energysim

Release 2.0

Digvijay Gusain

Jul 24, 2020

CONTENTS

Compatible with Python 3.6 and above.

CHAPTER

ONE

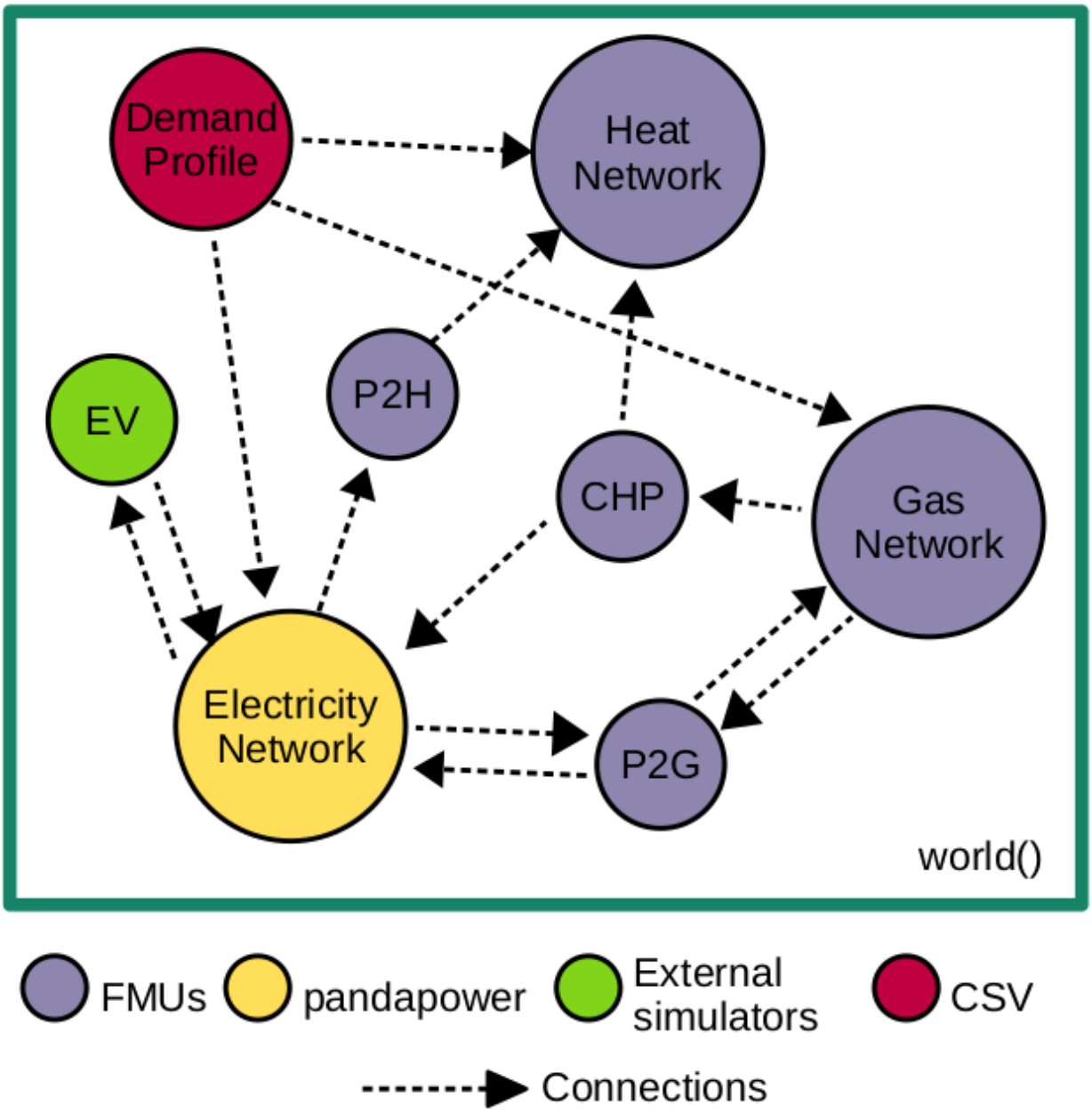
WHAT IS ENERGYSIM?

