oemof.network

Release 0.4.0

Stephan Günther

Apr 26, 2022

CONTENTS

CHAPTER

ONE

OVERVIEW

docs	
tests	
package	

The network/graph submodules of oemof.

- Free software: MIT license

1.1 Installation

```
pip install oemof-network
```

You can also install the in-development version with:

```
pip install https://github.com/oemof/oemof-network/archive/dev.zip
```

1.2 Documentation

https://oemof-network.readthedocs.io/

1.3 Development

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER

TWO

INSTALLATION

At the command line:

```
pip install oemof.network
```

CHAPTER

THREE

USAGE

3.1 oemof.network

The *oemof.network* library is part of the oemof installation. By now it can be used to define energy systems as a network with components and buses. Every component should be connected to one or more buses. After definition, a component has to explicitely be added to its energy system. Allowed components are sources, sinks and transformer.

The code of the example above:

```
from oemof.network import *
from oemof.energy_system import *

# create the energy system
es = EnergySystem()

# create bus 1
bus_1 = Bus(label="bus_1")

# create bus 2
bus_2 = Bus(label="bus_2")

# add bus 1 and bus 2 to energy system
es.add(bus_1, bus_2)

# create and add sink 1 to energy system
es.add(Sink(label='sink_1', inputs={bus_1: []}))

# create and add sink 2 to energy system
es.add(Sink(label='sink_2', inputs={bus_2: []}))

# create and add source to energy system
es.add(Source(label='source', outputs={bus_1: []}))

# create and add transformer to energy system
es.add(Transformer(label='transformer', inputs={bus_1: []}, outputs={bus_2: []}))
```

The network class is aimed to be very generic and might have some network analyse tools in the future. By now the network library is mainly used as the base for the solph library.

To use oemof.network in a project:

```
import oemof.network
```

3.1.1 oemof.network

The modeling of energy supply systems and its variety of components has a clearly structured approach within the oemof framework. Thus, energy supply systems with different levels of complexity can be based on equal basic module blocks. Those form an universal basic structure.

A *node* is either a *bus* or a *component*. A bus is always connected with one or several components. Likewise components are always connected with one or several buses. Based on their characteristics components are divided into several sub types.

Transformers have any number of inputs and outputs, e.g. a CHP takes from a bus of type 'gas' and feeds into a bus of type 'electricity' and a bus of type 'heat'. With additional information like parameters and transfer functions input and output can be specified. Using the example of a gas turbine, the resource consumption (input) is related to the provided end energy (output) by means of an conversion factor. Components of type *transformer* can also be used to model transmission lines.

A *sink* has only an input but no output. With *sink* consumers like households can be modeled. But also for modelling excess energy you would use a *sink*.

A *source* has exactly one output but no input. Thus for example, wind energy and photovoltaic plants can be modeled.

Components and buses can be combined to an energy system. Components and buses are nodes, connected among each other through edges which are the inputs and outputs of the components. Such a model can be interpreted mathematically as bipartite graph as buses are solely connected to components and vice versa. Thereby the in- and outputs of the components are the directed edges of the graph. The components and buses themselves are the nodes of the graph.

oemof.network is part of oemofs core and contains the base classes that are used in oemof-solph. You do not need to define your energy system on the network level as all components can be found in oemof-solph, too. You may want to inherit from oemof.network components if you want to create new components.

Graph

In the graph module you will find a function to create a networkx graph from an energy system or solph model. The networkx package provides many features to analyse, draw and export graphs. See the networkx documentation for more details. See the API-doc of `graph` for all details and an example. The graph module can be used with energy systems of solph as well.

CHAPTER

FOUR

API REFERENCE

4.1 oemof.network

4.1.1 oemof.network.energy_system

Basic EnergySystem class

This file is part of project oemof (github.com/oemof/oemof). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location oemof/oemof/energy_system.py

SPDX-FileCopyrightText: Stephan Günther <> SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de> SPDX-FileCopyrightText: Simon Hilpert <> SPDX-FileCopyrightText: Cord Kaldemeyer <>

SPDX-License-Identifier: MIT

`class` oemof.network.energy_system.**EnergySystem**(***kwargs*)

Bases: `object`

Defining an energy supply system to use oemof's solver libraries.

Note: The list of regions is not necessary to use the energy system with solph.

