fmdtools Tutorial

This tutorial notebook will show some of the basic commands needed to perform resilience analysis in fmdtools.

For some context, it may be helpful to look through the accompanying presentation. This notebook uses the model defined in `ex_pump.py`. In this notebook, we will:

- Load an environment and model
- Simulate the system in nominal and faulty scenarios
- Visualize and quantify the results

1.) Loading the environment and model

To load the `fmdtools` environment, we have to import it.

Since we're in a subfolder of the repository, we need to add the fmdtools folder to the system path. (This would be unnecessary if we installed using `pip`)

In [1]:
```
#First, import the fault propogation library as well as the model
#since the package is in a parallel location to examples...
import sys, os
sys.path.insert(1,os.path.join(".."))
```

There are a number of different syntaxes for importing modules. Because of the long names of the module trees, it is often helpful to load the modules individually and abbreviate (e.g. `import fmdtools.faultsim.propagate as propagate`). Below, import the propagate `fmdtools.faultsim.propagate` and `fmdtools.resultdisp` modules, as well as the `SampleApproach` from `fmdtools.modeldef`

In [2]:
```
import fmdtools.faultsim.propagate as propagate
import fmdtools.resultdisp as rd
from fmdtools.modeldef import SampleApproach
```

In [6]:
```
import fmdtools
fmdtools
```

Out[6]:
```
<module 'fmdtools' from 'C:\\Users\\dhulse\\Documents\\GitHub\\fmdtools_public
\\example_pump\\..\\fmdtools\\__init__.py'>
```

Now, import the Pump class defined in the ex_pump module.

In [3]:
```
from ex_pump import Pump
```

We can then use that to instantiate a model object. See:

In [4]:
```
mdl = Pump()
```

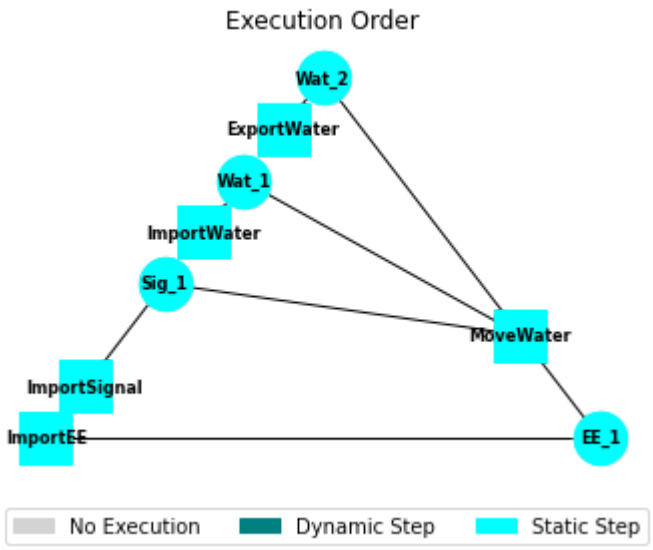
To get started, it can be helpful to view some of the aspects of the model. Try `dir(mdl)`, `mdl.fxns`, `mdl.flows`, `mdl.graph`, etc.

In [5]:
```
mdl.fxns
```

Out[5]:
```
{'ImportEE': ImportEE ImportEE function: ({'mode': 'nom'}, {'nom'}),
 'ImportWater': ImportWater ImportWater function: ({'mode': 'nom'}, {'nom'}),
 'ImportSignal': ImportSignal ImportSig function: ({'mode': 'nom'}, {'nom'}),
 'MoveWater': MoveWater MoveWat function: ({'eff': 1.0, 'mode': 'nom'}, {'no
m'}),
 'ExportWater': ExportWater ExportWater function: ({'mode': 'nom'}, {'nom'})}
```

We can also view the run order to see how the model will be simulated.