`energysim` is a python based cosimulation tool designed to simplify multi-energy cosimulations. The tool was initially called `FMUWorld`, since it focussed exclusively on combining models developed and packaged as Functional Mockup Units (FMUs). However, it has since been majorly updated to become a more generalisable cosimulation tool to include a more variety of energy system simulators.

The idea behind development of `energysim` is to simplify cosimulation to focus on the high-level applications, such as energy system planning, evaluation of control strategies, etc., rather than low-level cosimulation tasks such as message exchange, time progression coordination, etc.

Currently, `energysim` allows users to combine:

1. Dynamic models packaged as *Functional Mockup Units*.
2. Pandapower networks packaged as *pickle files*.
3. PyPSA models (still under testing) as *Excel files*.
4. User-defined external simulators interfaced with *.py functions*.
5. CSV data files


Demand Profile
Heat Network
EV
P2H
CHP
Gas Network
Electricity Network
P2G
world()
FMUs
pandapower
External simulators
CSV
Connections

CHAPTER

TWO

INSTALLATION

`energysim` can be installed with `pip` using:

```
pip install energysim
```

2.1 Dependencies

`energysim` requires the following packages to work:

1. FMPy
2. Pandapower
3. PyPSA
4. NumPy
5. Pandas
6. Matplotlib
7. NetworkX
8. tqdm

CHAPTER

THREE

USAGE

`energysim` cosimulation is designed for an easy-plug-and-play approach. The main component is the `world()` object. This is the "playground" where all simulators, and connections are added and the options for simulation are specified. `world()` can be imported by implementing:

```
from energysim import world
```

3.1 Initialization

Once `world` is imported, it can be initialized with basic simulation parameters using:

```
my_world = world(start_time=0, stop_time=1000, logging=True, t_macro=60)
```

`world` accepts the following parameters :

- `start_time` : simulation start time (0 by default).
- `stop_time` : simulation end time (1000 by default).
- `logging` : Flag to toggle update on simulation progress (True by default).
- `t_macro` : Time steps at which information between simulators needs to be exchanged. (60 by default).

3.2 Adding Simulators

After initializing the world cosimulation object, simulators can be added to the world using the `add_simulator()` method:

```
my_world.add_simulator(sim_type='fmu', sim_name='FMU1',
sim_loc=/path/to/sim, inputs=['v1', 'v2'], outputs=['var1','var2'], step_size=1)
```

where:

- `sim_type` : 'fmu', 'powerflow', 'csv', 'external'
- `sim_name` : Unique simulator name.
- `sim_loc` : A raw string address of simulator location.
- `outputs` : Variables that need to be recorded from the simulator during simulation.
- `inputs` : Input variables to the simulator.
- `step_size` : Internal step size for simulator (1e-3 by default).

Please see documentation on `add_simulator` to properly add simulators to `energysim`. The values to simulator input are kept constant for the duration between two macro time steps.

3.3 Connections between simulators

Once all the required simulators are added, the connections between them can be specified with a dictionary as follows

```
connections = {'sim1.output_variable1' : 'sim2.input_variable1',
   'sim3.output_variable2' : 'sim4.input_variable2',
   'sim1.output_variable3' : 'sim2.input_variable3',}
```

This dictionary can be passed onto the world object:

```
my_world.add_connections(connections)
```

3.4 Initializing simulator variables

Initialization is important to start-up simulator in a cosimulation. If the simulators are not internally initialized, or of users want to use different initial conditions for the simulators, it can easily be done in `energysim`. To provide initial values to the simulators, an `init` dictionary can be specified and given to the `world` object

```
initializations = {'sim_name1' : (['sim_variables'], [values]),
                   'sim_name2' : (['sim_variables'], [values])}
options = {'init' : initializations}
my_world.options(options)
```

3.5 Executing simulation

Finally, the `simulate()` function can be called to simulate the world. This returns a dictionary with simulator name as keys and the results of the simulator as pandas dataframe. `pbar` can be used to toggle the progress bar for the simulation:

```
results = my_world.simulate(pbar=True)
```

3.5.1 Adding simulators

Adding simulators in energysim is done by calling the method `add_simulator()`. As mentioned, four main types of simulators can be added: FMU, powerflow networks, CSV, and external simulators.

