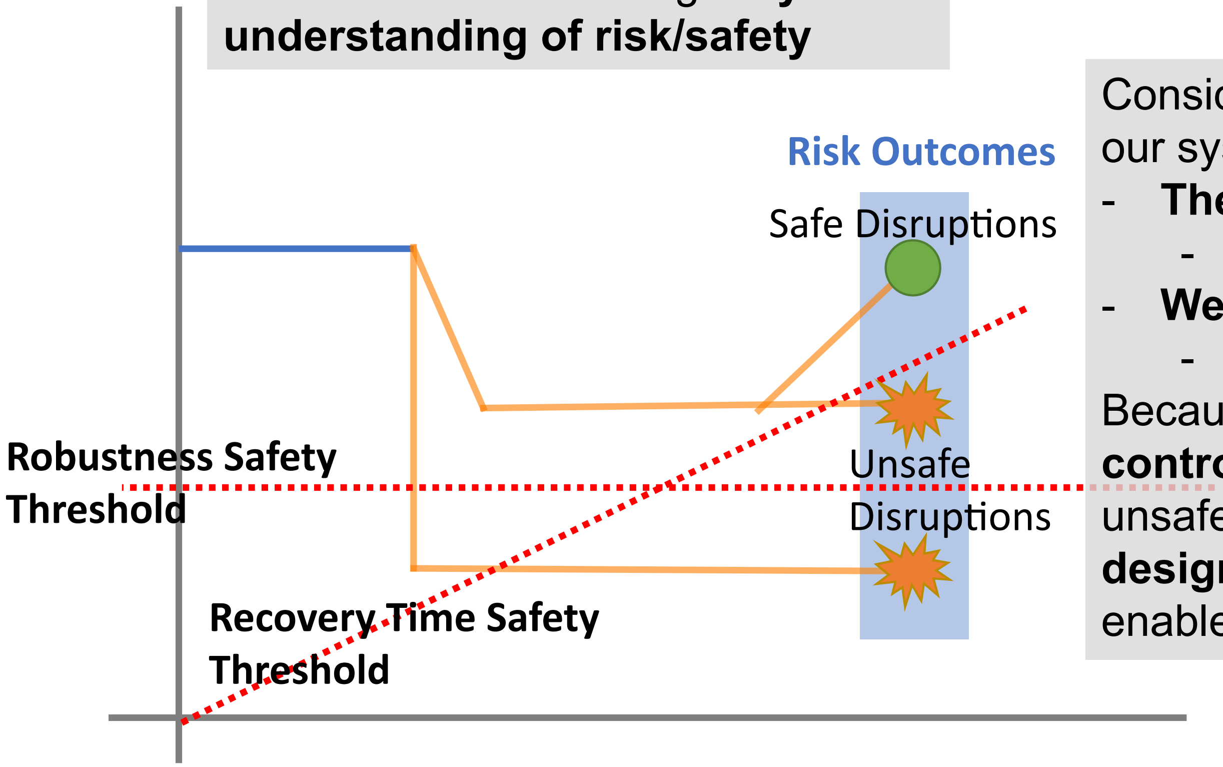


Expected Resilience: Resilience over a set of hazardous scenarios

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

Why is Resilience Important?

Resilience means taking a **dynamic understanding of risk/safety**



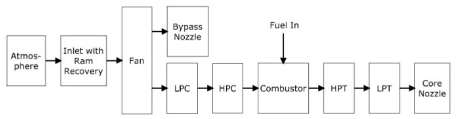
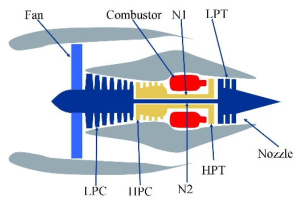
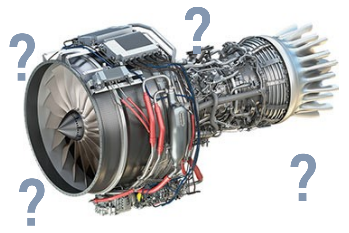
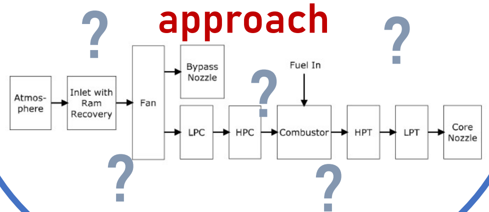
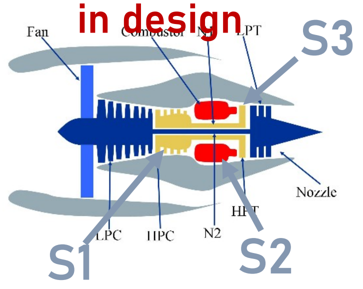
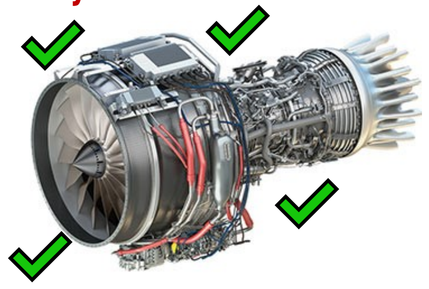
Considering resilience is important when our system has dynamic attributes, e.g.:

- **The system state changes over time**
  - (e.g., position, velocity, etc)
- **We can control this state**
  - (e.g., operators, autopilot)

Because we can use it to determine **how to control the system to a safe outcome** in unsafe circumstances and what **design/operational features** we need to enable this control