In [7]:
```
fig, ax = rd.graph.exec_order(mdl)
```



As shown, because all of the methods were defined as generic behaviors, they are each run in the static propagation step. No order is shown in the static step because the static propagation step iterates between model functions until the values have converged. Nevertheless, one can view the initial static order using:

In [8]:
```
mdl.staticfxns
```

Out[8]:
```
OrderedSet(['ImportEE', 'ImportWater', 'ImportSignal', 'MoveWater', 'ExportWate
r'])
```

and the dynamic step order (if there was one):

In [9]:
```
mdl.dynamicfxns
```

Out[9]:
```
OrderedSet()
```

We can also instantiate this model with different model parameters. By default, this model gets `params={'cost':{'repair', 'water'}, 'delay':10, 'units':'hrs'}` but we can pass any set of costs that is a subset of `{'repair', 'water', 'water_exp', 'ee'}` and any int `delay` parameter.

In [10]:
```
mdl2 = Pump(params={'cost':{'repair', 'water', 'ee'}, 'delay':20, 'units':'hrs'})
```

2.) and 3.) Simulate and visualize the results!

Now, we will use the methods in `propagate` and the visualization modules in `rd` to simulate the model and visualize the results.

2a.) Simulate nominal

To simulate the model in the nominal scenario, use the `propagate.nominal` method.

In [11]:
```
endresults_nominal, resgraph_nominal, mdlhist_nominal=propagate.nominal(mdl, trac
```

What do the results look like? Explore results structures.

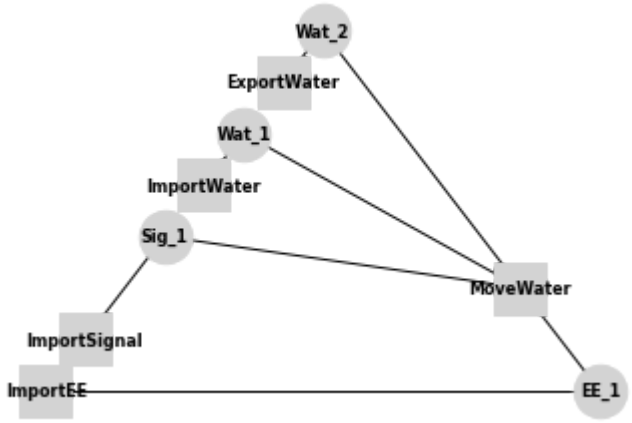
In [12]:
```
endresults_nominal
```

Out[12]:
```
{'faults': {},
 'classification': {'rate': 1.0, 'cost': 0.0, 'expected cost': 0.0}}
```

2b.) Visualize nominal model

First, we can show the model graph using `rd.graph.show` to see that it was set up correctly. We can do this both on the model graph itself `mdl.graph` and the results of the nominal run `resgraph_nominal` to verify both are fault-free.

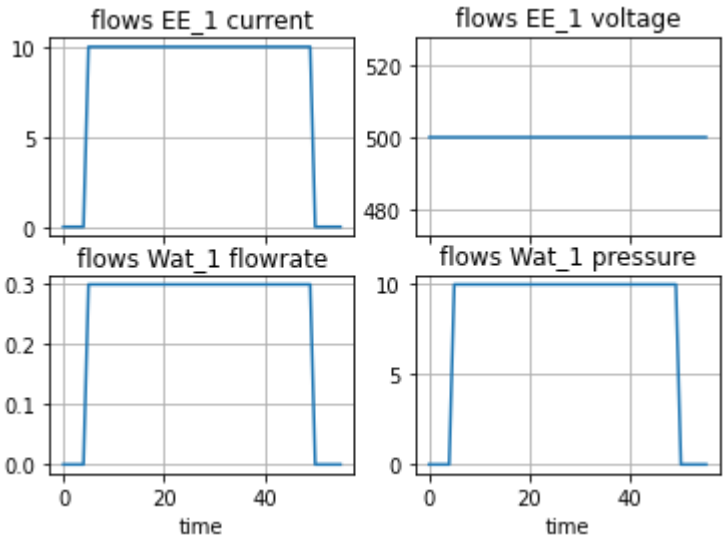
In [14]:
```
fig, ax = rd.graph.show(mdl)
```



We can also view the flow values of the model using "rd.plot.mdlhists". It may be helpful to only

view flows of interest.