The add_simulator requires **six** main arguments for all simulators. Optional arguments for FMUs, powerflows, and external simulators can be added if needed. These are:

- `sim_type` : Specifies simulator type. Can be either of 'fmu', 'powerflow', 'csv', 'external'
- `sim_name` : Each simulator must have a unique name. This is later required when specifying connections.
- `sim_loc` : A raw string address of simulator location.
- `outputs` : Variables that need to be recorded from the simulator during simulation.
- `inputs` : Input variables to the simulator that are to be used while defining the connections.

- `step_size` : Internal step size for simulator (1e-3 by default). Also known as micro-time steps. This is the time integration step required for solvers for the simulators. For example, some FMUs require integration time steps of 1e-3 secs, while for powerflow networks, the time step can be 15 mins (900s).

The `add_simulator()` works as follows:

```
my_world.add_simulator(sim_type = sim_type, sim_name = sim_name,
                       sim_loc = sim_loc, outputs = outputs, inputs = inputs,
                           step_size = step_size)
```

Additional arguments

Apart from the six required arguments, users can also specify additional arguments for each simulator. These are:

1. For FMUs: FMPy generally validates the FMUs before initialization process. Users can skip this validation by specifyng the argument *validate* = *False*. Generally, it has been observed, the FMUs packaged with OpenModelica fail validation. However, they can be simulated by setting the `validate = False`.
2. For powerflow: For powerflow simulators, users can specify whether to execute a AC powerflow or an optimal power flow, or a DCpowerflow. This can be done by providing the specifying the value of argument `pf`. It can be set to "pf", "dcpf", "opf", or "dcopf". Please note, this is only available for pandapower networks currently.
3. For csv: For csv files, the users can specify the delimiter by providing the argument `delimiter=','` or whatever the delimiter is.

Variable naming convention

Since variable extraction is an important part of cosimulation, it is important to become aware of the variable naming convention used for the simulators in `energysim`.

FMUs

For FMUs, the variable naming convention is similar to how it is generally accessed within Modelica based environments. For example a variable_k nested within the FMU model can be accessed using:

```
sim_name.Component_1.SubComponent_i.Variable_k
```

This is to be followed when specifying connections, initializations, or signal modifications.

Powerflow networks

Currently, `energysim` can only record bus voltages magnitude (V) and bus voltage angles (VA), along with active (P) and reactive power (Q) values for loads, static generators, external grid, generators. For inputs, it can set the active (P) and reactive power (Q) setpoints for loads, static generators, generators. All the elements within the network must have a valid name. This has to be ensured before importing the pandapower network within `energysim` environment.

Consider a network 'grid' with 3 buses named 'Bus1', 'Bus2', 'Bus3'. It has three loads named 'Load1', 'Load2', and 'Load3'. Similarly, a generator 'Gen1' is connected to one of the buses.

The following quantities can be specified to receive inputs in the connections dictionary:

- Gen1.P
- Gen1.Q
- Load1.P, Load2.P, Load3.P

- Load1.Q, Load2.Q, Load3.Q

The following quantities can be speified as outputs in `add_simulator` to be recorded:

- Bus1.V, Bus2.V, Bus3.V
- Bus1.Va, Bus2.Va, Bus3.Va
- Gen1.P
- Gen1.Q
- Load1.P, Load2.P, Load3.P
- Load1.Q, Load2.Q, Load3.Q

It must be clarified that energysim can only retreive or set variables in simulators when the simulators name are an exact match. Please make sure that component names and variable names such as P, Q are exactly how they are specified here.