Enabling a proactive design process - especially when we don't have data

Idea: the system should be **resilient-by-design**

	Concept Design	Embodiment Design	Implementation
Reactive Design	Choose concept  Atmos-phere Inlet with Ram Recovery Fan Bypass Nozzle Fuel In LPC HPC Combustor HPT LPT Core Nozzle	Design system  Fan Combustor N1 LPT Nozzle HPT LPC HPC N2	Retrofit for resilience 
Proactive Design	Establish resilience approach  ? ? ? Atmos-phere Inlet with Ram Recovery Fan Bypass Nozzle Fuel In LPC HPC Combustor HPT LPT Core Nozzle ? ?	Integrate resilient features in design  Fan Combustor N1 LPT S3 Nozzle HPT LPC HPC N2 S1 S2	Verify resilient function 

Why fmdtools? Possible Competitors:

- Uncertainty Quantification tools: (e.g. OpenCossan)
  - Does not incorporate fault modeling/propagation/visualization aspects
- MATLAB/modelica/etc. Fault Simulation tools
  - Rely on pre-existing model/software stack--Useful, but often difficult to hack/extend (**not open-source**)
- Safety Assessment tools: (e.g. Alyrica, Hip-Hops)
  - Focused on quantifying safety, not necessarily resilience
  - As a result, use **different model formalisms**!

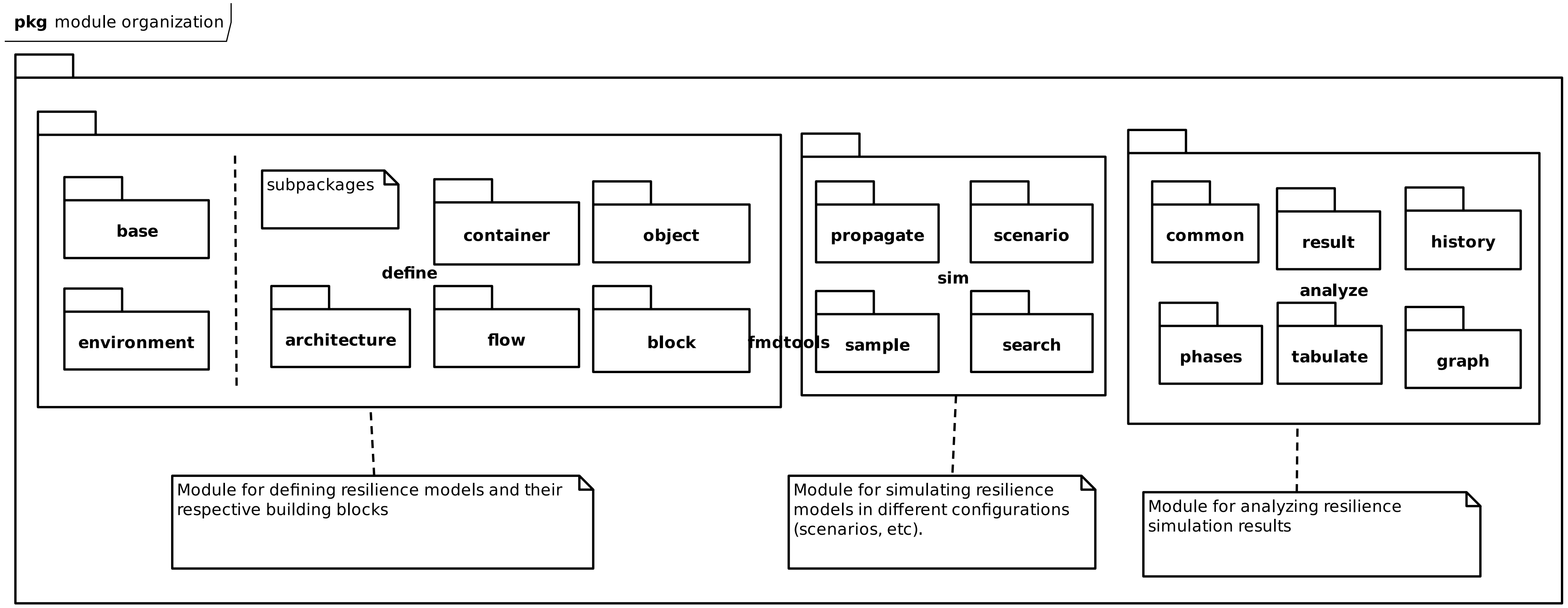
Why fmdtools? Pros:

- Highly Expressive, modular model representation.
  - faults from any component can propagate to any other connected component
  - highly-extensible code-based behavior representation
  - modularity enables **complex models** and modelling use-cases
- Research-oriented:
  - Written in/relies on the Python stack
  - Open source/free software
- Enables design:
  - Models can be parameterized and optimized!
  - Plug-and-play analyses and visualizations

Why not fmdtools? Cons:

- You already have a pre-existing system model
    - fmdtools models are built in fmdtools
    - if you have a simulink/modelica model, you may just want to use built-in tools
- You want to use this in production
    - fmdtools is Class E Software and thus mainly suitable for research (or, at least, we don't guarantee it)
    - Somewhat dynamic development history

What is fmdtools? A Python package for design, simulation, and analysis of resilience.


pkg module organization
fmdtools
define
base
environment
subpackages
container
object
architecture
flow
block
sim
propagate
scenario
sample
search
analyze
common
result
history
phases
tabulate
graph
Module for defining resilience models and their respective building blocks
Module for simulating resilience models in different configurations (scenarios, etc).
Module for analyzing resilience simulation results