Parameters

- **entities** (list of `Entity`, optional) – A list containing the already existing `Entities` that should be part of the energy system. Stored in the `entities` attribute. Defaults to *[]* if not supplied.
- **timeindex** (*pandas.datetimeindex*) – Defines the time range and, if equidistant, the timeindex for the energy system
- **timeincrement** (*numeric (sequence)*) – Define the timeincrement for the energy system
- **groupings** (*list*) – The elements of this list are used to construct `Groupings` or they are used directly if they are instances of `Grouping`. These groupings are then used to aggregate the entities added to this energy system into *groups*. By default, there'll always be one group for each `uid` containing exactly the entity with the given `uid`. See the *examples* for more information.

Variables

- **entities** (list of `Entity`) – A list containing the `Entities` that comprise the energy system. If this *EnergySystem* is set as the `registry` attribute, which is done automatically on *EnergySystem* construction, newly created `Entities` are automatically added to this list on construction.

- **groups** (*dict*) –
- **results** (*dictionary*) – A dictionary holding the results produced by the energy system. Is *None* while no results are produced. Currently only set after a call to `optimize()` after which it holds the return value of `om.results()`. See the documentation of that method for a detailed description of the structure of the results dictionary.
- **timeindex** (*pandas.index, optional*) – Define the time range and increment for the energy system. This is an optional attribute but might be import for other functions/methods that use the EnergySystem class as an input parameter.

Examples

Regardles of additional groupings, `entities` will always be grouped by their `uid`:

```
>>> from oemof.network.network import Bus, Sink
>>> es = EnergySystem()
>>> bus = Bus(label='electricity')
>>> es.add(bus)
>>> bus is es.groups['electricity']
True
>>> es.dump()
'Attributes dumped to:...
>>> es = EnergySystem()
>>> es.restore()
'Attributes restored from:...
>>> bus is es.groups['electricity']
False
>>> es.groups['electricity']
"<oemof.network.network.Bus: 'electricity'>"
```

For simple user defined groupings, you can just supply a function that computes a key from an `entity` and the resulting groups will be sets of `entities` stored under the returned keys, like in this example, where `entities` are grouped by their *type*:

```
>>> es = EnergySystem(groupings=[type])
>>> buses = set(Bus(label="Bus {}".format(i)) for i in range(9))
>>> es.add(*buses)
>>> components = set(Sink(label="Component {}".format(i))
...                  for i in range(9))
>>> es.add(*components)
>>> buses == es.groups[Bus]
True
>>> components == es.groups[Sink]
True
```

add(**nodes*)

Add nodes to this energy system.

dump(*dpath=None*, *filename=None*)

Dump an EnergySystem instance.

flows()

property groups

property nodes

restore(*dpath=None*, *filename=None*)

Restore an EnergySystem instance.

signals = {<function EnergySystem.add>: <blinker.base.NamedSignal object at 0x7f12a74532d0; <function EnergySystem.add>>}

A dictionary of blinker signals emitted by energy systems.

Currently only one signal is supported. This signal is emitted whenever a *Node <oemof.network.Node>* is *add`ed to an energy system. The signal's `sender* is set to the *node <oemof.network.Node>* that got added to the energy system so that *nodes <oemof.network.Node>* have an easy way to only receive signals for when they themselves get added to an energy system.

4.1.2 oemof.network.graph

Modules for creating and analysing energy system graphs.

This file is part of project oemof (github.com/oemof/oemof). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location oemof/oemof/graph.py

SPDX-FileCopyrightText: Simon Hilpert <> SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de>

SPDX-License-Identifier: MIT

oemof.network.graph.**create_nx_graph**(*energy_system=None*, *remove_nodes=None*, *filename=None*, *remove_nodes_with_substrings=None*, *remove_edges=None*)

Create a *networkx.DiGraph* for the passed energy system and plot it. See http://networkx.readthedocs.io/en/latest/ for more information.

Parameters

- **energy_system** (*oemof.solph.network.EnergySystem*)
- **filename** (*str*) – Absolute filename (with path) to write your graph in the graphml format. If no filename is given no file will be written.
- **remove_nodes** (*list of strings*) – Nodes to be removed e.g. ['node1', node2')]
- **remove_nodes_with_substrings** (*list of strings*) – Nodes that contain substrings to be removed e.g. ['elec', 'heat')]
- **remove_edges** (*list of string tuples*) – Edges to be removed e.g. [('resource_gas', 'gas_balance')]

Examples

```
>>> import os
>>> import pandas as pd
>>> from oemof.network.network import Bus, Sink, Transformer
>>> from oemof.network.energy_system import EnergySystem
>>> import oemof.network.graph as grph
>>> datetimeindex = pd.date_range('1/1/2017', periods=3, freq='H')
>>> es = EnergySystem(timeindex=datetimeindex)
>>> b_gas = Bus(label='b_gas', balanced=False)
```