CSV files

CSV simulators are used to attach csv data files to `world`. The csv file must have clearly specified columns. One of the columns must be 'time'.

The output variables for CSV simulators are the column names. Consider a csv file given by:

time,power
0,18
1,18
2,20
3,50
4,25
5,15

The "power" variable can be accessed using `sim_name.power`. `energysim` can then automatically read the power variable from csv files corresponding to simulation time. If the current simulation time is between two time values, `energysim` will read the value at time given by `index = int(np.argwhere(time_array>current_time)[0] - 1)` where `time_array` is the list of time values in the csv.

External simulators

Variables in external simulators can be accessed similar to other simulators:

```
sim_name.var1
```

Users must make sure that the variable names within the simulator and that defined in `energysim` connections dict are the same.

3.5.2 Working with external simulators

One of the USPs of `energysim` is that it can be coupled to external simulators fairly easily. However, it is expected that the users are familiar with intermediate level of python programming. In this section, we show how users can interface their own simulators with `energysim` very easily.

In the main code, users can add their simulators to `my_world` by using:

```
my_world.add_simulator(sim_type='external', sim_name = 'my_external_simulator', sim_
↪loc = sim_loc, outputs=['var1', 'var2'], step_size=1)
```

Here the `sim_name` is the file name of the interfaced simulator *my_external_simulator.py*. This file has the following template:

```
#make necessary imports


class external_simulator():

    def __init__(self, sim_name, sim_loc, inputs = [], outputs = [], step_size=1):
        self.sim_name = sim_name
        self.sim_loc = sim_loc
        self.inputs = inputs
        self.outputs = outputs
        self.step_size = step_size

    def init(self):
        #specify simulator initialization command

        #remove pass after initialization has been set
        pass

    def set_value(self, variable, value):
        #this should set the simulator paramaters as values. Return cmd not reqd

        #remove the pass after specifying set_value
        pass

    def get_value(self, variable, time):
        #this should return a list of values from simulator as a list corresponding␣
↪to parameters

        **Return reqd**

        #remove the pass after specifying get_value.
        pass

    def step(self, time):
        #use the time variable (if needed) to step the simulator to t=time

        #return is not required. remove the pass command afterwards.
        pass
```

The four functions inside `class external_simulator()` are all that `energysim` requires to interface with the simulator. Users are free to make imports, and create other functions which can be called within this file. Let us go through each function and their definitions.

init() method

Note that this is different from the `__init__()` class. This method is needed to initialize the simulator. You can use it to, for example, establish connection to another software, or package. Basically, start-up the simulator.

step(time) method

The `step` method is used by `energysim` coordinator to perform time stepping for each simulator. The coordinator steps the simulator by deltaT = `step_size` defined while adding the simulator to `world`. The step method is useful when the model consists consists of time-dependent equations and exibits dynamics behavior.

get_value(variable, time) method

The `get_value` method is used by the simulator to query `variable` value from the simulator. The coordinator queries the simulator by asking the value of `variable` at time=``time``. The `variable` is enclosed in a python list. The user must define in this method, how to obtain the value of that `variable` from its simulator.

set_value(variable, value) method

The `set_value` method is used by the simulator to set the variable to a particular value at the time instance when message are exchanged between simulators. In this method, users must specify how to set the `variable` to the specified `value`. Both the `variable` and `value` are enclosed within a list.

3.5.3 energysim features

Although key functions were highlighted in the main page, `energysim` comes with additional inbuilt methods which allow users to take more control of the cosimulation.

Adding signals

The `add_signal()` method of the `world` object provides the ability to add user-defined time (in)variant signals to the cosimulation objects. This is especially useful if some inputs of the cosimulation simulators need a constant signal, or a time varying signal (such as sin). In `energysim`, this can be added by:

```
def my_signal(time):
    return [1]

my_world.add_signal(sim_name='constant_signal', signal = my_signal, step_size=1)
```

Users need to make sure that the return value from the my_signal part is within square brackets ([ ]), i.e. a **list**. The value returned must also be a single value. If multiple values are returned, the signal function will not be added. The signal function can also be more complex. For example, instead of `return [1]`, the `my_signal` function can also `return np.sin(2*np.pi*time)`.