What is fmdtools? Repo Structure

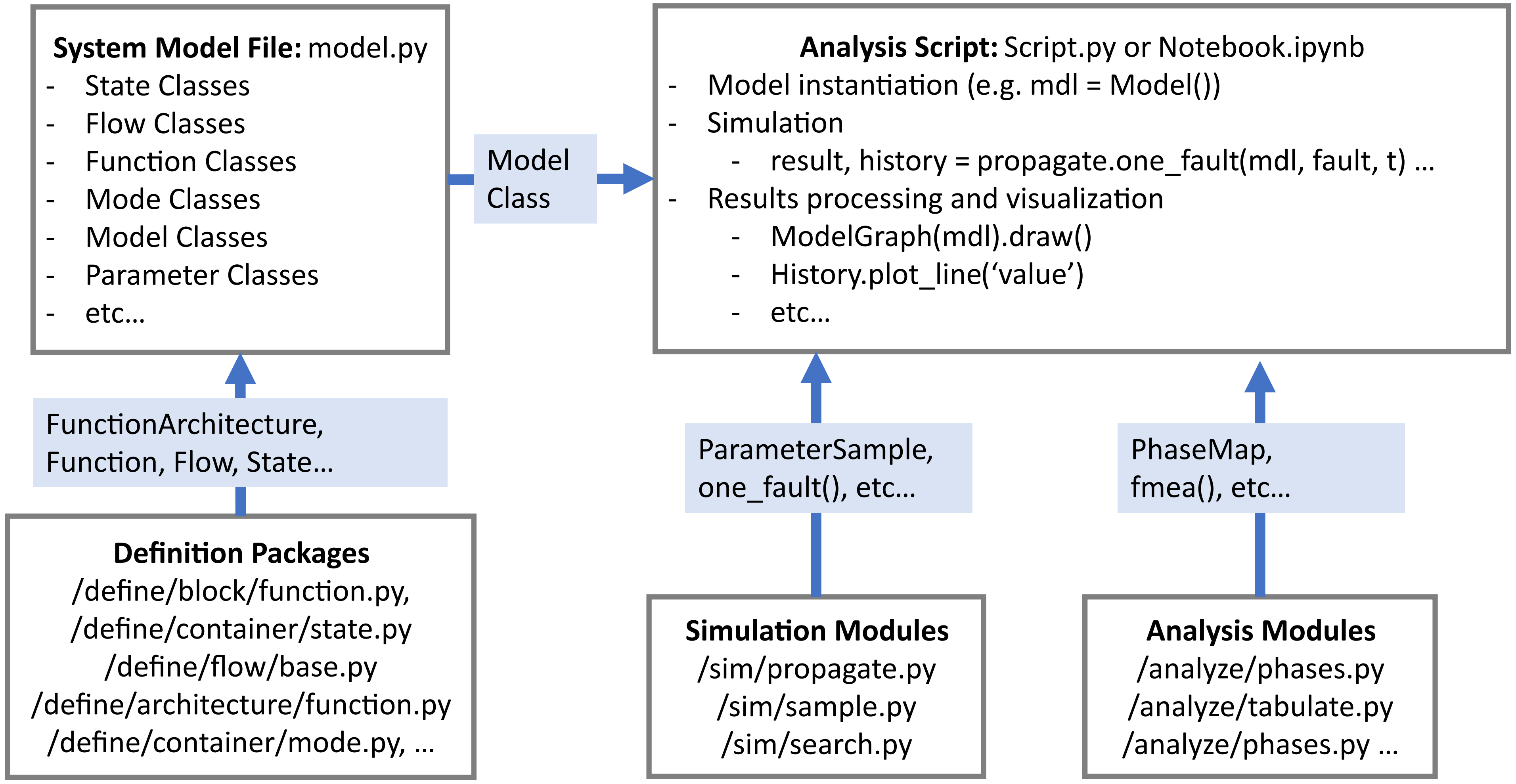
[Repository] (https://github.com/nasa/fmdtools/)

- `/fmdtools` : installable package directory
- `/examples` : example models with demonstrative notebooks and tests
- `/docs` : HTML Documentation (source files at `/docs-source`)
- `/tests` : stand-alone tests (and testing rigs)
- Basic information: `README.md` , `CONTRIBUTORS.md` , `PUBLICATIONS.md` , `LICENSE` , `fmdtools_Individual_CLA.pdf` , etc.
- Config/test files: `requirements.txt` , `pyproject.toml` , `conf.py` , `index.rst` , etc.

Activity: Download and Install fmdtools

- repo link: https://github.com/nasa/fmdtools/
- set up repo:
  - create `path/to/fmdtools` folder for repo
    - (usually in `/documents/GitHub`)
  - clone git into folder:
    - `git clone https://github.com/nasa/fmdtools.git`
    - can also use webpage
- package installation:
  - anaconda: Open Python from anaconda (e.g., open Spyder) and install with `pip install -e /path/to/fmdtools`
  - uv: run `uv pip install .` from fmdtools repository

Analysis Workflow/Structure



System Model File: model.py
- State Classes
- Flow Classes
- Function Classes
- Mode Classes
- Model Classes
- Parameter Classes
- etc…

Model Class

Analysis Script: Script.py or Notebook.ipynb
- Model instantiation (e.g. mdl = Model())
- Simulation
  - result, history = propagate.one_fault(mdl, fault, t) …
- Results processing and visualization
  - ModelGraph(mdl).draw()
  - History.plot_line(‘value’)
  - etc…

FunctionArchitecture, Function, Flow, State…

ParameterSample, one_fault(), etc…

PhaseMap, fmea(), etc…

Definition Packages
/define/block/function.py,
/define/container/state.py
/define/flow/base.py
/define/architecture/function.py
/define/container/mode.py, …

Simulation Modules
/sim/propagate.py
/sim/sample.py
/sim/search.py

Analysis Modules
/analyze/phases.py
/analyze/tabulate.py
/analyze/phases.py …

Defining a Model

- What do we want out of a model?
  - What behaviors and how much fidelity do we need?
  - What **functions/components** and interactions make up the system?
    - One function or multiple functions?
    - Is it a **controlled system**? Are there multiple **agents**?
- What type of simulation do we want to run?
  - Single-timestep vs multi-timestep vs network
- What scenarios do we want to study and how?
  - **Failure modes** and faulty behaviors
  - **Disturbances** and changes in parameters
  - What are the possible effects of hazards and how bad are they?
    - By what **metrics**?