(continues on next page)

(continued from previous page)

```
>>> bel1 = Bus(label='bel1')
>>> bel2 = Bus(label='bel2')
>>> demand_el = Sink(label='demand_el', inputs = [bel1])
>>> pp_gas = Transformer(label=('pp', 'gas'),
...                      inputs=[b_gas],
...                      outputs=[bel1],
...                      conversion_factors={bel1: 0.5})
>>> line_to2 = Transformer(label='line_to2', inputs=[bel1], outputs=[bel2])
>>> line_from2 = Transformer(label='line_from2',
...                          inputs=[bel2], outputs=[bel1])
>>> es.add(b_gas, bel1, demand_el, pp_gas, bel2, line_to2, line_from2)
>>> my_graph = grph.create_nx_graph(es)
>>> # export graph as .graphml for programs like Yed where it can be
>>> # sorted and customized. this is especially helpful for large graphs
>>> # grph.create_nx_graph(es, filename="my_graph.graphml")
>>> [my_graph.has_node(n)
...  for n in ['b_gas', 'bel1', "('pp', 'gas')", 'demand_el', 'tester']]
[True, True, True, True, False]
>>> list(nx.attracting_components(my_graph))
[{'demand_el'}]
>>> sorted(list(nx.strongly_connected_components(my_graph))[1])
['bel1', 'bel2', 'line_from2', 'line_to2']
>>> new_graph = grph.create_nx_graph(energy_system=es,
...                                  remove_nodes_with_substrings=['b_'],
...                                  remove_nodes=["('pp', 'gas')"],
...                                  remove_edges=[('bel2', 'line_from2')],
...                                  filename='test_graph')
>>> [new_graph.has_node(n)
...  for n in ['b_gas', 'bel1', "('pp', 'gas')", 'demand_el', 'tester']]
[False, True, False, True, False]
>>> my_graph.has_edge("('pp', 'gas')", 'bel1')
True
>>> new_graph.has_edge('bel2', 'line_from2')
False
>>> os.remove('test_graph.graphml')
```

Notes

Needs graphviz and networkx (>= v.1.11) to work properly. Tested on Ubuntu 16.04 x64 and solydxk (debian 9).

4.1.3 oemof.network.groupings

All you need to create groups of stuff in your energy system.

This file is part of project oemof (github.com/oemof/oemof). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location oemof/oemof/groupings.py

SPDX-FileCopyrightText: Stephan Günther <> SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de>

SPDX-License-Identifier: MIT

oemof.network.groupings.**DEFAULT = <oemof.network.groupings.Grouping object>**

The default *Grouping*.

This one is always present in an `energy system`. It stores every `entity` under its `uid` and raises an error if another `entity` with the same `uid` get's added to the `energy system`.

class oemof.network.groupings.**Flows**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)

Bases: *oemof.network.groupings.Nodes*

Specialises *Grouping* to group the flows connected to `nodes` into `sets`. Note that this specifically means that the `key`, and *value* functions act on a set of flows.

value(*flows*)

Returns a `set` containing only `flows`, so groups are `sets` of flows.

class oemof.network.groupings.**FlowsWithNodes**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)

Bases: *oemof.network.groupings.Nodes*

Specialises *Grouping* to act on the flows connected to `nodes` and create `sets` of `(source, target, flow)` tuples. Note that this specifically means that the `key`, and *value* functions act on sets like these.

value(*tuples*)

Returns a `set` containing only `tuples`, so groups are `sets` of `tuples`.

class oemof.network.groupings.**Grouping**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)

Bases: `object`

Used to aggregate `entities` in an `energy system` into `groups`.

The way *Groupings* work is that each *Grouping* `g` of an energy system is called whenever an `entity` is added to the energy system (and for each `entity` already present, if the energy system is created with existing enties). The call `g(e, groups)`, where `e` is an `entity` and `groups` is a dictionary mapping group keys to groups, then uses the three functions *key*, *value* and *merge* in the following way:

- *key(e)* is called to obtain a key `k` under which the group should be stored,
- *value(e)* is called to obtain a value `v` (the actual group) to store under `k`,
- if you supplied a *filter()* argument, `v` is `filtered` using that function,
- otherwise, if there is not yet anything stored under `groups[k]`, `groups[k]` is set to `v`. Otherwise *merge* is used to figure out how to merge `v` into the old value of `groups[k]`, i.e. `groups[k]` is set to *merge(v, groups[k])*.