In [15]:
```
fig = rd.plot.mdlhists(mdlhist_nominal, fxnflowvals={'Wat_1':['flowrate','pressur
```



Note: for quick access to the syntax and options for these methods, type the `?method` or `help(method)` in the terminal. For example `?rd.plot.mdlhists`

In [17]:
```
?rd.plot.mdlhists
```

2b.) Simulate a fault mode

To simulate the model in the nominal scenario, use the `propagate.one_fault` method. The set of possible faults is defined in the function definitions in `ex_pump.py`, and we can propagate a fault at any time in the operational interval (0-55 seconds).

In [18]:
```
endresults_fault, resgraph_fault, mdlhist_fault=propagate.one_fault(mdl, 'MoveWat
```

We can also view the results for from this. In this case `mdlhist` gives a history of results for both the nominal and faulty runs.

In [19]:
```
endresults_fault
```

Out[19]:
```
{'flows': {'EE_1': {'voltage': 0.0}},
 'faults': {'ImportEE': ['no_v'], 'MoveWater': ['short']},
 'classification': {'rate': 0.00055,
  'cost': 29000.000000000007,
  'expected cost': 1595000.0000000005}}
```

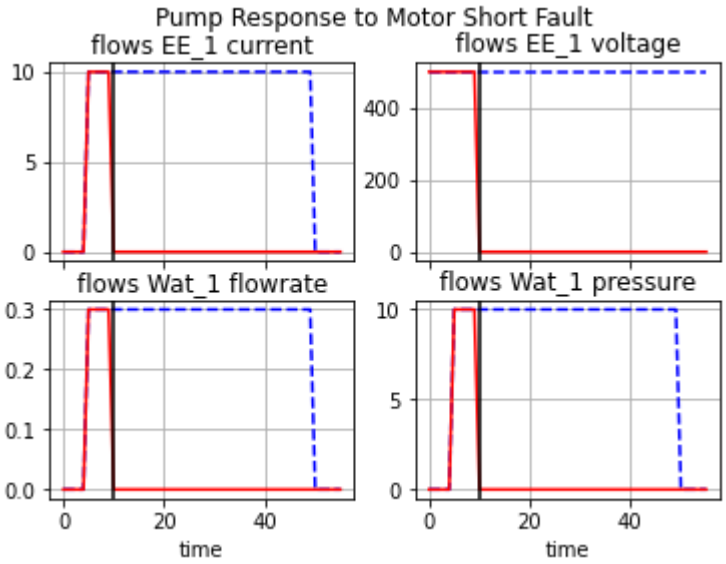
In [20]:
```
list(mdlhist_fault.keys())
```

Out[20]:
```
['nominal', 'faulty']
```

3b.) Visualize fault model states

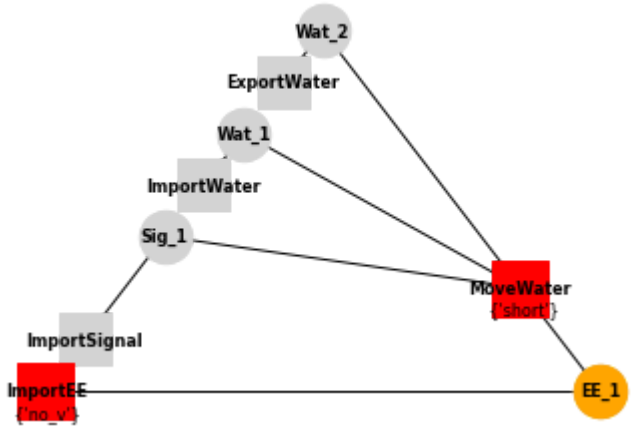
`rd.plot.mdlhistvals` also works for a mdlhists given from `propagate.one_fault`. We can view these results below. As shown, the function will give the nominal result in a blue dotted line and the faulty result in a red line.

In [21]:
```
fig, axs = rd.plot.mdlhists(mdlhist_fault, title='Pump Response to Motor Short Fa
```



We can also view this result graph using 'rd.graph.show'. In this case, it shows the state of the model at the final time-step of the model run. Thus, while the `EE_1` flow is shown in orange (because it is off-nominal), the Water flows are not, because they have the same state at the final time-step.