Defining a Model

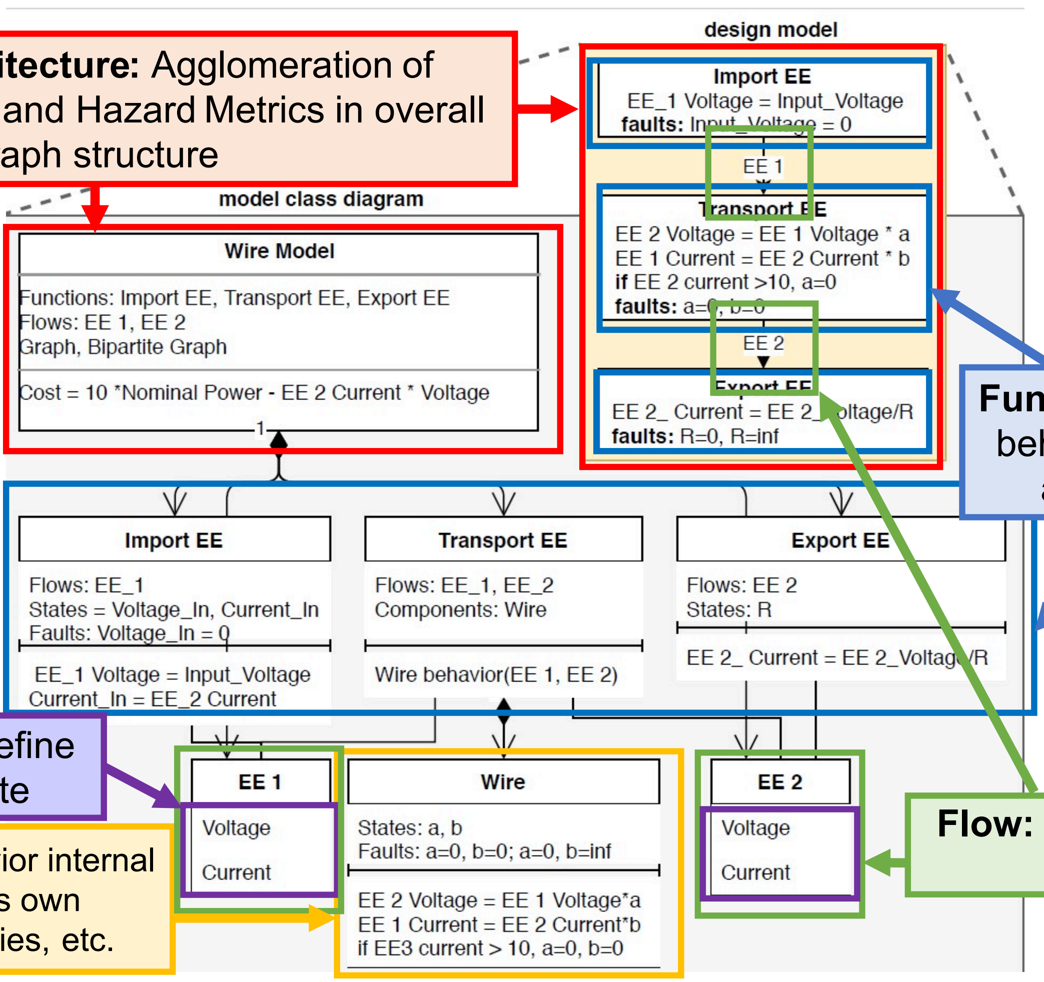
FunctionArchitecture: Agglomeration of Functions, Flows, and Hazard Metrics in overall graph structure

Function: Defines high-level system behaviors as well as failure modes and component architectures

State: Entries that define Function/Flow state

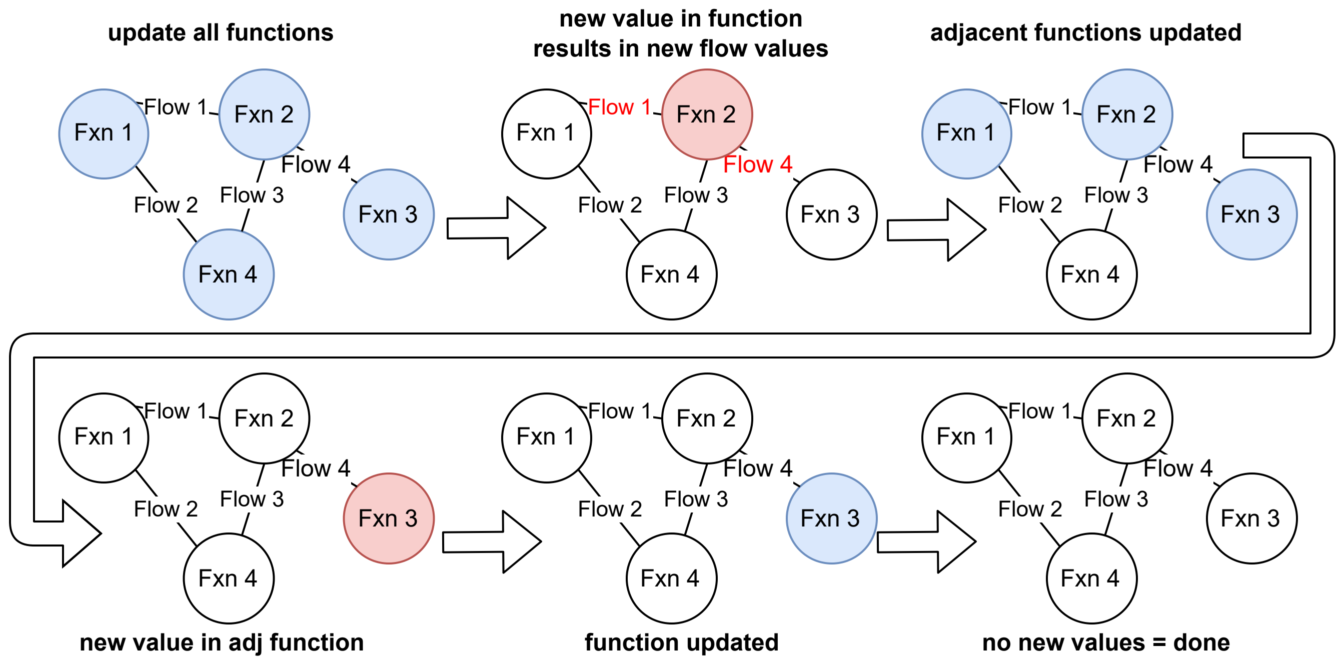
Component: Sub-behavior internal to a Function with its own behaviors and properties, etc.