Instead of trying to use this class directly, have a look at its subclasses, like *Nodes*, which should cater for most use cases.

Notes

When overriding methods using any of the constructor parameters, you don't have access to `self` in the corresponding function. If you need access to `self`, subclass *Grouping* and override the methods in the subclass.

A *Grouping* may be called more than once on the same object `e`, so one should make sure that user defined *Grouping* `g` is idempotent, i.e. `g(e, g(e, d)) == g(e, d)`.

Parameters

- **key** (*callable or hashable*) – Specifies (if not callable) or extracts (if callable) a *key* for each `entity` of the `energy system`.

- **constant_key** (*hashable (optional)*) – Specifies a constant *key*. Keys specified using this parameter are not called but taken as is.
- **value** (*callable, optional*) – Overrides the default behaviour of *value*.
- **filter** (*callable, optional*) – If supplied, whatever is returned by *value()* is `filtered` through this. Mostly useful in conjunction with static (i.e. non-callable) *keys*. See *filter()* for more details.
- **merge** (*callable, optional*) – Overrides the default behaviour of *merge*.

filter(*group*)

`Filter` the group returned by *value()* before storing it.

Should return a boolean value. If the `group` returned by *value()* is `iterable`, this function is used (via Python's `builtin filter`) to select the values which should be retained in `group`. If `group` is not `iterable`, it is simply called on `group` itself and the return value decides whether `group` is stored (`True`) or not (`False`).

key(*node*)

Obtain a key under which to store the group.

You have to supply this method yourself using the *key* parameter when creating *Grouping* instances.

Called for every `node` of the energy system. Expected to return the key (i.e. a valid `hashable`) under which the group *value(node)* will be stored. If it should be added to more than one group, return a `list` (or any other non-`hashable`, `iterable`) containing the group keys.

Return `None` if you don't want to store `e` in a group.

merge(*new*, *old*)

Merge a known `old` group with a `new` one.

This method is called if there is already a value stored under `group[key(e)]`. In that case, *merge(value(e), group[key(e)])* is called and should return the new group to store under *key(e)*.

The default behaviour is to raise an error if `new` and `old` are not identical.

value(*e*)

Generate the group obtained from `e`.

This methd returns the actual group obtained from `e`. Like *key*, it is called for every `e` in the energy system. If there is no group stored under *key(e)*, `groups[key(e)]` is set to *value(e)*. Otherwise *merge(value(e), groups[key(e)])* is called.

The default returns the `entity` itself.

class oemof.network.groupings.**Nodes**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)

Bases: *oemof.network.groupings.Grouping*

Specialises *Grouping* to group `nodes` into `sets`.

merge(*new*, *old*)

`Updates` `old` to be the union of `old` and `new`.

value(*e*)

Returns a `set` containing only `e`, so groups are `sets` of `node`.

4.1.4 oemof.network.network

This package (along with its subpackages) contains the classes used to model energy systems. An energy system is modelled as a graph/network of entities with very specific constraints on which types of entities are allowed to be connected.

This file is part of project oemof (github.com/oemof/oemof). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location oemof/oemof/network.py

SPDX-FileCopyrightText: Stephan Günther <> SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de> SPDX-FileCopyrightText: Simon Hilpert <> SPDX-FileCopyrightText: Cord Kaldemeyer <> SPDX-FileCopyrightText: Patrik Schönfeldt <patrik.schoenfeldt@dlr.de>

SPDX-License-Identifier: MIT

class oemof.network.network.**Bus**(**args*, ***kwargs*)

Bases: *oemof.network.network.Node*

class oemof.network.network.**Component**(**args*, ***kwargs*)

Bases: *oemof.network.network.Node*

class oemof.network.network.**Edge**(*input=None*, *output=None*, *flow=None*, *values=None*, ***kwargs*)

Bases: *oemof.network.network.Node*

Bus`es/:class:`Component`s are always connected by an :class:`Edge.

Edge`s connect a single non-:class:`Edge Node with another. They are directed and have a (sequence of) value(s) attached to them so they can be used to represent a flow from a source/an input to a target/an output.

Parameters

- **input, output** (*Bus* or *Component*, optional)
- **flow, values** (*object, optional*) – The (list of) object(s) representing the values flowing from this edge's input into its output. Note that these two names are aliases of each other, so *flow* and *values* are mutually exclusive.
- **Note that all of these parameters are also set as attributes with the same**
- **name.**

Label

alias of *oemof.network.network.EdgeLabel*

property flow

classmethod from_object(*o*)

Creates an *Edge* instance from a single object.