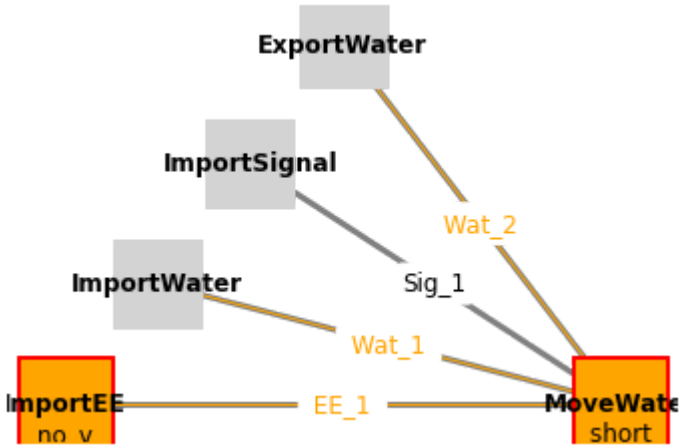
In [22]:
```
fig, ax = rd.graph.show(resgraph_fault)
```



If we want to view the graph at another time-step, we can use 'rd.process.hist' and 'rd.graph.result_from' to first process the model states into results which can be visualized and then map that onto a graph view at a given time.

In [23]:
```
reshist_fault, b, c = rd.process.hist(mdlhist_fault)
```

In [24]:
```
fig, ax = rd.graph.result_from(mdl, reshist_fault, 20, gtype='normal')
```



4a.) Simulate set of fault modes

To simulate the set of fault modes, we first choose a `SampleApproach` . For simplicity, we can choose default parameters at first.

In [25]:
```
app = SampleApproach(mdl)
```

In [26]:
```
endclasses_app, mdlhists_app = propagate.approach(mdl, app)
```

```
SCENARIOS COMPLETE: 100%|████████████████████████████████████████████████████
██████| 17/17 [00:00<00:00, 165.43it/s]
```

It can be helpful to view what these results look like--a dictionary of faults injected at particular times with their respective results dictionaries.

In [27]:
```
endclasses_app
```

Out[27]:

```
{'ImportEE no_v, t=27': {'rate': 0.0003600000000000001,
  'cost': 15174.999999999998,
  'expected cost': 546300.0},
 'ImportEE inf_v, t=27': {'rate': 9.000000000000002e-05,
  'cost': 20175.0,
  'expected cost': 181575.00000000003},
 'ImportWater no_wat, t=27': {'rate': 0.00015,
  'cost': 6174.999999999998,
  'expected cost': 92624.99999999996},
 'ImportSignal no_sig, t=27': {'rate': 1.2857142857142856e-05,
  'cost': 15174.999999999998,
  'expected cost': 19510.714285714283},
 'MoveWater mech_break, t=27': {'rate': 0.00023142857142857142,
  'cost': 10174.999999999998,
  'expected cost': 235478.5714285714},
 'MoveWater short, t=27': {'rate': 0.00012857142857142858,
  'cost': 25175.0,
  'expected cost': 323678.5714285714},
 'ExportWater block, t=27': {'rate': 0.00012857142857142858,
  'cost': 15152.5,
  'expected cost': 194817.85714285716},
 'ImportWater no_wat, t=2': {'rate': 1.6666666666666667e-05,
  'cost': 11125.000000000007,
  'expected cost': 18541.66666666668},
 'ImportSignal no_sig, t=2': {'rate': 2.1428571428571427e-06,
  'cost': 20125.000000000007,
  'expected cost': 4312.500000000001},
 'MoveWater mech_break, t=2': {'rate': 2.1428571428571427e-06,
  'cost': 15125.000000000007,
  'expected cost': 3241.07142857143},
 'MoveWater short, t=2': {'rate': 2.1428571428571428e-05,
  'cost': 30125.000000000007,
  'expected cost': 64553.57142857144},
 'ExportWater block, t=2': {'rate': 2.1428571428571428e-05,
  'cost': 20102.500000000007,
  'expected cost': 43076.78571428573},
 'ImportWater no_wat, t=52': {'rate': 1.6666666666666667e-05,
  'cost': 1000.0,
  'expected cost': 1666.6666666666667},
 'ImportSignal no_sig, t=52': {'rate': 1.4285714285714284e-06,
  'cost': 10000.0,
  'expected cost': 1428.5714285714284},
 'MoveWater mech_break, t=52': {'rate': 2.1428571428571427e-06,
  'cost': 5000.0,
  'expected cost': 1071.4285714285713},
 'MoveWater short, t=52': {'rate': 1.4285714285714285e-05,
  'cost': 10000.0,
  'expected cost': 14285.714285714286},
 'ExportWater block, t=52': {'rate': 1.4285714285714285e-05,
  'cost': 5000.0,
  'expected cost': 7142.857142857143},
 'nominal': {'rate': 1.0, 'cost': 0.0, 'expected cost': 0.0}}
```