In the connections dictionary, this signal can then be connected to other simulators by:

```
connections = {'constant_signal.y' : 'sim1.input_variable1'}
my_world.add_connections(connections)
```

The default step size is 1s for signals. However, it can be changed by specifying `step_size` argument in the `add_signal` method.

Modify signals before exchange

Many times, it is required in the simulation whereby output of a particular simulator needs to be "modified" before exchanging the value with a another simulator. This is fairly common in energy system integration simulations. For example, consider two simulators a CHP system and an electric network (EN). The power output from CHP simulator (an FMU) is obtained in Watts units. This power output of the CHP needs to be provided to the pandapower network. However, the EN accepts values only in MW. One way to address the problem is to change the output values of the CHP in the model itself and recompile the FMU. It may not always be possible to do so (FMU may be encrypted!). Therefore, `energysim` provides an inbuilt method to address such problems by supplying the `modify_dict` to `world` options. Two types of modifications can be applied: 1) multiply with a constant, and 2) multiply with a constant and add a constant. This is shown as follows

```
modifications = {'sim1.var1':[x], #multiplies var1 of sim1 by x before variables are␣
↪exchanged,
                 'sim2.var1':[x1, x2] #multiplies by x1 and adds x2
                }

options = {'init' : initializations,
           'modify_signal': modifications}

my_world.options(options)
```

In the example highlighted above, the modification can be set as:

```
#multiply electric power of chp by 1/1e6 to convert W -> MW before it is given to EN,
modifications = {'chp.e_power':[1/1e6]}
```

Enabling sensitivity analysis

An important part of energy system analysis is the parameter sensitivity. In `energysim`, this can be done by updating the `init` option to update parameters of the cosimulation.:

```
#configure energysim with simulators

for v1, v2 in [(0.1,0.2), (1,2), (10,20)]:
    sens = {'sim_name1' : (['sim_variables'], [values]),
            'sim_name2' : (['sim_variables'], [values])}
            options = {'init' : sens}
            my_world.options(options)
    res = my_world.simulate(pbar=False)
    #extract relevant results and store them
```

A more sophisticated functionality is planned to create an integrated sensitivity analysis with energysim.

Optimal Power Flow

By default, the powerflow network added in `energysim` are solved for ac powerflow. However, users can specify in the `add_simulator` arguments to solve for opf. This is shown below:

```
my_world.add_simulator(sim_type = 'powerflow', sim_name = 'grid',
        sim_loc = grid_loc, inputs = ['wind1.P'], outputs=['Bus 1.V', 'Bus 12.V',
↪'wind1.P'],
        step_size=3, pf = 'opf')
```

This feature is currently only available in pandapower networks.

Validation of FMUs

Internally, FMPy checks the validity of FMUs. To speedup, this flag can be set as `False` while adding simulators.

System Topology Plot

`energysim` uses `NetworkX` to generate topology of the cosimulation based on the connections dictionary. This can be visualised by:

```
my_world.plot(plot_edge_labels=False, node_size=300, node_color='r')
```

3.5.4 FAQ

OPFNotConvergedError

OPF and DCOPF functionalities are subject to pandapower optimization. Therefore, you must make sure that OPF converence is met within the pandapower network before integrating it with `energysim.world`.

FMU Initialization Error

If you get an initialization failed error for FMU, please check if it works independently. You can use the following code structure to check:

```
from fmpy import *
fmu_loc = /path/to/fmu
res=simulate_fmu(fmu_loc)
print(res)
```

If this works, but you still get initialization failed error with FMUs, you can try following remedies:

1. Clear cache, temp folders.
2. Restart application
3. Run it with admin privileges

3.5.5 Indices and tables

- genindex
- modindex
- search