This method inspects its argument and does something different depending on various cases:

- If *o* is an instance of *Edge*, *o* is returned unchanged.
- If *o* is a *Mapping*, the instance is created by calling *cls(**o)*,
- In all other cases, *o* will be used as the *values* keyword argument to `Edge`s constructor.

property input

property output

class oemof.network.network.**EdgeLabel**(*input*, *output*)

Bases: tuple

property input

Alias for field number 0

property output

Alias for field number 1

class oemof.network.network.**Inputs**(*target*)

Bases: collections.abc.MutableMapping

A special helper to map *n1.inputs[n2]* to *n2.outputs[n1]*.

class oemof.network.network.**Metaclass**

Bases: type

The metaclass for objects in an oemof energy system.

property registry

class oemof.network.network.**Node**(**args*, ***kwargs*)

Bases: object

Represents a Node in an energy system graph.

Abstract superclass of the two general types of nodes of an energy system graph, collecting attributes and operations common to all types of nodes. Users should neither instantiate nor subclass this, but use *Component*, *Bus*, *Edge* or one of their subclasses instead.

Parameters

- **label** (*hashable*, optional) – Used as the string representation of this node. If this parameter is not an instance of str it will be converted to a string and the result will be used as this node's *label*, which should be unique with respect to the other nodes in the energy system graph this node belongs to. If this parameter is not supplied, the string representation of this node will instead be generated based on this nodes *class* and *id*.
- **inputs** (*list or dict, optional*) – Either a list of this nodes' input nodes or a dictionary mapping input nodes to corresponding inflows (i.e. input values).
- **outputs** (*list or dict, optional*) – Either a list of this nodes' output nodes or a dictionary mapping output nodes to corresponding outflows (i.e. output values).

Variables **__slots__** (*str or iterable of str*) – See the Python documentation on __slots__ for more information.

property inputs

dict: Dictionary mapping input *Nodes* n to Edge`s from :obj:`n into self. If self is an *Edge*, returns a dict containing the *Edge*'s single input node as the key and the flow as the value.

property label

If this node was given a *label* on construction, this attribute holds the actual object passed as a parameter. Otherwise **:py:`node.label`** is a synonym for **:py:`str(node)`**.

property outputs

dict: Dictionary mapping output *Nodes* n to Edges from self into n. If self is an *Edge*, returns a dict containing the *Edge*'s single output node as the key and the flow as the value.

register()

```
registry_warning = FutureWarning('\nAutomatic registration of `Node`s is deprecated
in favour of\nexplicitly adding `Node`s to an `EnergySystem` via
`EnergySystem.add`.\nThis feature, i.e. the `Node.registry` attribute and
functionality\npertaining to it, will be removed in future versions.\n')
```

class oemof.network.network.**Outputs**(*source*)

Bases: collections.UserDict

Helper that intercepts modifications to update *Inputs* symmetrically.

class oemof.network.network.**Sink**(**args*, ***kwargs*)

Bases: *oemof.network.network.Component*

class oemof.network.network.**Source**(**args*, ***kwargs*)

Bases: *oemof.network.network.Component*

class oemof.network.network.**Transformer**(**args*, ***kwargs*)

Bases: *oemof.network.network.Component*

oemof.network.network.**registry_changed_to**(*r*)

Override registry during execution of a block and restore it afterwards.

oemof.network.network.**temporarily_modifies_registry**(*f*)

Decorator that disables *Node* registration during *f*'s execution.

It does so by setting *Node.registry* to *None* while *f* is executing, so *f* can freely set *Node.registry* to something else. The registration's original value is restored afterwards.

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

oemof.network could always use more documentation, whether as part of the official oemof.network docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/oemof/oemof.network/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *oemof.network* for local development:

1. Fork oemof.network (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:oemof/oemof.network.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER

SIX

AUTHORS

(alphabetic order)

- Cord Kaldemeyer
- Patrik Schönfeldt
- Simon Hilpert
- Stephan Günther
- Uwe Krien

CHAPTER

SEVEN

CHANGELOG

7.1 0.4.0.dev0 (2020-04-01)

- First release on PyPI.

7.2 0.4.0 (2022-04-26)

- Improved imports
- Improved testing
- Explicitly defined API

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

R

S

T

V