It can be helpful to explore different approach parameters (e.g. focussing on single faults, different

numbers of sample points, etc)

In [28]:
```
app2 = SampleApproach(mdl, faults=[('MoveWater', 'short')])
```

In [29]:
```
app2.list_modes()
```

Out[29]:
```
[('MoveWater', 'short')]
```

There are a number of different ways to sample the scenarios in the approach:

In [30]:
```
app3 = SampleApproach(mdl, defaultsamp={'samp':'evenspacing', 'numpts':5})
```

In [31]:
```
app3.times
```

Out[31]:
```
[0, 1, 2, 3, 4, 12, 20, 27, 34, 42, 50, 51, 52, 53, 54]
```

4b.) Visualize set of fault modes

Using this fault approach, we can now make an FMEA-like analyses of the different fault modes. `rd.tabulate.summfmea` organizes endclasses into a table for each fault.

In [32]:
```
summary_fmea = rd.tabulate.summfmea(endclasses_app, app)
summary_fmea
```

Out[32]:

		rate	cost	expected cost
ImportEE	**no_v**	0.000360	15175.000000	546300.000000
	inf_v	0.000090	20175.000000	181575.000000
ImportWater	**no_wat**	0.000183	6100.000000	112833.333333
ImportSignal	**no_sig**	0.000016	15100.000000	25251.785714
MoveWater	**mech_break**	0.000236	10100.000000	239791.071429
	short	0.000164	21766.666667	402517.857143
ExportWater	**block**	0.000164	13418.333333	245037.500000

We can also use `rd.tabulatefullmea` with the processed results histories to get a better picture of which flows and functions degrade in each scenario

In [33]:
```
reshist_fault, b, summaries = rd.process.hists(mdlhists_app)
```

In [34]:
```
rd.tabulate.fullfmea(endclasses_app, summaries)
```

Out[34]:

	degraded functions	degraded flows	rate	cost	expected cost
ImportEE no_v, t=27	[ImportEE]	[EE_1, Wat_1, Wat_2]	0.00036	15175.0	546300.0
ImportEE inf_v, t=27	[ImportEE]	[EE_1, Wat_1, Wat_2]	0.00009	20175.0	181575.0
ImportWater no_wat, t=27	[ImportWater]	[EE_1, Wat_1, Wat_2]	0.00015	6175.0	92625.0
ImportSignal no_sig, t=27	[ImportSignal]	[EE_1, Sig_1, Wat_1, Wat_2]	0.000013	15175.0	19510.714286
MoveWater mech_break, t=27	[MoveWater]	[EE_1, Wat_1, Wat_2]	0.000231	10175.0	235478.571429
MoveWater short, t=27	[ImportEE, MoveWater]	[EE_1, Wat_1, Wat_2]	0.000129	25175.0	323678.571429
ExportWater block, t=27	[MoveWater, ExportWater]	[EE_1, Wat_1, Wat_2]	0.000129	15152.5	194817.857143
ImportWater no_wat, t=2	[ImportWater]	[EE_1, Wat_1, Wat_2]	0.000017	11125.0	18541.666667
ImportSignal no_sig, t=2	[ImportSignal]	[EE_1, Sig_1, Wat_1, Wat_2]	0.000002	20125.0	4312.5
MoveWater mech_break, t=2	[MoveWater]	[EE_1, Wat_1, Wat_2]	0.000002	15125.0	3241.071429
MoveWater short, t=2	[ImportEE, MoveWater]	[EE_1, Wat_1, Wat_2]	0.000021	30125.0	64553.571429
ExportWater block, t=2	[MoveWater, ExportWater]	[EE_1, Wat_1, Wat_2]	0.000021	20102.5	43076.785714
ImportWater no_wat, t=52	[ImportWater]	[Wat_1]	0.000017	1000.0	1666.666667
ImportSignal no_sig, t=52	[ImportSignal]	[]	0.000001	10000.0	1428.571429
MoveWater mech_break, t=52	[MoveWater]	[]	0.000002	5000.0	1071.428571
MoveWater short, t=52	[MoveWater]	[]	0.000014	10000.0	14285.714286
ExportWater block, t=52	[ExportWater]	[Wat_2]	0.000014	5000.0	7142.857143
nominal	NaN	NaN	1.0	0.0	0.0