Flow: Data Structures (inputs/outputs) that connect Functions



design model
Import EE
EE_1 Voltage = Input_Voltage
faults: Input_Voltage = 0
EE 1
Transport EE
EE 2 Voltage = EE 1 Voltage * a
EE 1 Current = EE 2 Current * b
if EE 2 current >10, a=0
faults: a=0, b=0
EE 2
Export EE
EE 2_ Current = EE 2_Voltage/R
faults: R=0, R=inf
model class diagram
Wire Model
Functions: Import EE, Transport EE, Export EE
Flows: EE 1, EE 2
Graph, Bipartite Graph
Cost = 10 *Nominal Power - EE 2 Current * Voltage
1
Import EE
Flows: EE_1
States = Voltage_In, Current_In
Faults: Voltage_In = 0
EE_1 Voltage = Input_Voltage
Current_In = EE_2 Current
Transport EE
Flows: EE_1, EE_2
Components: Wire
Wire behavior(EE 1, EE 2)
Export EE
Flows: EE 2
States: R
EE 2_ Current = EE 2_Voltage/R
EE 1
Voltage
Current
Wire
States: a, b
Faults: a=0, b=0; a=0, b=inf
EE 2 Voltage = EE 1 Voltage*a
EE 1 Current = EE 2 Current*b
if EE3 current > 10, a=0, b=0
EE 2
Voltage
Current

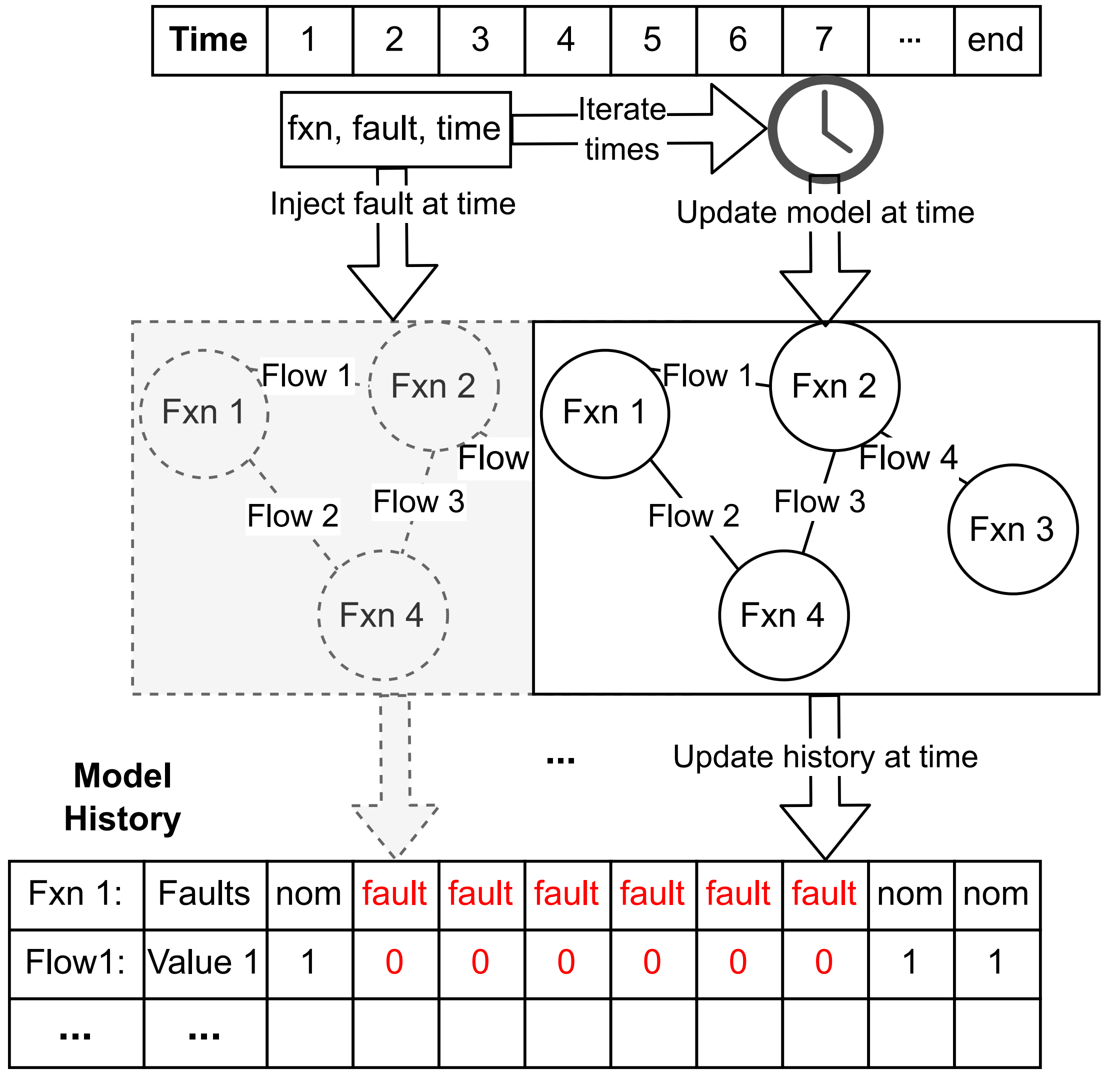
Concept: Static Propagation



In a single timestep, functions with `static_behavior()` methods simulate until behaviors converge (i.e., no new state values)

Concept: Propagation over Time

- Model increments (simulated + history updated) over each time-step until a **defined final time-step** or **specified indicator returns true**.
- Functions with `dynamic_behavior()` run once in defined order



Fxn 1:	Faults	nom	fault	fault	fault	fault	fault	fault	nom	nom
Flow1:	Value 1	1	0	0	0	0	0	0	1	1
...	...									

Containers - The building blocks of simulations

State classes are used to represent variables (called fields) that change over time

```
class StateName(State):
    varname1: float = 1.0
    varname2: str = “default_value”
```

Field name

Field type or class

Default initial value

Parameter classes are used to represent variables that don’t change over time, with similar syntax to States

```
class ParameterName(Parameter):
    varname1: float = 1.0
    varname1_range = (0.0, 10.0)
    varname2 : str = “default_value”
    varname2_set = (“default_value”, “other_possible_value”)
```

varname1_range defines allowable range for the variable varname1

varname2_set defines allowable values for the field varname2

Mode classes are used to represent modes (faults and operational modes) that could occur in the system

```
class ModeName(Mode):
    fault_faultname1 = (0.001, 200.0),
    fault_faultname2 = (0.00001, 100.0, {‘on’: 1.0})
    opermodes = (“off”, “on”)
    mode: str = “off”
```

Dictionary of fault names and their optional properties (rate, repair cost, phases they could occur and their rates)

List of potential operational modes (if multiple)

Default mode (if multiple modes/not nominal)

Flow Code Template

```
class FlowName(Flow):
    container_s = StateName
    container_p = ParameterName
    default_track = [‘s’,’m’]
  def indicate_XXX(self):
        Conditional statement (e.g., self.s.state>threshold) which
is logged in the history and may be used to terminate
simulations
```

Specifies which container classes play specific Flow **roles** (e.g., s corresponds to the state role, p corresponds to a parameter role, m corresponds to the mode role, etc)

Specifies **what should be tracked** in the FxnBlock history (fxnname.h) by default. May be a dict ({role:value}), list ([role1, role2]), or string (“all”,”none”, etc). Overwritten by track parameter at model instantiation.

These methods define Flow **indicators** and are called/tracked during simulation (in flow.i)

- Flows represent connections or shared variables between different functions. Think of them as Function inputs/outputs.
- Flows are build from container classes like states, along with their own methods/variables.

Function Code Template

flow_XXX is used to append a **flow** of given type that is named XXX to the function class. If the flow(s) has a different name outside the function (optional), flownames matches the external name to the internal name

Specifies **what should be tracked** in the FxnBlock history (fxnname.h) by default. May be a dict ({role:value}), list ([role1, role2]), or string (“all”,”none”, etc). Overwritten by the track parameter during model instantiation.

These methods define the **behavior** of the FxnBlock and thus simulate at each time-step of the simulation.

These methods define FxnBlock **indicators** and are called/tracked during simulation (in fxnname.i)

This method defines the Result to be returned **when simulated individually**

```
class FunctionName(Function):
    container_s = FunctionState
    container_m = FunctionMode
    container_t = FunctionTime
    flow_flowname1 = FlowClass1
    flownames = {“outsideflowname”:”flowname1”}
    default_sp = {‘end_time’: 100}
    default_track = [‘s’,’m’]
    def init_block(self, **kwargs):
        <e.g., self.s.x = 2.0>
    def static_behavior(self):
        Runs only in static propagation steps
    def dynamic_behavior(self):
        Runs only in dynamic propagation steps
def indicate_XXX(self):
        Conditional statement (e.g., self.s.state>threshold) which is logged in
the history and may be used to terminate simulations
    def find_classification(self, scen, mdlhists):
        Returns a Result dictionary (calculated at completion)
```

Specifies which classes play specific FxnBlock **roles** (e.g., s corresponds to the state role, m corresponds to the mode role, etc)

Default **keyword arguments for SimParam**. Only necessary when the functionblock will be simulated individually.

Optional method to call to set up FxnBlock in ways not already defined by role initalization (e.g., attaching local MultiFlows or setting initial values for States from Parameter)

Function Architecture Code Template

```
class ArchitectureName(FunctionArchitecture):
   container_p= ModelParam
   default_sp = {'end_time': 100}
   default_track = ["fxns", "flows]

   def init_architecture(self, **kwargs):
      self.add_flow("flowname", FlowClass)
      …
      self.add_fxn("functionname", FunctionClass, "flowname1", "flowname2")
      …

def indicate_XXX(self):
      Conditional statement
      (e.g., self.fxns["functionname"].s.state>threshold)
 which is logged in the history and may be used to terminate simulations

def find_classification(self, scen, mdlhists):
      Returns a Result dictionary (calculated at completion)
```

Default **keyword arguments for SimParam**. Defines max time of the simulation, along with phases, timestep, units, etc.

Points to a Parameter representing immutable model characteristics instantiated at the start of the simulation

Specifies **what should be tracked** in the Model history by default. May be a dict ({role:value}), list ([role1, role2]), or string ("all","none", etc). Overwritten by the track option in propagate.

Method to **instantiate the model and define its structure:**,

- **.add_flow** is used to instantiate a flow
- **.add_fxn** is used to instantiate a function and attach connected flows

These methods define Model **indicators** and are called/tracked during simulation (in modelname.i)

This method defines the Result to be returned by the model.

Demo Model Activity: examples/pump/ex_pump.py

Notice the definitions and structure:

- **States**: `WaterStates`, `EEStates`, `SignalStates`
- **Flows**: `Water`, `EE`, `Signal`
- **Functions**: `ImportEE`, `ImportWater`, `ExportWater`, `MoveWater`, `ImportSignal`
  - **Modes** (e.g., `ImportEEMode`, `ImportSigMode`)
    - Mode probability model
    - Actual modes in `fm_args` entry
  - others attributes, e.g., `Timer`
- **Model**: `Pump` connects functions, flows, and defines `end_classification`
- **Parameter**: `PumpParam` defines values we can change in the simulation

More Resources for Model Definition

- Note the docs for model definition are in https://nasa.github.io/fmdtools/docs-source/fmdtools.define.html
- Other examples also can be helpful: https://nasa.github.io/fmdtools/examples/Examples.html

Notebook Activity:

Open `/examples/pump/Tutorial_unfilled.ipynb`:

- Instantiate the model
  - `mdl = Pump()`
- Explore structure
  - Try different parameters!
  - Change things!

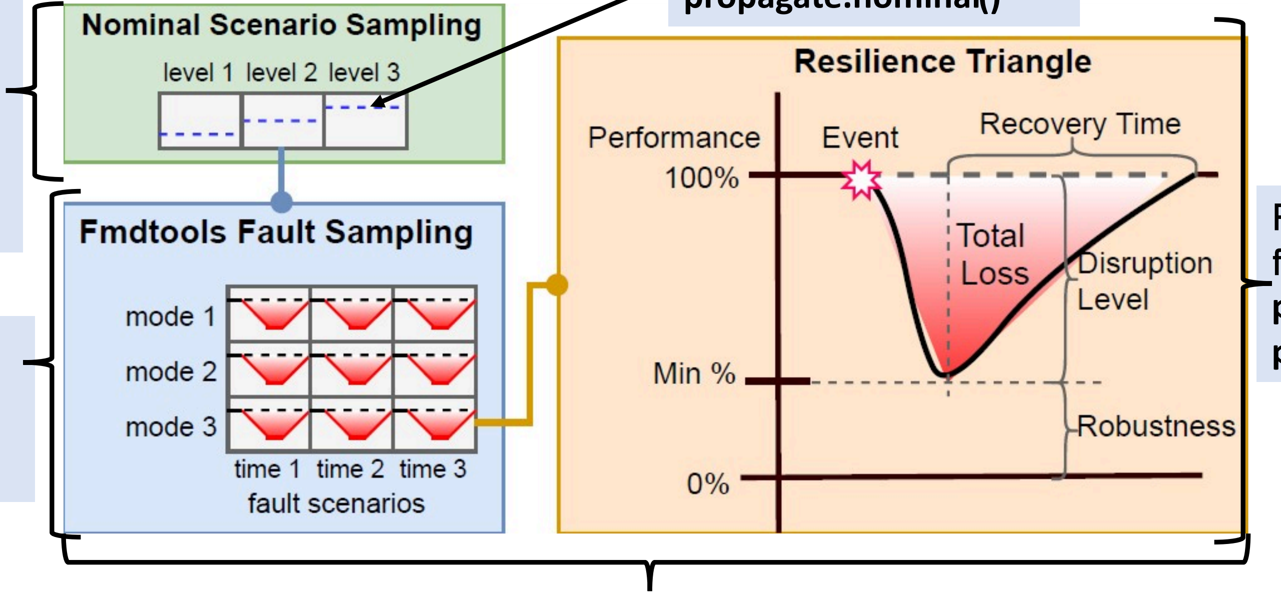
    What does the model directory look like?
  - `dir(mdl)`

Simulation Concepts: Types of Simulations

Nominal performance **propagate.nominal()**

Performance over a set of possible operational parameters **propagate. parameter_sample()**

Resilience over a set of fault modes **propagate. fault_sample()**



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

Resilience to a set of fault modes over a set of operational parameters **propagate.nested_sample()**

For more info on syntax/arguments, see documentation for `fmdtools.sim.propagate`.

Simulation Concepts: Sampling Approaches

These classes define **multi-run simulations** which can be used to quantify uncertain performance/resiliences:

- **SampleApproach/FaultSample**: Which faults to sample and when
    - Relies on **mode** information encoded in the model
    - Simulated using `propagate.fault_sample()`
- **ParameterSample**: Nominal parameters or random seeds to sample
    - Can be simulated in `propagate.parameter_sample()`
    - Can be simulated in conjunction with faults using `propagate.nested_sample`

See docs for: `fmdtools.sim.fault_sample`

Simulation Concepts: Things to Consider

Static/Dynamic propagation: How function states propagate to each other in a single time-step and multiple time-steps?

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

Activity: Simulate the Model

Run fault propagation methods:

- `propagate.nominal()`, `propagate.one_fault()`, `propagate.fault_sample()`

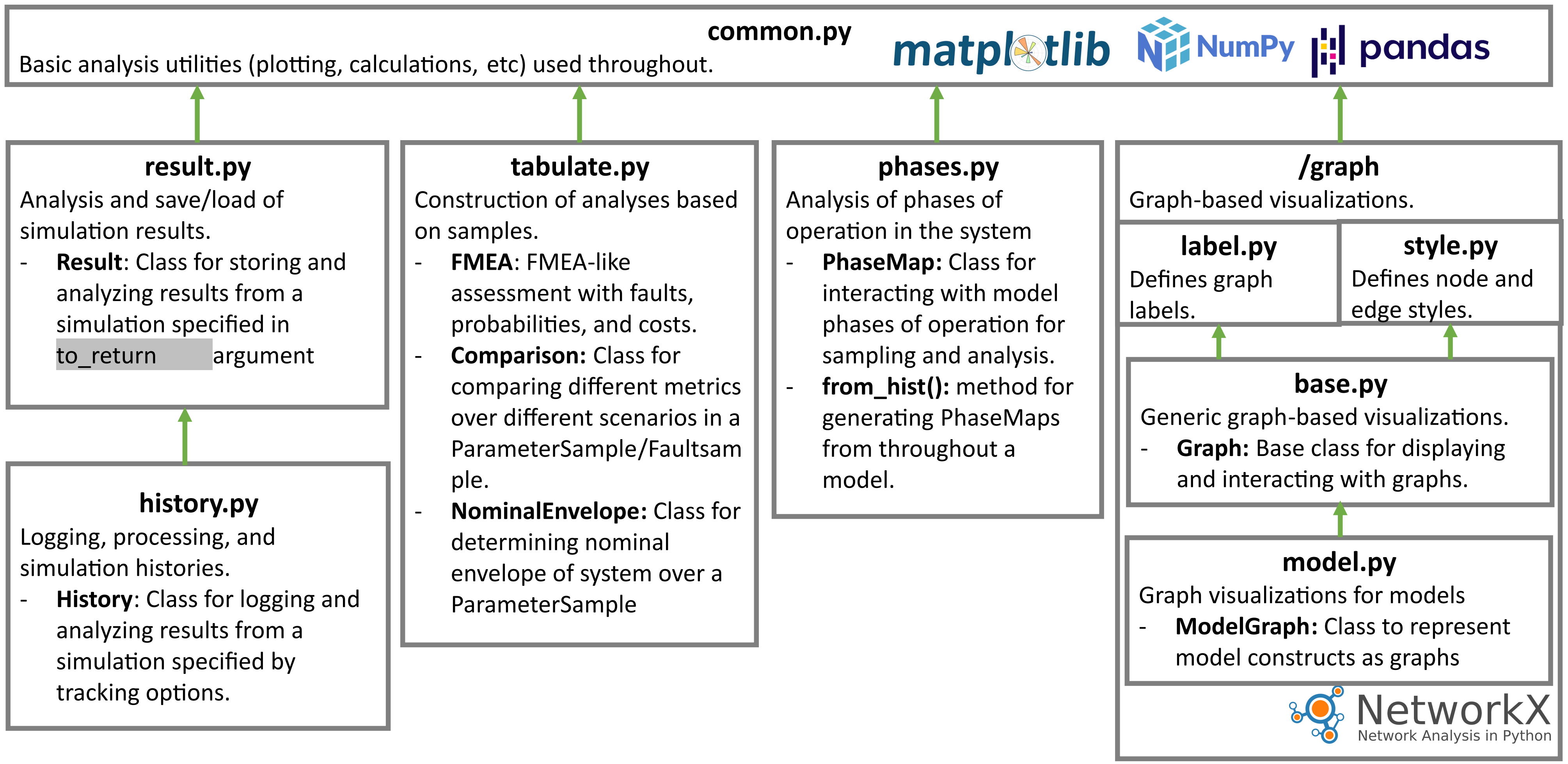
What do the results look like? Explore data structures:

- `analyze.result.Result`, `analyze.result.History`

Explore:

- What happens when you change `FaultSample` parameters?
- What happens when you change `Model` parameters?
- How do these methods compare in terms of computational time?

Analysis Modules - see docs for `fmdtools.analyze`



common.py
Basic analysis utilities (plotting, calculations, etc) used throughout.

result.py
Analysis and save/load of simulation results.
- **Result**: Class for storing and analyzing results from a simulation specified in to_return argument

history.py
Logging, processing, and simulation histories.
- **History**: Class for logging and analyzing results from a simulation specified by tracking options.

tabulate.py
Construction of analyses based on samples.
- **FMEA**: FMEA-like assessment with faults, probabilities, and costs.
- **Comparison:** Class for comparing different metrics over different scenarios in a ParameterSample/Faultsample.
- **NominalEnvelope:** Class for determining nominal envelope of system over a ParameterSample

phases.py
Analysis of phases of operation in the system
- **PhaseMap:** Class for interacting with model phases of operation for sampling and analysis.
- **from_hist():** method for generating PhaseMaps from throughout a model.

/graph
Graph-based visualizations.

label.py
Defines graph labels.

style.py
Defines node and edge styles.

base.py
Generic graph-based visualizations.
- **Graph:** Base class for displaying and interacting with graphs.

model.py
Graph visualizations for models
- **ModelGraph:** Class to represent model constructs as graphs

Analysis Activity

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?

Conclusions/Summary

- **fmdtools** is an environment for designing resilient systems
    - `/define` enables model definition
    - `/sim` is used to define simulations
    - `/analyze` is used to analyze and visualize simulation results
- I hope you agree that it has some powerful features!
    - Modeling expressiveness and clarity
    - Types of simulations that can be run
    - Powerful but easy-to-leverage plug-and-play analyses

Further Reading/Links

- More advanced topics (see examples), including Search and optimization, Human, Systems-of-Systems modeling, and Modeling Stochastic Behavior
- Model Development Guide: Has best practices for developing models in a strategic way (especially helpful for complex models)
- Overview Paper:
  - Hulse, D., Walsh, H., Dong, A., Hoyle, C., Tumer, I., Kulkarni, C., & Goebel, K. (2021). fmdtools: A fault propagation toolkit for resilience assessment in early design. International Journal of Prognostics and Health Management, 12(3).