5.) Saving Work

In detailed simulations, running a lot of computational simulations can take a considerable amount of time. As a result, it becomes impractical to run a new simulation every time one wishes to analyse its data. The `dill` package gives one a very simple method of saving and reloading a workspace one is working on:

- `dill.dump_session("filename.pkl")` to save the session, and
- `dill.load_session("filename.pk1")` to load the session.

Below, we demonstrate this by saving the session, clearing the workspace, and then loading the workspace again.

In [35]:
```
import dill
```

Viewing a workspace variable:

In [36]:
```
endclasses_app
```

Out[36]:

```
{'ImportEE no_v, t=27': {'rate': 0.0003600000000000001,
  'cost': 15174.999999999998,
  'expected cost': 546300.0},
 'ImportEE inf_v, t=27': {'rate': 9.000000000000002e-05,
  'cost': 20175.0,
  'expected cost': 181575.00000000003},
 'ImportWater no_wat, t=27': {'rate': 0.00015,
  'cost': 6174.999999999998,
  'expected cost': 92624.99999999996},
 'ImportSignal no_sig, t=27': {'rate': 1.2857142857142856e-05,
  'cost': 15174.999999999998,
  'expected cost': 19510.714285714283},
 'MoveWater mech_break, t=27': {'rate': 0.00023142857142857142,
  'cost': 10174.999999999998,
  'expected cost': 235478.5714285714},
 'MoveWater short, t=27': {'rate': 0.00012857142857142858,
  'cost': 25175.0,
  'expected cost': 323678.5714285714},
 'ExportWater block, t=27': {'rate': 0.00012857142857142858,
  'cost': 15152.5,
  'expected cost': 194817.85714285716},
 'ImportWater no_wat, t=2': {'rate': 1.6666666666666667e-05,
  'cost': 11125.000000000007,
  'expected cost': 18541.66666666668},
 'ImportSignal no_sig, t=2': {'rate': 2.1428571428571427e-06,
  'cost': 20125.000000000007,
  'expected cost': 4312.500000000001},
 'MoveWater mech_break, t=2': {'rate': 2.1428571428571427e-06,
  'cost': 15125.000000000007,
  'expected cost': 3241.07142857143},
 'MoveWater short, t=2': {'rate': 2.1428571428571428e-05,
  'cost': 30125.000000000007,
  'expected cost': 64553.57142857144},
 'ExportWater block, t=2': {'rate': 2.1428571428571428e-05,
  'cost': 20102.500000000007,
  'expected cost': 43076.78571428573},
 'ImportWater no_wat, t=52': {'rate': 1.6666666666666667e-05,
  'cost': 1000.0,
  'expected cost': 1666.6666666666667},
 'ImportSignal no_sig, t=52': {'rate': 1.4285714285714284e-06,
  'cost': 10000.0,
  'expected cost': 1428.5714285714284},
 'MoveWater mech_break, t=52': {'rate': 2.1428571428571427e-06,
  'cost': 5000.0,
  'expected cost': 1071.4285714285713},
 'MoveWater short, t=52': {'rate': 1.4285714285714285e-05,
  'cost': 10000.0,
  'expected cost': 14285.714285714286},
 'ExportWater block, t=52': {'rate': 1.4285714285714285e-05,
  'cost': 5000.0,
  'expected cost': 7142.857142857143},
 'nominal': {'rate': 1.0, 'cost': 0.0, 'expected cost': 0.0}}
```


Dumping the session:

In [37]:
```
dill.dump_session('Tutorial.pkl')
```

Reseting the workspace and viewing the variables:

In [38]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

In [39]:
```
endclasses_app
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\1/ipykernel_2352/2177989256.py in <module>
----> 1 endclasses_app

NameError: name 'endclasses_app' is not defined
```

Loading Session:

In [40]:
```
import dill
dill.load_session('Tutorial.pkl')
```

In [41]:
```
endclasses_app
```

Out[41]:

```
{'ImportEE no_v, t=27': {'rate': 0.0003600000000000001,
  'cost': 15174.999999999998,
  'expected cost': 546300.0},
 'ImportEE inf_v, t=27': {'rate': 9.000000000000002e-05,
  'cost': 20175.0,
  'expected cost': 181575.00000000003},
 'ImportWater no_wat, t=27': {'rate': 0.00015,
  'cost': 6174.999999999998,
  'expected cost': 92624.99999999996},
 'ImportSignal no_sig, t=27': {'rate': 1.2857142857142856e-05,
  'cost': 15174.999999999998,
  'expected cost': 19510.714285714283},
 'MoveWater mech_break, t=27': {'rate': 0.00023142857142857142,
  'cost': 10174.999999999998,
  'expected cost': 235478.5714285714},
 'MoveWater short, t=27': {'rate': 0.00012857142857142858,
  'cost': 25175.0,
  'expected cost': 323678.5714285714},
 'ExportWater block, t=27': {'rate': 0.00012857142857142858,
  'cost': 15152.5,
  'expected cost': 194817.85714285716},
 'ImportWater no_wat, t=2': {'rate': 1.6666666666666667e-05,
  'cost': 11125.000000000007,
  'expected cost': 18541.66666666668},
 'ImportSignal no_sig, t=2': {'rate': 2.1428571428571427e-06,
  'cost': 20125.000000000007,
  'expected cost': 4312.500000000001},
 'MoveWater mech_break, t=2': {'rate': 2.1428571428571427e-06,
  'cost': 15125.000000000007,
  'expected cost': 3241.07142857143},
 'MoveWater short, t=2': {'rate': 2.1428571428571428e-05,
  'cost': 30125.000000000007,
  'expected cost': 64553.57142857144},
 'ExportWater block, t=2': {'rate': 2.1428571428571428e-05,
  'cost': 20102.500000000007,
  'expected cost': 43076.78571428573},
 'ImportWater no_wat, t=52': {'rate': 1.6666666666666667e-05,
  'cost': 1000.0,
  'expected cost': 1666.6666666666667},
 'ImportSignal no_sig, t=52': {'rate': 1.4285714285714284e-06,
  'cost': 10000.0,
  'expected cost': 1428.5714285714284},
 'MoveWater mech_break, t=52': {'rate': 2.1428571428571427e-06,
  'cost': 5000.0,
  'expected cost': 1071.4285714285713},
 'MoveWater short, t=52': {'rate': 1.4285714285714285e-05,
  'cost': 10000.0,
  'expected cost': 14285.714285714286},
 'ExportWater block, t=52': {'rate': 1.4285714285714285e-05,
  'cost': 5000.0,
  'expected cost': 7142.857142857143},
 'nominal': {'rate': 1.0, 'cost': 0.0, 'expected cost': 0.0}}
```


`dill` can also save/load individual variables, which may be more helpful when a notebook

contains multiple large simulations. For this, use:

- `dill.dump(variablename, 'filename.pkl')` and
- `variablename = dill.load('filename.pkl)` , respectively.

See `dill` documentation: https://dill.readthedocs.io/en/latest/dill.html (https://dill.readthedocs.io/en/latest/dill.html)

Sometimes `dill` may not be able to dump the session, if there are variables in the session which are unpickleable. Frequent offenders are variables which hold iterators (e.g. self.a={1:2}.keys(), so it can be helpful to look out for those. You can debug dill by running `dill.detect.trace(True)` or using other methods in the `dill.detect` module.

In [ ]: