oemof.solph

Release 0.4.1

Jun 24, 2020

Contents

CHAPTER 1

Overview

docs

tests

package

1.1 Introduction

The oemof.solph package is part of the Open energy modelling framework (oemof). This an organisational framework to bundle tools for energy (system) modelling. oemof-solph is a model generator for energy system modelling and optimisation.

The `oemof.solph` package is very often called just `oemof` as it was part of the oemof meta package. Now you need to install `oemof.solph` separately, but everything else is still the same. Since v0.4.0. it is not possible to install just oemof, use `pip install oemof.solph` instead.

Everybody is welcome to use and/or develop oemof.solph. Read our contribution section.

Contribution is already possible on a low level by simply fixing typos in oemof's documentation or rephrasing sections which are unclear. If you want to support us that way please fork the oemof repository to your own github account and make changes as described in the github guidelines: https://guides.github.com/activities/hello-world/

If you have questions regarding the use of oemof you can visit the forum at openmod-initiative.org and open a new thread if your questions hasn't been already answered.

Keep in touch! - You can become a watcher at our github site, but this will bring you quite a few mails and might be more interesting for developers. If you just want to get the latest news, like when is the next oemof meeting, you can follow our news-blog at oemof.org.

1.2 Documentation

The oemof.solph documentation is powered by readthedocs. Use the project site of oemof.solph to choose the version of the documentation. Go to the download page to download different versions and formats (pdf, html, epub) of the documentation.

1.3 Installation

If you have a working Python3 environment, use pypi to install the latest oemof version. Python >= 3.6 is recommended. Lower versions may work but are not tested.

```
pip install oemof.solph
```

If you want to use the latest features, you might want to install the **developer version**. See section 'Developing oemof' for more information. The developer version is not recommended for productive use:

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

For running an oemof-solph optimisation model, you need to install a solver. Following you will find guidelines for the installation process for different operation systems.

1.3.1 Installing a solver

There are various commercial and open-source solvers that can be used with oemof. There are two common Open-Source solvers available (CBC, GLPK), while oemof recommends CBC (Coin-or branch and cut). But sometimes its worth comparing the results of different solvers. Other commercial solvers like Gurobi or Cplex can be used as well. Have a look at the pyomo docs to learn about which solvers are supported.

Check the solver installation by executing the test_installation example below (section *Installation test*).

Linux

To install the solvers have a look at the package repository of your Linux distribution or search for precompiled packages. GLPK and CBC ares available at Debian, Feodora, Ubuntu and others.

Windows

1. Download CBC (64 or 32 bit)

2. Download GLPK (64/32 bit)
3. Unpack CBC/GLPK to any folder (e.g. C:/Users/Somebody/my_programs)
4. Add the path of the executable files of both solvers to the PATH variable using this tutorial
5. Restart Windows

Check the solver installation by executing the test_installation example (see the *Installation test* section).

Mac OSX

Please follow the installation instructions on the respective homepages for details.

CBC-solver: https://projects.coin-or.org/Cbc

GLPK-solver: http://arnab-deka.com/posts/2010/02/installing-glpk-on-a-mac/

If you install the CBC solver via brew (highly recommended), it should work without additional configuration.

1.3.2 Installation test

Test the installation and the installed solver by running the installation test in your virtual environment:

```
oemof_installation_test
```

If the installation was successful, you will receive something like this:

```
*********
Solver installed with oemof:
glpk: working
cplex: not working
cbc: working
gurobi: not working
*********
oemof.solph successfully installed.
```

as an output.

1.4 Contributing

A warm welcome to all who want to join the developers and contribute to oemof.solph. Information on the details and how to approach us can be found in the documentation .

1.5 Citing

The core ideas of oemof are described in DOI:10.1016/j.esr.2018.07.001 (preprint at arXiv:1808.0807). To allow citing specific versions of oemof, we use the zenodo project to get a DOI for each version.

1.6 Examples

The linkage of specific modules of the various packages is called an application (app) and depicts for example a concrete energy system model. You can find a large variety of helpful examples in oemof's example repository on github to download or clone. The examples show optimisations of different energy systems and are supposed to

help new users to understand the framework's structure. There is some elaboration on the examples in the respective repository. The repository has sections for each major release.

You are welcome to contribute your own examples via a pull request or by sending us an e-mail (see here for contact information).

1.7 License

CHAPTER 2

User's guide

Solph is an oemof-package, designed to create and solve linear or mixed-integer linear optimization problems. The package is based on pyomo. To create an energy system model generic and specific components are available. To get started with solph, checkout the examples in the *Examples* section.

This User's guide provides a user-friendly introduction into oemof-solph, which includes small examples and nice illustrations. However, the functionality of oemof-solph go beyond the content of this User's guide section. So, if you want to know all details of a certain component or a function, please go the *API Reference*. There, you will find a detailed and complete description of all oemof-solph modules.

2.1 How can I use solph?

To use solph you have to install oemof and at least one solver (see *Installation*), which can be used together with pyomo (e.g. CBC, GLPK, Gurobi, Cplex). See the pyomo installation guide for all supported solver. You can test it by executing one of the existing examples (see *Examples*, or directly oemof's example repository). Be aware that the examples require the CBC solver but you can change the solver name in the example files to your solver.

Once the example work you are close to your first energy model.

2.1.1 Handling of Warnings

The solph library is designed to be as generic as possible to make it possible to use it in different use cases. This concept makes it difficult to raise Error or Warnings because sometimes untypical combinations of parameters are allowed even though they might be wrong in over 99% of the use cases.

Therefore, a SuspiciousUsageWarning was introduced. This warning will warn you if you do something untypical. If you are sure that you know what you are doing you can switch the warning off.

See the debugging module of oemof-tools for more information.

2.1.2 Set up an energy system

In most cases an EnergySystem object is defined when we start to build up an energy system model. The EnergySystem object will be the main container for the model.

To define an EnergySystem we need a Datetime index to define the time range and increment of our model. An easy way to this is to use the pandas time_range function. The following code example defines the year 2011 in hourly steps. See pandas date_range guide for more information.

```
import pandas as pd
my_index = pd.date_range('1/1/2011', periods=8760, freq='H')
```

This index can be used to define the EnergySystem:

```
import oemof.solph as solph
my_energysystem = solph.EnergySystem(timeindex=my_index)
```

Now you can start to add the components of the network.

2.1.3 Add components to the energy system

After defining an instance of the EnergySystem class you have to add all nodes you define in the following to your EnergySystem.

Basically, there are two types of *nodes* - *components* and *buses*. Every Component has to be connected with one or more *buses*. The connection between a *component* and a *bus* is the *flow*.

All solph *components* can be used to set up an energy system model but you should read the documentation of each *component* to learn about usage and restrictions. For example it is not possible to combine every *component* with every *flow*. Furthermore, you can add your own *components* in your application (see below) but we would be pleased to integrate them into solph if they are of general interest. To do so please use the module oemof.solph.custom as described here: http://oemof.readthedocs.io/en/latest/developing_oemof.html#contribute-to-new-components

An example of a simple energy system shows the usage of the nodes for real world representations:

The figure shows a simple energy system using the four basic network classes and the Bus class. If you remove the transmission line (transport 1 and transport 2) you get two systems but they are still one energy system in terms of solph and will be optimised at once.

There are different ways to add components to an *energy system*. The following line adds a *bus* object to the *energy system* defined above.

```
my_energysystem.add(solph.Bus())
```

It is also possible to assign the bus to a variable and add it afterwards. In that case it is easy to add as many objects as you like.

```
my_bus1 = solph.Bus()
my_bus2 = solph.Bus()
my_energysystem.add(my_bus1, my_bus2)
```

Therefore it is also possible to add lists or dictionaries with components but you have to dissolve them.

```
# add a list
my_energysystem.add(*my_list)

# add a dictionary
my_energysystem.add(*my_dictionary.values())
```

Bus

All flows into and out of a *bus* are balanced. Therefore an instance of the Bus class represents a grid or network without losses. To define an instance of a Bus only a unique label is necessary. If you do not set a label a random label is used but this makes it difficult to get the results later on.

To make it easier to connect the bus to a component you can optionally assign a variable for later use.

```
solph.Bus(label='natural_gas')
electricity_bus = solph.Bus(label='electricity')
```

Note: See the *Bus* class for all parameters and the mathematical background.

Flow

The flow class has to be used to connect. An instance of the Flow class is normally used in combination with the definition of a component. A Flow can be limited by upper and lower bounds (constant or time-dependent) or by summarised limits. For all parameters see the API documentation of the *Flow* class or the examples of the nodes below. A basic flow can be defined without any parameter.

```
solph.Flow()
```

Oemof has different types of *flows* but you should be aware that you cannot connect every *flow* type with every *component*.

Note: See the *Flow* class for all parameters and the mathematical background.

Components

Components are divided in three categories. Basic components (solph.network), additional components (solph.components) and custom components (solph.custom). The custom section was created to lower the entry barrier for new components. Be aware that these components are in an experimental state. Let us know if you have used and tested these components. This is the first step to move them to the components section.

See *Solph components* for a list of all components.

2.1.4 Optimise your energy system

The typical optimisation of an energy system in solph is the dispatch optimisation, which means that the use of the sources is optimised to satisfy the demand at least costs. Therefore, variable cost can be defined for all components. The cost for gas should be defined in the gas source while the variable costs of the gas power plant are caused by operating material. You can deviate from this scheme but you should keep it consistent to make it understandable for others.

Costs do not have to be monetary costs but could be emissions or other variable units.

Furthermore, it is possible to optimise the capacity of different components (see *Using the investment mode*).

```
# set up a simple least cost optimisation
om = solph.Model(my_energysystem)

# solve the energy model using the CBC solver
om.solve(solver='cbc', solve_kwargs={'tee': True})
```

If you want to analyse the lp-file to see all equations and bounds you can write the file to you disc. In that case you should reduce the timesteps to 3. This will increase the readability of the file.

```
# set up a simple least cost optimisation
om = solph.Model(my_energysystem)

# write the lp file for debugging or other reasons
om.write('path/my_model.lp', io_options={'symbolic_solver_labels': True})
```

2.1.5 Analysing your results

If you want to analyse your results, you should first dump your EnergySystem instance, otherwise you have to run the simulation again.

```
my_energysystem.results = processing.results(om)
my_energysystem.dump('my_path', 'my_dump.oemof')
```

If you need the meta results of the solver you can do the following:

```
my_energysystem.results['main'] = processing.results(om)
my_energysystem.results['meta'] = processing.meta_results(om)
my_energysystem.dump('my_path', 'my_dump.oemof')
```

To restore the dump you can simply create an EnergySystem instance and restore your dump into it.

```
import oemof.solph as solph
my_energysystem = solph.EnergySystem()
my_energysystem.restore('my_path', 'my_dump.oemof')
results = my_energysystem.results

# If you use meta results do the following instead of the previous line.
results = my_energysystem.results['main']
meta = my_energysystem.results['meta']
```

If you call dump/restore without any parameters, the dump will be stored as *'es_dump.oemof'* into the *'.oemof/dumps/'* folder created in your HOME directory.

See *Handling Results* to learn how to process, plot and analyse the results.

2.2 Solph components

- *Sink (basic)*
- *Source (basic)*
- *Transformer (basic)*
- *ExtractionTurbineCHP (component)*
- *GenericCHP (component)*
- *Link (custom)*
- *GenericStorage (component)*
- *ElectricalLine (custom)*
- *GenericCAES (custom)*
- *SinkDSM (custom)*

2.2.1 Sink (basic)

A sink is normally used to define the demand within an energy model but it can also be used to detect excesses.

The example shows the electricity demand of the electricity_bus defined above. The *'my_demand_series'* should be sequence of normalised valueswhile the *'nominal_value'* is the maximum demand the normalised sequence is multiplied with. Giving *'my_demand_series'* as parameter *'fix'* means that the demand cannot be changed by the solver.

```
solph.Sink(label='electricity_demand', inputs={electricity_bus: solph.Flow(
    fix=my_demand_series, nominal_value=nominal_demand)})
```

In contrast to the demand sink the excess sink has normally less restrictions but is open to take the whole excess.

```
solph.Sink(label='electricity_excess', inputs={electricity_bus: solph.Flow()})
```

Note: The Sink class is only a plug and provides no additional constraints or variables.

2.2.2 Source (basic)

A source can represent a pv-system, a wind power plant, an import of natural gas or a slack variable to avoid creating an in-feasible model.

While a wind power plant will have an hourly feed-in depending on the weather conditions the natural_gas import might be restricted by maximum value (*nominal_value*) and an annual limit (*summed_max*). As we do have to pay for imported gas we should set variable costs. Comparable to the demand series an *fix* is used to define a fixed the normalised output of a wind power plant. Alternatively, you might use *max* to allow for easy curtailment. The *nominal_value* sets the installed capacity.

```
solph.Source(
    label='import_natural_gas',
    outputs={my_energysystem.groups['natural_gas']: solph.Flow(
        nominal_value=1000, summed_max=1000000, variable_costs=50)})

solph.Source(label='wind', outputs={electricity_bus: solph.Flow(
    fix=wind_power_feedin_series, nominal_value=1000000)})
```

Note: The Source class is only a plug and provides no additional constraints or variables.

2.2.3 Transformer (basic)

An instance of the Transformer class can represent a node with multiple input and output flows such as a power plant, a transport line or any kind of a transforming process as electrolysis, a cooling device or a heat pump. The efficiency has to be constant within one time step to get a linear transformation. You can define a different efficiency for every time step (e.g. the thermal powerplant efficiency according to the ambient temperature) but this series has to be predefined and cannot be changed within the optimisation.

A condensing power plant can be defined by a transformer with one input (fuel) and one output (electricity).

```
b_gas = solph.Bus(label='natural_gas')
b_el = solph.Bus(label='electricity')

solph.Transformer(
    label="pp_gas",
    inputs={bgas: solph.Flow()},
    outputs={b_el: solph.Flow(nominal_value=10e10)},
    conversion_factors={electricity_bus: 0.58})
```

A CHP power plant would be defined in the same manner but with two outputs:

```
b_gas = solph.Bus(label='natural_gas')
b_el = solph.Bus(label='electricity')
b_th = solph.Bus(label='heat')

solph.Transformer(
    label='pp_chp',
    inputs={b_gas: Flow()},
    outputs={b_el: Flow(nominal_value=30),
             b_th: Flow(nominal_value=40)},
    conversion_factors={b_el: 0.3, b_th: 0.4})
```

A CHP power plant with 70% coal and 30% natural gas can be defined with two inputs and two outputs:

```
b_gas = solph.Bus(label='natural_gas')
b_coal = solph.Bus(label='hard_coal')
b_el = solph.Bus(label='electricity')
b_th = solph.Bus(label='heat')

solph.Transformer(
    label='pp_chp',
    inputs={b_gas: Flow(), b_coal: Flow()},
    outputs={b_el: Flow(nominal_value=30),
             b_th: Flow(nominal_value=40)},
    conversion_factors={b_el: 0.3, b_th: 0.4,
                        b_coal: 0.7, b_gas: 0.3})
```

A heat pump would be defined in the same manner. New buses are defined to make the code cleaner:

```
b_el = solph.Bus(label='electricity')
b_th_low = solph.Bus(label='low_temp_heat')
b_th_high = solph.Bus(label='high_temp_heat')

# The cop (coefficient of performance) of the heat pump can be defined as
# a scalar or a sequence.
cop = 3

solph.Transformer(
    label='heat_pump',
    inputs={b_el: Flow(), b_th_low: Flow()},
    outputs={b_th_high: Flow()},
    conversion_factors={b_el: 1/cop,
                        b_th_low: (cop-1)/cop})
```

If the low-temperature reservoir is nearly infinite (ambient air heat pump) the low temperature bus is not needed and, therefore, a Transformer with one input is sufficient.

Note: See the `Transformer` class for all parameters and the mathematical background.

2.2.4 ExtractionTurbineCHP (component)

The `ExtractionTurbineCHP` inherits from the *Transformer (basic)* class. Like the name indicates, the application example for the component is a flexible combined heat and power (chp) plant. Of course, an instance of this class can represent also another component with one input and two output flows and a flexible ratio between these flows, with the following constraints:

$$(1)\dot{H}_{Fuel}(t) = \frac{P_{el}(t) + \dot{Q}_{th}(t) \cdot \beta(t)}{\eta_{el,woExtr}(t)}$$

$$(2)P_{el}(t) \geq \dot{Q}_{th}(t) \cdot C_b = \dot{Q}_{th}(t) \cdot \frac{\eta_{el,maxExtr}(t)}{\eta_{th,maxExtr}(t)}$$

where β is defined as:

$$\beta(t) = \frac{\eta_{el,woExtr}(t) - \eta_{el,maxExtr}(t)}{\eta_{th,maxExtr}(t)}$$

where the first equation is the result of the relation between the input flow and the two output flows, the second equation stems from how the two output flows relate to each other, and the symbols used are defined as follows (with Variables (V) and Parameters (P)):

symbol	attribute	type	explanation
$\dot{H}_{Fuel}$	`flow[i, n, t]`	V	fuel input flow
P_{el}	`flow[n, main_output, t]`	V	electric power
$\dot{Q}_{th}$	`flow[n, tapped_output, t]`	V	thermal output
β	`main_flow_loss_index[n, t]`	P	power loss index
$\eta_{el,woExtr}$	`conversion_factor_full_condensation[n, t]`	P	electric efficiency without heat extraction
$\eta_{el,maxExtr}$	`conversion_factors[main_output][n, t]`	P	electric efficiency with max heat extraction
$\eta_{th,maxExtr}$	`conversion_factors[tapped_output][n, t]`	P	thermal efficiency with maximal heat extraction

These constraints are applied in addition to those of a standard `Transformer`. The constraints limit the range of the possible operation points, like the following picture shows. For a certain flow of fuel, there is a line of operation points, whose slope is defined by the power loss factor β (in some contexts also referred to as C_v). The second constraint limits the decrease of electrical power and incorporates the backpressure coefficient C_b.

For now, `ExtractionTurbineCHP` instances must have one input and two output flows. The class allows the definition of a different efficiency for every time step that can be passed as a series of parameters that are fixed before the optimisation. In contrast to the `Transformer`, a main flow and a tapped flow is defined. For the main flow you can define a separate conversion factor that applies when the second flow is zero (*'conversion_factor_full_condensation'*).

```
solph.ExtractionTurbineCHP(
    label='variable_chp_gas',
    inputs={b_gas: solph.Flow(nominal_value=10e10)},
    outputs={b_el: solph.Flow(), b_th: solph.Flow()},
    conversion_factors={b_el: 0.3, b_th: 0.5},
    conversion_factor_full_condensation={b_el: 0.5})
```

The key of the parameter *'conversion_factor_full_condensation'* defines which of the two flows is the main flow. In the example above, the flow to the Bus *'b_el'* is the main flow and the flow to the Bus *'b_th'* is the tapped flow. The following plot shows how the variable chp (right) schedules it's electrical and thermal power production in contrast to a fixed chp (left). The plot is the output of an example in the example repository.

Note: See the *ExtractionTurbineCHP* class for all parameters and the mathematical background.

2.2.5 GenericCHP (component)

With the GenericCHP class it is possible to model different types of CHP plants (combined cycle extraction turbines, back pressure turbines and motoric CHP), which use different ranges of operation, as shown in the figure below.

Combined cycle extraction turbines: The minimal and maximal electric power without district heating (red dots in the figure) define maximum load and minimum load of the plant. Beta defines electrical power loss through heat extraction. The minimal thermal condenser load to cooling water and the share of flue gas losses at maximal heat extraction determine the right boundary of the operation range.

```
solph.components.GenericCHP(
    label='combined_cycle_extraction_turbine',
    fuel_input={bgas: solph.Flow(
        H_L_FG_share_max=[0.19 for p in range(0, periods)])},
    electrical_output={bel: solph.Flow(
        P_max_woDH=[200 for p in range(0, periods)],
        P_min_woDH=[80 for p in range(0, periods)],
        Eta_el_max_woDH=[0.53 for p in range(0, periods)],
        Eta_el_min_woDH=[0.43 for p in range(0, periods)])},
    heat_output={bth: solph.Flow(
        Q_CW_min=[30 for p in range(0, periods)])},
    Beta=[0.19 for p in range(0, periods)],
    back_pressure=False)
```

For modeling a back pressure CHP, the attribute *back_pressure* has to be set to True. The ratio of power and heat production in a back pressure plant is fixed, therefore the operation range is just a line (see figure). Again, the *P_min_woDH* and *P_max_woDH*, the efficiencies at these points and the share of flue gas losses at maximal heat extraction have to be specified. In this case "without district heating" is not to be taken literally since an operation without heat production is not possible. It is advised to set *Beta* to zero, so the minimal and maximal electric power without district heating are the same as in the operation point (see figure). The minimal thermal condenser load to cooling water has to be zero, because there is no condenser besides the district heating unit.

```
solph.components.GenericCHP(
    label='back_pressure_turbine',
    fuel_input={bgas: solph.Flow(
        H_L_FG_share_max=[0.19 for p in range(0, periods)])},
```

(continues on next page)

(continued from previous page)

```
    electrical_output={bel: solph.Flow(
        P_max_woDH=[200 for p in range(0, periods)],
        P_min_woDH=[80 for p in range(0, periods)],
        Eta_el_max_woDH=[0.53 for p in range(0, periods)],
        Eta_el_min_woDH=[0.43 for p in range(0, periods)])},
    heat_output={bth: solph.Flow(
        Q_CW_min=[0 for p in range(0, periods)])},
    Beta=[0 for p in range(0, periods)],
    back_pressure=True)
```

A motoric chp has no condenser, so *Q_CW_min* is zero. Electrical power does not depend on the amount of heat used so *Beta* is zero. The minimal and maximal electric power (without district heating) and the efficiencies at these points are needed, whereas the use of electrical power without using thermal energy is not possible. With *Beta=0* there is no difference between these points and the electrical output in the operation range. As a consequence of the functionality of a motoric CHP, share of flue gas losses at maximal heat extraction but also at minimal heat extraction have to be specified.

```
solph.components.GenericCHP(
    label='motoric_chp',
    fuel_input={bgas: solph.Flow(
        H_L_FG_share_max=[0.18 for p in range(0, periods)],
        H_L_FG_share_min=[0.41 for p in range(0, periods)])},
    electrical_output={bel: solph.Flow(
        P_max_woDH=[200 for p in range(0, periods)],
        P_min_woDH=[100 for p in range(0, periods)],
        Eta_el_max_woDH=[0.44 for p in range(0, periods)],
        Eta_el_min_woDH=[0.40 for p in range(0, periods)])},
    heat_output={bth: solph.Flow(
        Q_CW_min=[0 for p in range(0, periods)])},
    Beta=[0 for p in range(0, periods)],
    back_pressure=False)
```

Modeling different types of plants means telling the component to use different constraints. Constraint 1 to 9 are active in all three cases. Constraint 10 depends on the attribute back_pressure. If true, the constraint is an equality, if not it is a less or equal. Constraint 11 is only needed for modeling motoric CHP which is done by setting the attribute *H_L_FG_share_min*.

$$
\begin{aligned}
&(1) && \dot{H}_F(t) = fuel\ input \\
&(2) && \dot{Q}(t) = heat\ output \\
&(3) && P_{el}(t) = power\ output \\
&(4) && \dot{H}_F(t) = \alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot P_{el,woDH}(t) \\
&(5) && \dot{H}_F(t) = \alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot (P_{el}(t) + \beta \cdot \dot{Q}(t)) \\
&(6) && \dot{H}_F(t) \leq Y(t) \cdot \frac{P_{el,max,woDH}(t)}{\eta_{el,max,woDH}(t)} \\
&(7) && \dot{H}_F(t) \geq Y(t) \cdot \frac{P_{el,min,woDH}(t)}{\eta_{el,min,woDH}(t)} \\
&(8) && \dot{H}_{L,FG,max}(t) = \dot{H}_F(t) \cdot \dot{H}_{L,FG,sharemax}(t) \\
&(9) && \dot{H}_{L,FG,min}(t) = \dot{H}_F(t) \cdot \dot{H}_{L,FG,sharemin}(t) \\
&(10) && P_{el}(t) + \dot{Q}(t) + \dot{H}_{L,FG,max}(t) + \dot{Q}_{CW,min}(t) \cdot Y(t) = / \leq \dot{H}_F(t)
\end{aligned}
$$

where $= / \leq$ depends on the CHP being back pressure or not.

The coefficients α_0 and α_1 can be determined given the efficiencies maximal/minimal load:

$$\eta_{el,max,woDH}(t) = \frac{P_{el,max,woDH}(t)}{\alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot P_{el,max,woDH}(t)}$$
$$\eta_{el,min,woDH}(t) = \frac{P_{el,min,woDH}(t)}{\alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot P_{el,min,woDH}(t)}$$

If $\dot{H}_{L,FG,min}$ is given, e.g. for a motoric CHP:

$$P_{el}(t) + \dot{Q}(t) + \dot{H}_{L,FG,min}(t) + \dot{Q}_{CW,min}(t) \cdot Y(t) \geq \dot{H}_F(t) \tag{11}$$

The symbols used are defined as follows (with Variables (V) and Parameters (P)):

math. symbol	attribute	type	explanation
$\dot{H}_F$	`H_F[n,t]`	V	input of enthalpy through fuel input
P_{el}	`P[n,t]`	V	provided electric power
$P_{el,woDH}$	`P_woDH[n,t]`	V	electric power without district heating
$P_{el,min,woDH}$	`P_min_woDH[n,t]`	P	min. electric power without district heating
$P_{el,max,woDH}$	`P_max_woDH[n,t]`	P	max. electric power without district heating
$\dot{Q}$	`Q[n,t]`	V	provided heat
$\dot{Q}_{CW,min}$	`Q_CW_min[n,t]`	P	minimal therm. condenser load to cooling water
$\dot{H}_{L,FG,min}$	`H_L_FG_min[n,t]`	V	flue gas enthalpy loss at min heat extraction
$\dot{H}_{L,FG,max}$	`H_L_FG_max[n,t]`	V	flue gas enthalpy loss at max heat extraction
$\dot{H}_{L,FG,sharemin}$	`H_L_FG_share_min[n,t]`	P	share of flue gas loss at min heat extraction
$\dot{H}_{L,FG,sharemax}$	`H_L_FG_share_max[n,t]`	P	share of flue gas loss at max heat extraction
Y	`Y[n,t]`	V	status variable on/off
α_0	`n.alphas[0][n,t]`	P	coefficient describing efficiency
α_1	`n.alphas[1][n,t]`	P	coefficient describing efficiency
β	`Beta[n,t]`	P	power loss index
$\eta_{el,min,woDH}$	`Eta_el_min_woDH[n,t]`	P	el. eff. at min. fuel flow w/o distr. heating
$\eta_{el,max,woDH}$	`Eta_el_max_woDH[n,t]`	P	el. eff. at max. fuel flow w/o distr. heating

Note: See the *GenericCHP* class for all parameters and the mathematical background.

2.2.6 GenericStorage (component)

In contrast to the three classes above the storage class is a pure solph class and is not inherited from the oemof-network module. The `nominal_storage_capacity` of the storage signifies the storage capacity. You can either set it to the net capacity or to the gross capacity and limit it using the min/max attribute. To limit the input and output flows, you can define the `nominal_value` in the Flow objects. Furthermore, an efficiency for loading, unloading and a loss rate can be defined.

```
solph.GenericStorage(
    label='storage',
    inputs={b_el: solph.Flow(nominal_value=9, variable_costs=10)},
    outputs={b_el: solph.Flow(nominal_value=25, variable_costs=10)},
    loss_rate=0.001, nominal_storage_capacity=50,
    inflow_conversion_factor=0.98, outflow_conversion_factor=0.8)
```

For initialising the state of charge before the first time step (time step zero) the parameter `initial_storage_level` (default value: `None`) can be set by a numeric value as fraction of the storage capacity. Additionally the parameter `balanced` (default value: `True`) sets the relation of the state of charge of time step zero and the last time step. If `balanced=True`, the state of charge in the last time step is equal to initial value in time step zero. Use `balanced=False` with caution as energy might be added to or taken from the energy system due to different states of charge in time step zero and the last time step. Generally, with these two parameters four configurations are possible, which might result in different solutions of the same optimization model:

- `initial_storage_level=None`, `balanced=True` (default setting): The state of charge in time step zero is a result of the optimization. The state of charge of the last time step is equal to time step zero. Thus, the storage is not violating the energy conservation by adding or taking energy from the system due to different states of charge at the beginning and at the end of the optimization period.
- `initial_storage_level=0.5`, `balanced=True`: The state of charge in time step zero is fixed to 0.5 (50 % charged). The state of charge in the last time step is also constrained by 0.5 due to the coupling parameter `balanced` set to `True`.
- `initial_storage_level=None`, `balanced=False`: Both, the state of charge in time step zero and the last time step are a result of the optimization and not coupled.
- `initial_storage_level=0.5`, `balanced=False`: The state of charge in time step zero is constrained by a given value. The state of charge of the last time step is a result of the optimization.

The following code block shows an example of the storage parametrization for the second configuration:

```
solph.GenericStorage(
    label='storage',
    inputs={b_el: solph.Flow(nominal_value=9, variable_costs=10)},
    outputs={b_el: solph.Flow(nominal_value=25, variable_costs=10)},
    loss_rate=0.001, nominal_storage_capacity=50,
    initial_storage_level=0.5, balanced=True,
    inflow_conversion_factor=0.98, outflow_conversion_factor=0.8)
```

If you want to view the temporal course of the state of charge of your storage after the optimisation, you need to check the `storage_content` in the results:

```
from oemof.solph import processing, views
results = processing.results(om)
column_name = (('your_storage_label', 'None'), 'storage_content')
SC = views.node(results, 'your_storage_label')['sequences'][column_name]
```

The `storage_content` is the absolute value of the current stored energy. By calling:

```
views.node(results, 'your_storage_label')['scalars']
```

you get the results of the scalar values of your storage, e.g. the initial storage content before time step zero (`init_content`).

For more information see the definition of the *GenericStorage* class or check the example repository of oemof.

Using an investment object with the GenericStorage component

Based on the *GenericStorage* object the *GenericInvestmentStorageBlock* adds two main investment possibilities.

- Invest into the flow parameters e.g. a turbine or a pump
- Invest into capacity of the storage e.g. a basin or a battery cell

Investment in this context refers to the value of the variable for the 'nominal_value' (installed capacity) in the investment mode.

As an addition to other flow-investments, the storage class implements the possibility to couple or decouple the flows with the capacity of the storage. Three parameters are responsible for connecting the flows and the capacity of the storage:

- ' *invest_relation_input_capacity* ' fixes the input flow investment to the capacity investment. A ratio of '1' means that the storage can be filled within one time-period.
- ' *invest_relation_output_capacity* ' fixes the output flow investment to the capacity investment. A ratio of '1' means that the storage can be emptied within one period.
- ' *invest_relation_input_output* ' fixes the input flow investment to the output flow investment. For values <1, the input will be smaller and for values >1 the input flow will be larger.

You should not set all 3 parameters at the same time, since it will lead to overdetermination.

The following example pictures a Pumped Hydroelectric Energy Storage (PHES). Both flows and the storage itself (representing: pump, turbine, basin) are free in their investment. You can set the parameters to *None* or delete them as *None* is the default value.

```
solph.GenericStorage(
    label='PHES',
    inputs={b_el: solph.Flow(investment= solph.Investment(ep_costs=500))},
    outputs={b_el: solph.Flow(investment= solph.Investment(ep_costs=500)},
    loss_rate=0.001,
    inflow_conversion_factor=0.98, outflow_conversion_factor=0.8),
    investment = solph.Investment(ep_costs=40))
```

The following example describes a battery with flows coupled to the capacity of the storage.

```
solph.GenericStorage(
    label='battery',
    inputs={b_el: solph.Flow()},
    outputs={b_el: solph.Flow()},
    loss_rate=0.001,
    inflow_conversion_factor=0.98,
     outflow_conversion_factor=0.8,
    invest_relation_input_capacity = 1/6,
    invest_relation_output_capacity = 1/6,
    investment = solph.Investment(ep_costs=400))
```

Note: See the `GenericStorage` class for all parameters and the mathematical background.

2.2.7 OffsetTransformer (component)

The *OffsetTransformer* object makes it possible to create a Transformer with different efficiencies in part load condition. For this object it is necessary to define the inflow as a nonconvex flow and to set a minimum load. The following

example illustrates how to define an OffsetTransformer for given information for the output:

```
eta_min = 0.5        # efficiency at minimal operation point
eta_max = 0.8        # efficiency at nominal operation point
P_out_min = 20       # absolute minimal output power
P_out_max = 100      # absolute nominal output power

# calculate limits of input power flow
P_in_min = P_out_min / eta_min
P_in_max = P_out_max / eta_max

# calculate coefficients of input-output line equation
c1 = (P_out_max-P_out_min)/(P_in_max-P_in_min)
c0 = P_out_max - c1*P_in_max

# define OffsetTransformer
solph.custom.OffsetTransformer(
    label='boiler',
    inputs={bfuel: solph.Flow(
        nominal_value=P_in_max,
        max=1,
        min=P_in_min/P_in_max,
        nonconvex=solph.NonConvex())},
    outputs={bth: solph.Flow()},
    coefficients = [c0, c1])
```

This example represents a boiler, which is supplied by fuel and generates heat. It is assumed that the nominal thermal power of the boiler (output power) is 100 (kW) and the efficiency at nominal power is 80 %. The boiler cannot operate under 20 % of nominal power, in this case 20 (kW) and the efficiency at that part load is 50 %. Note that the nonconvex flow has to be defined for the input flow. By using the OffsetTransformer a linear relation of in- and output power with a power dependent efficiency is generated. The following figures illustrate the relations:

Now, it becomes clear, why this object has been named *OffsetTransformer*. The linear equation of in- and outflow does not hit the origin, but is offset. By multiplying the Offset C_0 with the binary status variable of the nonconvex flow, the origin (0, 0) becomes part of the solution space and the boiler is allowed to switch off:

$$P_{out}(t) = C_1(t) \cdot P_{in}(t) + C_0(t) \cdot Y(t)$$

Table 1: Variables (V) and Parameters (P)

symbol	attribute	type	explanation
$P_{out}(t)$	`flow[n, o, t]`	V	Power of output
$P_{in}(t)$	`flow[i, n, t]`	V	Power of input
$Y(t)$	`status[i, n, t]`	V	binary status variable of nonconvex input flow
$C_1(t)$	`coefficients[1][n, t]`	P	linear coefficient 1 (slope)
$C_0(t)$	`coefficients[0][n, t]`	P	linear coefficient 0 (y-intersection)

The following figures shows the efficiency dependent on the output power, which results in a nonlinear relation:

$$\eta = C_1 \cdot P_{out}(t) / (P_{out}(t) - C_0)$$

The parameters C_0 and C_1 can be given by scalars or by series in order to define a different efficiency equation for every timestep.

Note: See the `OffsetTransformer` class for all parameters and the mathematical background.

2.2.8 ElectricalLine (custom)

Electrical line.

Note: See the `ElectricalLine` class for all parameters and the mathematical background.

2.2.9 GenericCAES (custom)

Compressed Air Energy Storage (CAES). The following constraints describe the CAES:

$$(1) \quad P_{cmp}(t) = electrical\_input(t) \quad \forall t \in T$$

$$(2) \quad P_{cmp\_max}(t) = m_{cmp\_max} \cdot CAS_{fil}(t-1) + b_{cmp\_max} \quad \forall t \in [1, t_{max}]$$

$$(3) \quad P_{cmp\_max}(t) = b_{cmp\_max} \quad \forall t \notin [1, t_{max}]$$

$$(4) \quad P_{cmp}(t) \leq P_{cmp\_max}(t) \quad \forall t \in T$$

$$(5) \quad P_{cmp}(t) \geq P_{cmp\_min} \cdot ST_{cmp}(t) \quad \forall t \in T$$

$$(6) \quad P_{cmp}(t) = m_{cmp\_max} \cdot CAS_{fil\_max} + b_{cmp\_max} \cdot ST_{cmp}(t) \quad \forall t \in T$$

$$(7) \quad \dot{Q}_{cmp}(t) = m_{cmp\_q} \cdot P_{cmp}(t) + b_{cmp\_q} \cdot ST_{cmp}(t) \quad \forall t \in T$$

$$(8) \quad \dot{Q}_{cmp}(t) = \dot{Q}_{cmp_out}(t) + \dot{Q}_{tes\_in}(t) \quad \forall t \in T$$

$$(9) \quad r_{cmp\_tes} \cdot \dot{Q}_{cmp\_out}(t) = (1 - r_{cmp\_tes})\, \dot{Q}_{tes\_in}(t) \quad \forall t \in T$$

$$(10) \quad P_{exp}(t) = electrical\_output(t) \quad \forall t \in T$$

$$(11) \quad P_{exp\_max}(t) = m_{exp\_max} CAS_{fil}(t-1) + b_{exp\_max} \quad \forall t \in [1, t_{\max}]$$

$$(12) \quad P_{exp\_max}(t) = b_{exp\_max} \quad \forall t \notin [1, t_{\max}]$$

$$(13) \quad P_{exp}(t) \leq P_{exp\_max}(t) \quad \forall t \in T$$

$$(14) \quad P_{exp}(t) \geq P_{exp\_min}(t) \cdot ST_{exp}(t) \quad \forall t \in T$$

$$(15) \quad P_{exp}(t) \leq m_{exp\_max} \cdot CAS_{fil\_max} + b_{exp\_max} \cdot ST_{exp}(t) \quad \forall t \in T$$

$$(16) \quad \dot{Q}_{exp}(t) = m_{exp\_q} \cdot P_{exp}(t) + b_{cxp\_q} \cdot ST_{cxp}(t) \quad \forall t \in T$$

$$(17) \quad \dot{Q}_{exp\_in}(t) = fuel\_input(t) \quad \forall t \in T$$

$$(18) \quad \dot{Q}_{exp}(t) = \dot{Q}_{exp\_in}(t) + \dot{Q}_{tes\_out}(t) + \dot{Q}_{cxp\_add}(t) \quad \forall t \in T$$

$$(19) \quad r_{exp\_tes} \cdot \dot{Q}_{exp\_in}(t) = (1 - r_{exp\_tes})(\dot{Q}_{tes\_out}(t) + \dot{Q}_{exp\_add}(t)) \quad \forall t \in T$$

$$(20) \quad \dot{E}_{cas\_in}(t) = m_{cas\_in} \cdot P_{cmp}(t) + b_{cas\_in} \cdot ST_{cmp}(t) \quad \forall t \in T$$

$$(21) \quad \dot{E}_{cas\_out}(t) = m_{cas\_out} \cdot P_{cmp}(t) + b_{cas\_out} \cdot ST_{cmp}(t) \quad \forall t \in T$$

$$(22) \quad \eta_{cas\_tmp} \cdot CAS_{fil}(t) = CAS_{fil}(t-1) + \tau \left(\dot{E}_{cas\_in}(t) - \dot{E}_{cas\_out}(t) \right) \quad \forall t \in [1, t_{max}]$$

$$(23) \quad \eta_{cas\_tmp} \cdot CAS_{fil}(t) = \tau \left(\dot{E}_{cas\_in}(t) - \dot{E}_{cas\_out}(t) \right) \quad \forall t \notin [1, t_{max}]$$

$$(24) \quad CAS_{fil}(t) \leq CAS_{fil\_max} \quad \forall t \in T$$

$$(25) \quad TES_{fil}(t) = TES_{fil}(t-1) + \tau \left(\dot{Q}_{tes\_in}(t) - \dot{Q}_{tes\_out}(t) \right) \quad \forall t \in [1, t_{max}]$$

$$(26) \quad TES_{fil}(t) = \tau \left(\dot{Q}_{tes\_in}(t) - \dot{Q}_{tes\_out}(t) \right) \quad \forall t \notin [1, t_{max}]$$

$$(27) \quad TES_{fil}(t) \leq TES_{fil\_max} \quad \forall t \in T$$

Table: Symbols and attribute names of variables and parameters

Table 2: Variables (V) and Parameters (P)

symbol	attribute	type	explanation
ST_{cmp}	`cmp_st[n,t]`	V	Status of compression
P_{cmp}	`cmp_p[n,t]`	V	Compression power
P_{cmp_max}	`cmp_p_max[n,t]`	V	Max. compression power
$\dot{Q}_{cmp}$	`cmp_q_out_sum[n,t]`	V	**Summed** heat flow in compression

Continued on next page

Table 2 – continued from previous page

symbol	attribute	type	explanation
$\dot{Q}_{cmp\_out}$	`cmp_q_waste[n, t]`	V	Waste heat flow from compression
$ST_{exp}(t)$	`exp_st[n,t]`	V	Status of expansion (binary)
$P_{exp}(t)$	`exp_p[n,t]`	V	Expansion power
$P_{exp\_max}(t)$	`exp_p_max[n,t]`	V	Max. expansion power
$\dot{Q}_{exp}(t)$	`exp_q_in_sum[n, t]`	V	Summed heat flow in expansion
$\dot{Q}_{exp\_in}(t)$	`exp_q_fuel_in[n, t]`	V	Heat (external) flow into expansion
$\dot{Q}_{exp\_add}(t)$	`exp_q_add_in[n, t]`	V	Additional heat flow into expansion
$CAV_{fil}(t)$	`cav_level[n,t]`	V	**Filling level** if CAE
$\dot{E}_{cas\_in}(t)$	`cav_e_in[n,t]`	V	Exergy flow into CAS
$\dot{E}_{cas\_out}(t)$	`cav_e_out[n,t]`	V	Exergy flow from CAS
$TES_{fil}(t)$	`tes_level[n,t]`	V	Filling level of Thermal Energy Storage (TES)
$\dot{Q}_{tes\_in}(t)$	`tes_e_in[n,t]`	V	**Heat** flow into TES
$\dot{Q}_{tes\_out}(t)$	`tes_e_out[n,t]`	V	**Heat** flow from TES
b_{cmp_max}	`cmp_p_max_b[n, t]`	P	**Specific** y-intersection
b_{cmp_q}	`cmp_q_out_b[n, t]`	P	Specific y-intersection
b_{exp_max}	`exp_p_max_b[n, t]`	P	Specific y-intersection
b_{exp_q}	`exp_q_in_b[n,t]`	P	Specific y-intersection
b_{cas_in}	`cav_e_in_b[n,t]`	P	Specific y-intersection
b_{cas_out}	`cav_e_out_b[n, t]`	P	Specific y-intersection
m_{cmp_max}	`cmp_p_max_m[n, t]`	P	**Specific** slope
m_{cmp_q}	`cmp_q_out_m[n, t]`	P	**Specific** slope
m_{exp_max}	`exp_p_max_m[n, t]`	P	**Specific** slope
m_{exp_q}	`exp_q_in_m[n,t]`	P	**Specific** slope

Continued on next page

Table 2 – continued from previous page

symbol	attribute	type	explanation
m_{cas_in}	`cav_e_in_m[n,t]`	P	**Specific** slope
m_{cas_out}	`cav_e_out_m[n, t]`	P	**Specific** slope
P_{cmp_min}	`cmp_p_min[n,t]`	P	Min. compression power
r_{cmp_tes}	`cmp_q_tes_share[n, t]`	P	**Ratio** between waste heat flow and heat flow into TES
r_{exp_tes}	`exp_q_tes_share[n, t]`	P	**Ratio** **between external heat flow into expansion** additional source
τ	`m.timeincrement[n, t]`	P	**Time interval** length
TES_{fil_max}	`tes_level_max[n, t]`	P	Max. filling level of TES
CAS_{fil_max}	`cav_level_max[n, t]`	P	**Max.** filling level of TES
τ	`cav_eta_tmp[n, t]`	P	**Temporal efficiency** (loss factor to take intertemporal losses into account)
electrical_input	`flow[list(n.electrical_input.keys())[0], n, t]`	P	Electr. power input into compression
electrical_output	`flow[n, list(n.electrical_output.keys())[0], t]`	P	Electr. power output of expansion
fuel_input	`flow[list(n.fuel_input.keys())[0], n, t]`	P	**Heat input** (external) into Expansion

Note: See the `GenericCAES` class for all parameters and the mathematical background.

2.2.10 Link (custom)

Link.

Note: See the *Link* class for all parameters and the mathematical background.

2.2.11 SinkDSM (custom)

SinkDSM can used to represent flexibility in a demand time series. Elasticity of the demand is described by upper (`capacity_up`) and lower (`capacity_down`) bounds where within the demand is allowed to vary. Upwards shifted demand is then balanced with downwards shifted demand.

At the moment, *SinkDSM* provides two method how the Demand-Side Management (DSM) flexibility is represented in constraints

- "delay": Implementation of the DSM modeling method proposed by Zerrahn & Schill (2015): On the representation of demand-side management in power system models, in: Energy (84), pp. 840-845, 10.1016/j.energy.2015.03.037. Details: *SinkDSMDelayBlock*
- "interval": Is a fairly simple approach. Within a defined windows of time steps, demand can be shifted within the defined bounds of elasticity. The window sequentially moves forwards. Details: *SinkDSMIntervalBlock*

Cost can be associated to either demand up shifts or demand down shifts.

This small example of PV, grid and SinkDSM shows how to use the component

```
# Create some data
pv_day = [(-(1 / 6 * x ** 2) + 6) / 6 for x in range(-6, 7)]
pv_ts = [0] * 6 + pv_day + [0] * 6
data_dict = {"demand_el": [3] * len(pv_ts),
             "pv": pv_ts,
             "Cap_up": [0.5] * len(pv_ts),
             "Cap_do": [0.5] * len(pv_ts)}
data = pd.DataFrame.from_dict(data_dict)

# Do timestamp stuff
datetimeindex = pd.date_range(start='1/1/2013', periods=len(data.index), freq='H')
data['timestamp'] = datetimeindex
data.set_index('timestamp', inplace=True)

# Create Energy System
es = solph.EnergySystem(timeindex=datetimeindex)
Node.registry = es

# Create bus representing electricity grid
b_elec = solph.Bus(label='Electricity bus')

# Create a back supply
grid = solph.Source(label='Grid',
                    outputs={
                        b_elec: solph.Flow(
                            nominal_value=10000,
                            variable_costs=50)}
                    )

# PV supply from time series
```

(continues on next page)

(continued from previous page)

```
s_wind = solph.Source(label='wind',
                      outputs={
                          b_elec: solph.Flow(
                              fix=data['pv'],
                              nominal_value=3.5)}
                      )

# Create DSM Sink
demand_dsm = solph.custom.SinkDSM(label='DSM',
                                  inputs={b_elec: solph.Flow()},
                                  capacity_up=data['Cap_up'],
                                  capacity_down=data['Cap_do'],
                                  delay_time=6,
                                  demand=data['demand_el'],
                                  method="delay",
                                  cost_dsm_down=5)
```

Yielding the following results

Note:

- This component is a candidate component. It's implemented as a custom component for users that like to use and test the component at early stage. Please report issues to improve the component.
- See the *SinkDSM* class for all parameters and the mathematical background.

2.3 Using the investment mode

As described in *Optimise your energy system* the typical way to optimise an energy system is the dispatch optimisation based on marginal costs. Solph also provides a combined dispatch and investment optimisation. Based on investment costs you can compare the usage of existing components against building up new capacity. The annual savings by building up new capacity must therefore compensate the annuity of the investment costs (the time period does not have to be one year but depends on your Datetime index).

See the API of the *Investment* class to see all possible parameters.

Basically an instance of the investment class can be added to a Flow or a Storage. All parameters that usually refer to the *nominal_value/capacity* will now refer to the investment variables and existing capacity. It is also possible to set a maximum limit for the capacity that can be build. If existing capacity is considered for a component with investment mode enabled, the *ep_costs* still apply only to the newly built capacity.

The investment object can be used in Flows and some components. See the *Solph components* section for detailed information of each component.

For example if you want to find out what would be the optimal capacity of a wind power plant to decrease the costs of an existing energy system, you can define this model and add an investment source. The *wind_power_time_series* has to be a normalised feed-in time series of you wind power plant. The maximum value might be caused by limited space for wind turbines.

```
solph.Source(label='new_wind_pp', outputs={electricity: solph.Flow(
    fix=wind_power_time_series,
    investment=solph.Investment(ep_costs=epc, maximum=50000))})
```

Let's slightly alter the case and consider for already existing wind power capacity of 20,000 kW. We're still expecting the total wind power capacity, thus we allow for 30,000 kW of new installations and formulate as follows.

```
solph.Source(label='new_wind_pp', outputs={electricity: solph.Flow(
    fix=wind_power_time_series,
        investment=solph.Investment(ep_costs=epc,
                                    maximum=30000,
                                    existing=20000))})
```

The periodical costs (*ep_costs*) are typically calculated as follows:

```
capex = 1000  # investment cost
lifetime = 20  # life expectancy
wacc = 0.05  # weighted average of capital cost
epc = capex * (wacc * (1 + wacc) ** lifetime) / ((1 + wacc) ** lifetime - 1)
```

This also implemented in `annuity()`. The code above would look like this:

```
from oemof.tools import economics
epc = economics.annuity(1000, 20, 0.05)
```

So far, the investment costs and the installed capacity are mathematically a line through origin. But what if there is a minimum threshold for doing an investment, e.g. you cannot buy gas turbines lower than a certain nominal power, or, the marginal costs of bigger plants decrease. Therefore, you can use the parameter *nonconvex* and *offset* of the investment class. Both, work with investment in flows and storages. Here is an example of an transformer:

```
trafo = solph.Transformer(
    label='transformer_nonconvex',
    inputs={bus_0: solph.Flow()},
    outputs={bus_1: solph.Flow(
        investment=solph.Investment(
            ep_costs=4,
            maximum=100,
            minimum=20,
            nonconvex=True,
            offset=400))},
    conversion_factors={bus_1: 0.9})
```

In this examples, it is assumed, that independent of the size of the transformer, there are always fix investment costs of 400 (€). The minimum investment size is 20 (kW) and the costs per installed unit are 4 (€/kW). With this option, you could theoretically approximate every cost function you want. But be aware that for every nonconvex investment flow or storage you are using, an additional binary variable is created. This might boost your computing time into the limitless.

The following figures illustrates the use of the nonconvex investment flow. Here, $c_{invest,fix}$ is the *offset* value and $c_{invest,var}$ is the *ep_costs* value:

In case of a convex investment (which is the default setting *nonconvex=Flase*), the *minimum* attribute leads to a forced investment, whereas in the nonconvex case, the investment can become zero as well.

The calculation of the specific costs per kilowatt installed capacity results in the following relation for convex and nonconvex investments:

See *InvestmentFlow* and *GenericInvestmentStorageBlock* for all the mathematical background, like variables and constraints, which are used.

Note: At the moment the investment class is not compatible with the MIP classes *NonConvex*.

2.4 Mixed Integer (Linear) Problems

Solph also allows you to model components with respect to more technical details such as a minimal power production. Therefore, the class *NonConvex* exists in the *options* module. Note that the usage of this class is currently not compatible with the *Investment* class.

If you want to use the functionality of the options-module, the only thing you have to do is to invoke a class instance inside your Flow() - declaration:

```
b_gas = solph.Bus(label='natural_gas')
b_el = solph.Bus(label='electricity')
b_th = solph.Bus(label='heat')

solph.Transformer(
    label='pp_chp',
    inputs={b_gas: Flow()},
    outputs={b_el: Flow(nominal_value=30,
                        min=0.5,
                        nonconvex=NonConvex()),
             b_th: Flow(nominal_value=40)},
    conversion_factors={b_el: 0.3, b_th: 0.4})
```

The NonConvex() object of the electrical output of the created LinearTransformer will create a 'status' variable for the flow. This will be used to model for example minimal/maximal power production constraints if the attributes *min*/*max* of the flow are set. It will also be used to include start up constraints and costs if corresponding attributes of the class are provided. For more information see the API of the *NonConvex* class and its corresponding block class `NonConvex`.

Note: The usage of this class can sometimes be tricky as there are many interdenpendencies. So check out the examples and do not hesitate to ask the developers if your model does not work as expected.

2.5 Adding additional constraints

You can add additional constraints to your *Model*. See flexible_modelling in the example repository to learn how to do it.

Some predefined additional constraints can be found in the *constraints* module.

- Emission limit for the model -> *emission_limit()*
- Generic integral limit (general form of emission limit) -> *generic_integral_limit()*
- Coupling of two variables e.g. investment variables) with a factor -> *equate_variables()*
- Overall investment limit -> *investment_limit()*
- Generic investment limit -> *additional_investment_flow_limit()*
- Limit active flow count -> *limit_active_flow_count()*
- Limit active flow count by keyword -> *limit_active_flow_count_by_keyword()*

2.6 The Grouping module (Sets)

To construct constraints, variables and objective expressions inside the *blocks* and the *models* modules, so called groups are used. Consequently, certain constraints are created for all elements of a specific group. Thus, mathematically the groups depict sets of elements inside the model.

The grouping is handled by the solph grouping module *groupings* which is based on the oemof core `groupings` functionality. You do not need to understand how the underlying functionality works. Instead, checkout how the solph grouping module is used to create groups.

The simplest form is a function that looks at every node of the energy system and returns a key for the group depending e.g. on node attributes:

```
 def constraint_grouping(node):
     if isinstance(node, Bus) and node.balanced:
         return blocks.Bus
     if isinstance(node, Transformer):
         return blocks.Transformer
GROUPINGS = [constraint_grouping]
```

This function can be passed in a list to `groupings` of *oemof.solph.network.EnergySystem*. So that we end up with two groups, one with all Transformers and one with all Buses that are balanced. These groups are simply stored in a dictionary. There are some advanced functionalities to group two connected nodes with their connecting flow and others (see for example: `FlowsWithNodes`).

2.7 Using the Excel (csv) reader

Alternatively to a manual creation of energy system component objects as describe above, can also be created from a excel sheet (libreoffice, gnumeric...).

The idea is to create different sheets within one spreadsheet file for different components. Afterwards you can loop over the rows with the attributes in the columns. The name of the columns may differ from the name of the attribute. You may even create two sheets for the GenericStorage class with attributes such as C-rate for batteries or capacity of turbine for a PHES.

Once you have create your specific excel reader you can lower the entry barrier for other users. It is some sort of a GUI in form of platform independent spreadsheet software and to make data and models exchangeable in one archive.

See oemof's example repository for an excel reader example.

2.8 Handling Results

The main purpose of the processing module is to collect and organise results. The views module will provide some typical representations of the results. Plots are not part of solph, because plots are highly individual. However, the provided pandas.DataFrames are a good start for plots. Some basic functions for plotting of optimisation results can be found in the separate repository oemof_visio.

The `processing.results` function gives back the results as a python dictionary holding pandas Series for scalar values and pandas DataFrames for all nodes and flows between them. This way we can make use of the full power of the pandas package available to process the results.

See the pandas documentation to learn how to visualise, read or write or how to access parts of the DataFrame to process them.

The results chapter consists of three parts:

The first step is the processing of the results (*Collecting results*) This is followed by basic examples of the general analysis of the results (*General approach*) and finally the use of functionality already included in solph for providing a quick access to your results (*Easy access*). Especially for larger energy systems the general approach will help you to write your own results processing functions.

2.8.1 Collecting results

Collecting results can be done with the help of the processing module. A solved model is needed:

```
[...]
model.solve(solver=solver)
results = solph.processing.results(model)
```

The scalars and sequences describe nodes (with keys like (node, None)) and flows between nodes (with keys like (node_1, node_2)). You can directly extract the data in the dictionary by using these keys, where "node" is the name of the object you want to address. Processing the results is the prerequisite for the examples in the following sections.

2.8.2 General approach

As stated above, after processing you will get a dictionary with all result data. If you want to access your results directly via labels, you can continue with *Easy access*. For a systematic analysis list comprehensions are the easiest way of filtering and analysing your results.

The keys of the results dictionary are tuples containing two nodes. Since flows have a starting node and an ending node, you get a list of all flows by filtering the results using the following expression:

```
flows = [x for x in results.keys() if x[1] is not None]
```

On the same way you can get a list of all nodes by applying:

```
nodes = [x for x in results.keys() if x[1] is None]
```

Probably you will just get storages as nodes, if you have some in your energy system. Note, that just nodes containing decision variables are listed, e.g. a Source or a Transformer object does not have decision variables. These are in the flows from or to the nodes.

All items within the results dictionary are dictionaries and have two items with 'scalars' and 'sequences' as keys:

```
for flow in flows:
    print(flow)
    print(results[flow]['scalars'])
    print(results[flow]['sequences'])
```

There many options of filtering the flows and nodes as you prefer. The following will give you all flows which are outputs of transformer:

```
flows_from_transformer = [x for x in flows if isinstance(
    x[0], solph.Transformer)]
```

You can filter your flows, if the label of in- or output contains a given string, e.g.:

```
flows_to_elec = [x for x in results.keys() if 'elec' in x[1].label]
```

Getting all labels of the starting node of your investment flows:

```
flows_invest = [x[0].label for x in flows if hasattr(
    results[x]['scalars'], 'invest')]
```

2.8.3 Easy access

The solph package provides some functions which will help you to access your results directly via labels, which is helpful especially for small energy systems. So, if you want to address objects by their label, you can convert the results dictionary such that the keys are changed to strings given by the labels:

```
views.convert_keys_to_strings(results)
print(results[('wind', 'bus_electricity')]['sequences']
```

Another option is to access data belonging to a grouping by the name of the grouping (note also this section on groupings. Given the label of an object, e.g. 'wind' you can access the grouping by its label and use this to extract data from the results dictionary.

```
node_wind = energysystem.groups['wind']
print(results[(node_wind, bus_electricity)])
```

However, in many situations it might be convenient to use the views module to collect information on a specific node. You can request all data related to a specific node by using either the node's variable name or its label:

```
data_wind = solph.views.node(results, 'wind')
```

A function for collecting and printing meta results, i.e. information on the objective function, the problem and the solver, is provided as well:

```
meta_results = solph.processing.meta_results(om)
pp.pprint(meta_results)
```

CHAPTER 3

API Reference

3.1 oemof.solph package

3.1.1 Submodules

3.1.2 oemof.solph.blocks module

Creating sets, variables, constraints and parts of the objective function for the specified groups.

SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de> SPDX-FileCopyrightText: Simon Hilpert SPDX-FileCopyrightText: Cord Kaldemeyer SPDX-FileCopyrightText: Patrik Schönfeldt SPDX-FileCopyrightText: Birgit Schachler SPDX-FileCopyrightText: jnnr SPDX-FileCopyrightText: jmloenneberga

SPDX-License-Identifier: MIT

class oemof.solph.blocks.**Bus**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for all balanced buses.

The following constraints are build:

Bus balance `om.Bus.balance[i, o, t]`

$$\sum_{i \in INPUTS(n)} flow(i, n, t) = \sum_{o \in OUTPUTS(n)} flow(n, o, t),$$
$$\forall n \in \text{BUSES}, \forall t \in \text{TIMESTEPS}.$$

class oemof.solph.blocks.**Flow**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Flow block with definitions for standard flows.

The following variables are created:

negative_gradient : Difference of a flow in consecutive timesteps if flow is reduced indexed by NEGATIVE_GRADIENT_FLOWS, TIMESTEPS.

positive_gradient : Difference of a flow in consecutive timesteps if flow is increased indexed by NEGATIVE_GRADIENT_FLOWS, TIMESTEPS.

The following sets are created: (-> see basic sets at *Model*)

SUMMED_MAX_FLOWS A set of flows with the attribute `summed_max` being not None.

SUMMED_MIN_FLOWS A set of flows with the attribute `summed_min` being not None.

NEGATIVE_GRADIENT_FLOWS A set of flows with the attribute `negative_gradient` being not None.

POSITIVE_GRADIENT_FLOWS A set of flows with the attribute `positive_gradient` being not None

INTEGER_FLOWS A set of flows where the attribute `integer` is True (forces flow to only take integer values)

The following constraints are build:

Flow max sum `om.Flow.summed_max[i, o]`

$$\sum_t flow(i, o, t) \cdot \tau \leq summed\_max(i, o) \cdot nominal\_value(i, o),$$
$$\forall (i, o) \in \text{SUMMED\_MAX\_FLOWS}.$$

Flow min sum `om.Flow.summed_min[i, o]`

$$\sum_t flow(i, o, t) \cdot \tau \geq summed\_min(i, o) \cdot nominal\_value(i, o),$$
$$\forall (i, o) \in \text{SUMMED\_MIN\_FLOWS}.$$

Negative gradient constraint

`om.Flow.negative_gradient_constr[i, o]:`

$$flow(i, o, t-1) - flow(i, o, t) \geq negative\_gradient(i, o, t),$$
$$\forall (i, o) \in \text{NEGATIVE\_GRADIENT\_FLOWS},$$
$$\forall t \in \text{TIMESTEPS}.$$

Positive gradient constraint

`om.Flow.positive_gradient_constr[i, o]:`

$$flow(i, o, t) - flow(i, o, t-1) \geq positive\_gradient(i, o, t),$$
$$\forall (i, o) \in \text{POSITIVE\_GRADIENT\_FLOWS},$$
$$\forall t \in \text{TIMESTEPS}.$$

The following parts of the objective function are created:

If `variable_costs` are set by the user:

$$\sum_{(i,o)} \sum_t flow(i, o, t) \cdot variable\_costs(i, o, t)$$

The expression can be accessed by `om.Flow.variable_costs` and their value after optimization by `om.Flow.variable_costs()` .

class oemof.solph.blocks.**InvestmentFlow**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for all flows with `Investment` being not None.

See *oemof.solph.options.Investment* for all parameters of the *Investment* class.

See *oemof.solph.network.Flow* for all parameters of the *Flow* class.

Variables

All *InvestmentFlow* are indexed by a starting and ending node (i, o), which is omitted in the following for the sake of convenience. The following variables are created:

- $P(t)$

  Actual flow value (created in *oemof.solph.models.BaseModel*).

- P_{invest}

  Value of the investment variable, i.e. equivalent to the nominal value of the flows after optimization.

- b_{invest}

  Binary variable for the status of the investment, if `nonconvex` is *True*.

Constraints

Depending on the attributes of the *InvestmentFlow* and *Flow*, different constraints are created. The following constraint is created for all *InvestmentFlow*:

Upper bound for the flow value

$$P(t) \leq (P_{invest} + P_{exist}) \cdot f_{max}(t)$$

Depeding on the attribute `nonconvex`, the constraints for the bounds of the decision variable P_{invest} are different:

- `nonconvex = False`

$$P_{invest,min} \leq P_{invest} \leq P_{invest,max}$$

- `nonconvex = True`

$$P_{invest,min} \cdot b_{invest} \leq P_{invest}$$
$$P_{invest} \leq P_{invest,max} \cdot b_{invest}$$

For all *InvestmentFlow* (independent of the attribute `nonconvex`), the following additional constraints are created, if the appropriate attribute of the *Flow* (see *oemof.solph.network.Flow*) is set:

- `fix` is not None

  Actual value constraint for investments with fixed flow values

$$P(t) = (P_{invest} + P_{exist}) \cdot f_{fix}(t)$$

- `min != 0`

Lower bound for the flow values

$$P(t) \geq (P_{invest} + P_{exist}) \cdot f_{min}(t)$$

- `summed_max is not None`

  Upper bound for the sum of all flow values (e.g. maximum full load hours)

$$\sum_t P(t) \cdot \tau(t) \leq (P_{invest} + P_{exist}) \cdot f_{sum,min}$$

- `summed_min is not None`

  Lower bound for the sum of all flow values (e.g. minimum full load hours)

$$\sum_t P(t) \cdot \tau(t) \geq (P_{invest} + P_{exist}) \cdot f_{sum,min}$$

Objective function

The part of the objective function added by the *InvestmentFlow* also depends on whether a convex or nonconvex *InvestmentFlow* is selected. The following parts of the objective function are created:

- `nonconvex = False`

$$P_{invest} \cdot c_{invest,var}$$

- `nonconvex = True`

$$P_{invest} \cdot c_{invest,var} + c_{invest,fix} \cdot b_{invest}$$

The total value of all costs of all *InvestmentFlow* can be retrieved calling `om.InvestmentFlow.investment_costs.expr()`.

Table 1: List of Variables (in csv table syntax)

symbol	attribute	explanation
$P(t)$	`flow[n, o, t]`	Actual flow value
P_{invest}	`invest[i, o]`	Invested flow capacity
b_{invest}	`invest_status[i, o]`	Binary status of investment

List of Variables (in rst table syntax):

symbol	attribute	explanation
$P(t)$	`flow[n, o, t]`	Actual flow value
P_{invest}	`invest[i, o]`	Invested flow capacity
b_{invest}	`invest_status[i, o]`	Binary status of investment

Grid table style:

symbol	attribute	explanation
$P(t)$	`flow[n, o, t]`	Actual flow value
P_{invest}	`invest[i, o]`	Invested flow capacity
b_{invest}	`invest_status[i, o]`	Binary status of investment

Table 2: List of Parameters

symbol	attribute	explanation
P_{exist}	`flows[i, o].investment.existing`	Existing flow capacity
$P_{invest,min}$	`flows[i, o].investment.minimum`	Minimum investment capacity
$P_{invest,max}$	`flows[i, o].investment.maximum`	Maximum investment capacity
$c_{invest,var}$	`flows[i, o].investment.ep_costs`	Variable investment costs
$c_{invest,fix}$	`flows[i, o].investment.offset`	Fix investment costs
f_{actual}	`flows[i, o].fix[t]`	Normed fixed value for the flow variable
f_{max}	`flows[i, o].max[t]`	Normed maximum value of the flow
f_{min}	`flows[i, o].min[t]`	Normed minimum value of the flow
$f_{sum,max}$	`flows[i, o].summed_max`	Specific maximum of summed flow values (per installed capacity)
$f_{sum,min}$	`flows[i, o].summed_min`	Specific minimum of summed flow values (per installed capacity)
$\tau(t)$	`timeincrement[t]`	Time step width for each time step

Note: In case of a nonconvex investment flow (`nonconvex=True`), the existing flow capacity P_{exist} needs to be zero. At least, it is not tested yet, whether this works out, or makes any sense at all.

Note: See also *oemof.solph.network.Flow*, *oemof.solph.blocks.Flow* and *oemof.solph.options.Investment*

class oemof.solph.blocks.**NonConvexFlow**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

The following sets are created: (-> see basic sets at *Model* **)**

A set of flows with the attribute `nonconvex` of type *options.NonConvex*.

MIN_FLOWS A subset of set NONCONVEX_FLOWS with the attribute `min` being not None in the first timestep.

ACTIVITYCOSTFLOWS A subset of set NONCONVEX_FLOWS with the attribute `activity_costs` being not None.

STARTUPFLOWS A subset of set NONCONVEX_FLOWS with the attribute `maximum_startups` or `startup_costs` being not None.

MAXSTARTUPFLOWS A subset of set STARTUPFLOWS with the attribute `maximum_startups` being not None.

SHUTDOWNFLOWS A subset of set NONCONVEX_FLOWS with the attribute `maximum_shutdowns` or `shutdown_costs` being not None.

MAXSHUTDOWNFLOWS A subset of set SHUTDOWNFLOWS with the attribute `maximum_shutdowns` being not None.

MINUPTIMEFLOWS A subset of set NONCONVEX_FLOWS with the attribute `minimum_uptime` being not None.

MINDOWNTIMEFLOWS A subset of set NONCONVEX_FLOWS with the attribute `minimum_downtime` being not None.

The following variables are created:

Status variable (binary) `om.NonConvexFlow.status`: Variable indicating if flow is >= 0 indexed by FLOWS

Startup variable (binary) `om.NonConvexFlow.startup`: Variable indicating startup of flow (component) indexed by STARTUPFLOWS

Shutdown variable (binary) `om.NonConvexFlow.shutdown`: Variable indicating shutdown of flow (component) indexed by SHUTDOWNFLOWS

The following constraints are created:

Minimum flow constraint `om.NonConvexFlow.min[i,o,t]`

$$flow(i, o, t) \geq min(i, o, t) \cdot nominal\_value \cdot status(i, o, t),$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall (i, o) \in \text{NONCONVEX\_FLOWS}.$$

Maximum flow constraint `om.NonConvexFlow.max[i,o,t]`

$$flow(i, o, t) \leq max(i, o, t) \cdot nominal\_value \cdot status(i, o, t),$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall (i, o) \in \text{NONCONVEX\_FLOWS}.$$

Startup constraint `om.NonConvexFlow.startup_constr[i,o,t]`

$$startup(i, o, t) \geq status(i, o, t) - status(i, o, t-1)$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall (i, o) \in \text{STARTUPFLOWS}.$$

Maximum startups constraint

`om.NonConvexFlow.max_startup_constr[i,o,t]`

$$\sum_{t \in \text{TIMESTEPS}} startup(i, o, t) \leq N_{start}(i, o) \forall (i, o) \in \text{MAXSTARTUPFLOWS}.$$

Shutdown constraint `om.NonConvexFlow.shutdown_constr[i,o,t]`

$$shutdown(i, o, t) \geq status(i, o, t-1) - status(i, o, t)$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall (i, o) \in \text{SHUTDOWNFLOWS}.$$

Maximum shutdowns constraint

`om.NonConvexFlow.max_startup_constr[i,o,t]`

$$\sum_{t \in \text{TIMESTEPS}} startup(i, o, t) \leq N_{shutdown}(i, o) \forall (i, o) \in \text{MAXSHUTDOWNFLOWS}.$$

Minimum uptime constraint `om.NonConvexFlow.uptime_constr[i,o,t]`

$$(status(i,o,t) - status(i,o,t-1)) \cdot minimum\_uptime(i,o) \leq \sum_{n=0}^{minimum\_uptime-1} status(i,o,t+n)$$
$$\forall t \in \text{TIMESTEPS}|$$
$$t \neq \{0..minimum\_uptime\} \cup \{t\_max - minimum\_uptime..t\_max\},$$
$$\forall (i,o) \in \text{MINUPTIMEFLOWS}.$$

$$status(i,o,t) = initial\_status(i,o)$$
$$\forall t \in \text{TIMESTEPS}|$$
$$t = \{0..minimum\_uptime\} \cup \{t\_max - minimum\_uptime..t\_max\},$$
$$\forall (i,o) \in \text{MINUPTIMEFLOWS}.$$

Minimum downtime constraint `om.NonConvexFlow.downtime_constr[i,o,t]`

$$(status(i,o,t-1) - status(i,o,t)) \cdot minimum\_downtime(i,o) \leq minimum\_downtime(i,o) - \sum_{n=0}^{minimum\_downtime-1} status(i,o,t+n)$$
$$\forall t \in \text{TIMESTEPS}|$$
$$t \neq \{0..minimum\_downtime\} \cup \{t\_max - minimum\_downtime..t\_max\},$$
$$\forall (i,o) \in \text{MINDOWNTIMEFLOWS}.$$

$$status(i,o,t) = initial\_status(i,o)$$
$$\forall t \in \text{TIMESTEPS}|$$
$$t = \{0..minimum\_downtime\} \cup \{t\_max - minimum\_downtime..t\_max\},$$
$$\forall (i,o) \in \text{MINDOWNTIMEFLOWS}.$$

The following parts of the objective function are created:

If `nonconvex.startup_costs` is set by the user:

$$\sum_{i,o \in STARTUPFLOWS} \sum_{t} startup(i,o,t) \cdot startup\_costs(i,o)$$

If `nonconvex.shutdown_costs` is set by the user:

$$\sum_{i,o \in SHUTDOWNFLOWS} \sum_{t} shutdown(i,o,t) \cdot shutdown\_costs(i,o)$$

If `nonconvex.activity_costs` is set by the user:

$$\sum_{i,o \in ACTIVITYCOSTFLOWS} \sum_{t} status(i,o,t) \cdot activity\_costs(i,o)$$

class oemof.solph.blocks.**Transformer**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for the linear relation of nodes with type *Transformer*

The following sets are created: (-> see basic sets at *Model*)

TRANSFORMERS A set with all *Transformer* objects.

The following constraints are created:

Linear relation `om.Transformer.relation[i,o,t]`

$$flow(i, n, t)/conversion\_factor(n, i, t) = flow(n, o, t)/conversion\_factor(n, o, t),$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall n \in \text{TRANSFORMERS},$$
$$\forall i \in \text{INPUTS(n)},$$
$$\forall o \in \text{OUTPUTS(n)}.$$

3.1.3 oemof.solph.components module

This module is designed to hold components with their classes and associated individual constraints (blocks) and groupings. Therefore this module holds the class definition and the block directly located by each other.

SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de> SPDX-FileCopyrightText: Simon Hilpert SPDX-FileCopyrightText: Cord Kaldemeyer SPDX-FileCopyrightText: Patrik Schönfeldt SPDX-FileCopyrightText: FranziPl SPDX-FileCopyrightText: jnnr SPDX-FileCopyrightText: Stephan Günther SPDX-FileCopyrightText: FabianTU SPDX-FileCopyrightText: Johannes Röder

SPDX-License-Identifier: MIT

class oemof.solph.components.**ExtractionTurbineCHP**(*conversion_factor_full_condensation*, **args*, ***kwargs*)

Bases: *oemof.solph.network.Transformer*

A CHP with an extraction turbine in a linear model. For more options see the *GenericCHP* class.

One main output flow has to be defined and is tapped by the remaining flow. The conversion factors have to be defined for the maximum tapped flow (full CHP mode) and for no tapped flow (full condensing mode). Even though it is possible to limit the variability of the tapped flow, so that the full condensing mode will never be reached.

Parameters

- **conversion_factors** (*dict*) – Dictionary containing conversion factors for conversion of inflow to specified outflow. Keys are output bus objects. The dictionary values can either be a scalar or a sequence with length of time horizon for simulation.
- **conversion_factor_full_condensation** (*dict*) – The efficiency of the main flow if there is no tapped flow. Only one key is allowed. Use one of the keys of the conversion factors. The key indicates the main flow. The other output flow is the tapped flow.

Note:

The following sets, variables, constraints and objective parts are created

- *ExtractionTurbineCHPBlock*

Examples

```
>>> from oemof import solph
>>> bel = solph.Bus(label='electricityBus')
>>> bth = solph.Bus(label='heatBus')
>>> bgas = solph.Bus(label='commodityBus')
>>> et_chp = solph.components.ExtractionTurbineCHP(
...     label='variable_chp_gas',
...     inputs={bgas: solph.Flow(nominal_value=10e10)},
...     outputs={bel: solph.Flow(), bth: solph.Flow()},
...     conversion_factors={bel: 0.3, bth: 0.5},
...     conversion_factor_full_condensation={bel: 0.5})
```

constraint_group()

class oemof.solph.components.**ExtractionTurbineCHPBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for the linear relation of nodes with type *ExtractionTurbineCHP*

The following two constraints are created:

$$(1) \dot{H}_{Fuel}(t) = \frac{P_{el}(t) + \dot{Q}_{th}(t) \cdot \beta(t)}{\eta_{el,woExtr}(t)}$$

$$(2) P_{el}(t) \geq \dot{Q}_{th}(t) \cdot C_b = \dot{Q}_{th}(t) \cdot \frac{\eta_{el,maxExtr}(t)}{\eta_{th,maxExtr}(t)}$$

where β is defined as:

$$\beta(t) = \frac{\eta_{el,woExtr}(t) - \eta_{el,maxExtr}(t)}{\eta_{th,maxExtr}(t)}$$

where the first equation is the result of the relation between the input flow and the two output flows, the second equation stems from how the two output flows relate to each other, and the symbols used are defined as follows (with Variables (V) and Parameters (P)):

symbol	attribute	type	explanation
$\dot{H}_{Fuel}$	flow[i, n, t]	V	fuel input flow
P_{el}	flow[n, main_output, t]	V	electric power
$\dot{Q}_{th}$	flow[n, tapped_output, t]	V	thermal output
β	main_flow_loss_index[n, t]	P	power loss index
$\eta_{el,woExtr}$	conversion_factor_full_condensation[n, t]	P	electric efficiency without heat extraction
$\eta_{el,maxExtr}$	conversion_factors[main_output][n, t]	P	electric efficiency with max heat extraction
$\eta_{th,maxExtr}$	conversion_factors[tapped_output][n, t]	P	thermal efficiency with maximal heat extraction

CONSTRAINT_GROUP = True

class oemof.solph.components.**GenericCHP**(**args*, ***kwargs*)

Bases: oemof.network.network.Transformer

Component *GenericCHP* to model combined heat and power plants.

Can be used to model (combined cycle) extraction or back-pressure turbines and used a mixed-integer linear formulation. Thus, it induces more computational effort than the *ExtractionTurbineCHP* for the benefit of higher accuracy.

The full set of equations is described in: Mollenhauer, E., Christidis, A. & Tsatsaronis, G. Evaluation of an energy- and exergy-based generic modeling approach of combined heat and power plants Int J Energy Environ Eng (2016) 7: 167. https://doi.org/10.1007/s40095-016-0204-6

For a general understanding of (MI)LP CHP representation, see: Fabricio I. Salgado, P. Short - Term Operation Planning on Cogeneration Systems: A Survey Electric Power Systems Research (2007) Electric Power Systems Research Volume 78, Issue 5, May 2008, Pages 835-848 https://doi.org/10.1016/j.epsr.2007.06.001

Note: An adaption for the flow parameter *H_L_FG_share_max* has been made to set the flue gas losses at maximum heat extraction *H_L_FG_max* as share of the fuel flow *H_F* e.g. for combined cycle extraction turbines. The flow parameter *H_L_FG_share_min* can be used to set the flue gas losses at minimum heat extraction *H_L_FG_min* as share of the fuel flow *H_F* e.g. for motoric CHPs. The boolean component parameter *back_pressure* can be set to model back-pressure characteristics.

Also have a look at the examples on how to use it.

Parameters

- **fuel_input** (*dict*) – Dictionary with key-value-pair of *oemof.Bus* and *oemof.Flow* object for the fuel input.
- **electrical_output** (*dict*) – Dictionary with key-value-pair of *oemof.Bus* and *oemof.Flow* object for the electrical output. Related parameters like *P_max_woDH* are passed as attributes of the *oemof.Flow* object.
- **heat_output** (*dict*) – Dictionary with key-value-pair of *oemof.Bus* and *oemof.Flow* object for the heat output. Related parameters like *Q_CW_min* are passed as attributes of the *oemof.Flow* object.
- **Beta** (*list of numerical values*) – Beta values in same dimension as all other parameters (length of optimization period).
- **back_pressure** (*boolean*) – Flag to use back-pressure characteristics. Set to *True* and *Q_CW_min* to zero for back-pressure turbines. See paper above for more information.

Note:

The following sets, variables, constraints and objective parts are created

- *GenericCHPBlock*

Examples

```
>>> from oemof import solph
>>> bel = solph.Bus(label='electricityBus')
>>> bth = solph.Bus(label='heatBus')
>>> bgas = solph.Bus(label='commodityBus')
>>> ccet = solph.components.GenericCHP(
...     label='combined_cycle_extraction_turbine',
...     fuel_input={bgas: solph.Flow(
```

(continues on next page)

(continued from previous page)

```
...         H_L_FG_share_max=[0.183])},
...     electrical_output={bel: solph.Flow(
...         P_max_woDH=[155.946],
...         P_min_woDH=[68.787],
...         Eta_el_max_woDH=[0.525],
...         Eta_el_min_woDH=[0.444])},
...     heat_output={bth: solph.Flow(
...         Q_CW_min=[10.552])},
...     Beta=[0.122], back_pressure=False)
>>> type(ccet)
<class 'oemof.solph.components.GenericCHP'>
```

alphas
Compute or return the _alphas attribute.

constraint_group()

class oemof.solph.components.**GenericCHPBlock**(**args*, ***kwargs*)
Bases: pyomo.core.base.block.SimpleBlock

Block for the relation of the n nodes with type class:.*GenericCHP*.

The following constraints are created:

(1) $\dot{H}_F(t) = fuel\ input$

(2) $\dot{Q}(t) = heat\ output$

(3) $P_{el}(t) = power\ output$

(4) $\dot{H}_F(t) = \alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot P_{el,woDH}(t)$

(5) $\dot{H}_F(t) = \alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot (P_{el}(t) + \beta \cdot \dot{Q}(t))$

(6) $\dot{H}_F(t) \leq Y(t) \cdot \frac{P_{el,max,woDH}(t)}{\eta_{el,max,woDH}(t)}$

(7) $\dot{H}_F(t) \geq Y(t) \cdot \frac{P_{el,min,woDH}(t)}{\eta_{el,min,woDH}(t)}$

(8) $\dot{H}_{L,FG,max}(t) = \dot{H}_F(t) \cdot \dot{H}_{L,FG,sharemax}(t)$

(9) $\dot{H}_{L,FG,min}(t) = \dot{H}_F(t) \cdot \dot{H}_{L,FG,sharemin}(t)$

(10) $P_{el}(t) + \dot{Q}(t) + \dot{H}_{L,FG,max}(t) + \dot{Q}_{CW,min}(t) \cdot Y(t) = / \leq \dot{H}_F(t)$

where $= / \leq$ depends on the CHP being back pressure or not.

The coefficients α_0 and α_1 can be determined given the efficiencies maximal/minimal load:

$$\eta_{el,max,woDH}(t) = \frac{P_{el,max,woDH}(t)}{\alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot P_{el,max,woDH}(t)}$$

$$\eta_{el,min,woDH}(t) = \frac{P_{el,min,woDH}(t)}{\alpha_0(t) \cdot Y(t) + \alpha_1(t) \cdot P_{el,min,woDH}(t)}$$

For the attribute $\dot{H}_{L,FG,min}$ being not None, e.g. for a motoric CHP, **the following is created:**

Constraint:

(11) $P_{el}(t) + \dot{Q}(t) + \dot{H}_{L,FG,min}(t) + \dot{Q}_{CW,min}(t) \cdot Y(t) \geq \dot{H}_F(t)$

The symbols used are defined as follows (with Variables (V) and Parameters (P)):

math. symbol	attribute	type	explanation
$\dot{H}_F$	`H_F[n,t]`	V	input of enthalpy through fuel input
P_{el}	`P[n,t]`	V	provided electric power
$P_{el,woDH}$	`P_woDH[n,t]`	V	electric power without district heating
$P_{el,min,woDH}$	`P_min_woDH[n,t]`	P	min. electric power without district heating
$P_{el,max,woDH}$	`P_max_woDH[n,t]`	P	max. electric power without district heating
$\dot{Q}$	`Q[n,t]`	V	provided heat
$\dot{Q}_{CW,min}$	`Q_CW_min[n,t]`	P	minimal therm. condenser load to cooling water
$\dot{H}_{L,FG,min}$	`H_L_FG_min[n,t]`	V	flue gas enthalpy loss at min heat extraction
$\dot{H}_{L,FG,max}$	`H_L_FG_max[n,t]`	V	flue gas enthalpy loss at max heat extraction
$\dot{H}_{L,FG,sharemin}$	`H_L_FG_share_min[n,t]`	P	share of flue gas loss at min heat extraction
$\dot{H}_{L,FG,sharemax}$	`H_L_FG_share_max[n,t]`	P	share of flue gas loss at max heat extraction
Y	`Y[n,t]`	V	status variable on/off
α_0	`n.alphas[0][n,t]`	P	coefficient describing efficiency
α_1	`n.alphas[1][n,t]`	P	coefficient describing efficiency
β	`Beta[n,t]`	P	power loss index
$\eta_{el,min,woDH}$	`Eta_el_min_woDH[n,t]`	P	el. eff. at min. fuel flow w/o distr. heating
$\eta_{el,max,woDH}$	`Eta_el_max_woDH[n,t]`	P	el. eff. at max. fuel flow w/o distr. heating

CONSTRAINT_GROUP = True

class oemof.solph.components.**GenericInvestmentStorageBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for all storages with `Investment` being not None. See *oemof.solph.options.Investment* for all parameters of the Investment class.

Variables

All Storages are indexed by n, which is omitted in the following for the sake of convenience. The following variables are created as attributes of `om.InvestmentStorage`:

- $P_i(t)$

  Inflow of the storage (created in *oemof.solph.models.BaseModel*).

- $P_o(t)$

  Outflow of the storage (created in *oemof.solph.models.BaseModel*).

- $E(t)$

  Current storage content (Absolute level of stored energy).

- E_{invest}

  Invested (nominal) capacity of the storage.

- $E(-1)$

  Initial storage content (before timestep 0).

- b_{invest}

  Binary variable for the status of the investment, if `nonconvex` is *True*.

Constraints

The following constraints are created for all investment storages:

Storage balance (Same as for *GenericStorageBlock*)

$$
\begin{aligned}
E(t) =& E(t-1) \cdot (1 - \beta(t))^{\tau(t)/(t_u)} \\
& - \gamma(t) \cdot (E_{exist} + E_{invest}) \cdot \tau(t)/(t_u) \\
& - \delta(t) \cdot \tau(t)/(t_u) \\
& - \frac{P_o(t)}{\eta_o(t)} \cdot \tau(t) + P_i(t) \cdot \eta_i(t) \cdot \tau(t)
\end{aligned}
$$

Depending on the attribute `nonconvex`, the constraints for the bounds of the decision variable E_{invest} are different:

- `nonconvex = False`

$$E_{invest,min} \leq E_{invest} \leq E_{invest,max}$$

- `nonconvex = True`

$$
\begin{aligned}
E_{invest,min} \cdot b_{invest} &\leq E_{invest} \\
E_{invest} &\leq E_{invest,max} \cdot b_{invest}
\end{aligned}
$$

The following constraints are created depending on the attributes of the *components.GenericStorage*:

- `initial_storage_level is None`

  Constraint for a variable initial storage content:

$$E(-1) \leq E_{invest} + E_{exist}$$

- `initial_storage_level is not None`

  An initial value for the storage content is given:

$$E(-1) = (E_{invest} + E_{exist}) \cdot c(-1)$$

- `balanced=True`

  The energy content of storage of the first and the last timestep are set equal:

$$E(-1) = E(t_{last})$$

- `invest_relation_input_capacity is not None`

  Connect the invest variables of the storage and the input flow:

$$P_{i,invest} + P_{i,exist} = (E_{invest} + E_{exist}) \cdot r_{cap,in}$$

- `invest_relation_output_capacity is not None`

  Connect the invest variables of the storage and the output flow:

$$P_{o,invest} + P_{o,exist} = (E_{invest} + E_{exist}) \cdot r_{cap,out}$$

- `invest_relation_input_output is not None`

Connect the invest variables of the input and the output flow:

$$P_{i,invest} + P_{i,exist} = (P_{o,invest} + P_{o,exist}) \cdot r_{in,out}$$

- `max_storage_level`

  Rule for upper bound constraint for the storage content:

$$E(t) \leq E_{invest} \cdot c_{max}(t)$$

- `min_storage_level`

  Rule for lower bound constraint for the storage content:

$$E(t) \geq E_{invest} \cdot c_{min}(t)$$

Objective function

The part of the objective function added by the investment storages also depends on whether a convex or non-convex investment option is selected. The following parts of the objective function are created:

- `nonconvex = False`

$$E_{invest} \cdot c_{invest,var}$$

- `nonconvex = True`

$$E_{invest} \cdot c_{invest,var} + c_{invest,fix} \cdot b_{invest}$$

The total value of all investment costs of all *InvestmentStorages* can be retrieved calling `om.GenericInvestmentStorageBlock.investment_costs.expr()`.

Table 3: List of Variables

symbol	attribute	explanation
$P_i(t)$	`flow[i[n], n, t]`	Inflow of the storage
$P_o(t)$	`flow[n, o[n], t]`	Outlfow of the storage
$E(t)$	`storage_content[n, t]`	Current storage content (current absolute stored energy)
E_{invest}	`invest[n, t]`	Invested (nominal) capacity of the storage
$E(-1)$	`init_cap[n]`	Initial storage capacity (before timestep 0)
b_{invest}	`invest_status[i, o]`	Binary variable for the status of investment
$P_{i,invest}$	`InvestmentFlow.invest[i[n], n]`	Invested (nominal) inflow (Investmentflow)
$P_{o,invest}$	`InvestmentFlow.invest[n, o[n]]`	Invested (nominal) outflow (Investmentflow)

Table 4: List of Parameters

symbol	attribute	explanation
E_{exist}	`flows[i, o].investment.existing`	Existing storage capacity
$E_{invest,min}$	`flows[i, o].investment.minimum`	Minimum investment value
$E_{invest,max}$	`flows[i, o].investment.maximum`	Maximum investment value
$P_{i,exist}$	`flows[i[n], n].investment.existing`	Existing inflow capacity
$P_{o,exist}$	`flows[n, o[n]].investment.existing`	Existing outlfow capacity
$c_{invest,var}$	`flows[i, o].investment.ep_costs`	Variable investment costs
$c_{invest,fix}$	`flows[i, o].investment.offset`	Fix investment costs
$r_{cap,in}$	`invest_relation_input_capacity`	Relation of storage capacity and nominal inflow
$r_{cap,out}$	`invest_relation_output_capacity`	Relation of storage capacity and nominal outflow
$r_{in,out}$	`invest_relation_input_output`	Relation of nominal in- and outflow
$\beta(t)$	`loss_rate[t]`	Fraction of lost energy as share of $E(t)$ per time unit
$\gamma(t)$	`fixed_losses_relative[t]`	Fixed loss of energy relative to $E_{invest} + E_{exist}$ per time unit
$\delta(t)$	`fixed_losses_absolute[t]`	Absolute fixed loss of energy per time unit
$\eta_i(t)$	`inflow_conversion_factor[t]`	Conversion factor (i.e. efficiency) when storing energy
$\eta_o(t)$	`outflow_conversion_factor[t]`	Conversion factor when (i.e. efficiency) taking stored energy
$c(-1)$	`initial_storage_level`	Initial relativ storage content (before timestep 0)
c_{max}	`flows[i, o].max[t]`	Normed maximum value of storage content
c_{min}	`flows[i, o].min[t]`	Normed minimum value of storage content
$\tau(t)$		Duration of time step
t_u		Time unit of losses $\beta(t)$, $\gamma(t)$, $\delta(t)$ and timeincrement $\tau(t)$

CONSTRAINT_GROUP = True

class oemof.solph.components.**GenericStorage**(**args*, *max_storage_level=1*, *min_storage_level=0*, ***kwargs*)

Bases: `oemof.network.network.Transformer`

Component *GenericStorage* to model with basic characteristics of storages.

Parameters

- **nominal_storage_capacity** (numeric, E_{nom}) – Absolute nominal capacity of the storage
- **invest_relation_input_capacity** (numeric or None, $r_{cap,in}$) – Ratio between the investment

variable of the input Flow and the investment variable of the storage: $\dot{E}_{in,invest} = E_{invest} \cdot r_{cap,in}$

- **invest_relation_output_capacity** (numeric or None, $r_{cap,out}$) – Ratio between the investment variable of the output Flow and the investment variable of the storage: $\dot{E}_{out,invest} = E_{invest} \cdot r_{cap,out}$
- **invest_relation_input_output** (numeric or None, $r_{in,out}$) – Ratio between the investment variable of the output Flow and the investment variable of the input flow. This ratio used to fix the flow investments to each other. Values < 1 set the input flow lower than the output and > 1 will set the input flow higher than the output flow. If None no relation will be set: $\dot{E}_{in,invest} = \dot{E}_{out,invest} \cdot r_{in,out}$
- **initial_storage_level** (numeric, $c(-1)$) – The relative storage content in the timestep before the first time step of optimization (between 0 and 1).
- **balanced** (*boolean*) – Couple storage level of first and last time step. (Total inflow and total outflow are balanced.)
- **loss_rate** (*numeric (iterable or scalar)*) – The relative loss of the storage content per time unit.
- **fixed_losses_relative** (numeric (iterable or scalar), $\gamma(t)$) – Losses independent of state of charge between two consecutive timesteps relative to nominal storage capacity.
- **fixed_losses_absolute** (numeric (iterable or scalar), $\delta(t)$) – Losses independent of state of charge and independent of nominal storage capacity between two consecutive timesteps.
- **inflow_conversion_factor** (numeric (iterable or scalar), $\eta_i(t)$) – The relative conversion factor, i.e. efficiency associated with the inflow of the storage.
- **outflow_conversion_factor** (numeric (iterable or scalar), $\eta_o(t)$) – see: inflow_conversion_factor
- **min_storage_level** (numeric (iterable or scalar), $c_{min}(t)$) – The normed minimum storage content as fraction of the nominal storage capacity (between 0 and 1). To set different values in every time step use a sequence.
- **max_storage_level** (numeric (iterable or scalar), $c_{max}(t)$) – see: min_storage_level
- **investment** (*oemof.solph.options.Investment* object) – Object indicating if a nominal_value of the flow is determined by the optimization problem. Note: This will refer all attributes to an investment variable instead of to the nominal_storage_capacity. The nominal_storage_capacity should not be set (or set to None) if an investment object is used.

Note:

The following sets, variables, constraints and objective parts are created

- *GenericStorageBlock* (if no Investment object present)
- *GenericInvestmentStorageBlock* (if Investment object present)

Examples

Basic usage examples of the GenericStorage with a random selection of attributes. See the Flow class for all Flow attributes.

```
>>> from oemof import solph
```

```
>>> my_bus = solph.Bus('my_bus')
```

```
>>> my_storage = solph.components.GenericStorage(
...     label='storage',
...     nominal_storage_capacity=1000,
...     inputs={my_bus: solph.Flow(nominal_value=200, variable_costs=10)},
...     outputs={my_bus: solph.Flow(nominal_value=200)},
...     loss_rate=0.01,
...     initial_storage_level=0,
...     max_storage_level = 0.9,
...     inflow_conversion_factor=0.9,
...     outflow_conversion_factor=0.93)
```

```
>>> my_investment_storage = solph.components.GenericStorage(
...     label='storage',
...     investment=solph.Investment(ep_costs=50),
...     inputs={my_bus: solph.Flow()},
...     outputs={my_bus: solph.Flow()},
...     loss_rate=0.02,
...     initial_storage_level=None,
...     invest_relation_input_capacity=1/6,
...     invest_relation_output_capacity=1/6,
...     inflow_conversion_factor=1,
...     outflow_conversion_factor=0.8)
```

constraint_group()

class oemof.solph.components.**GenericStorageBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Storage without an *Investment* object.

The following sets are created: (-> see basic sets at *Model*)

STORAGES

A set with all `Storage` objects, which do not have an attr:*investment* of type *Investment*.

STORAGES_BALANCED A set of all `Storage` objects, with 'balanced' attribute set to True.

STORAGES_WITH_INVEST_FLOW_REL A set with all `Storage` objects with two investment flows coupled with the 'invest_relation_input_output' attribute.

The following variables are created:

storage_content Storage content for every storage and timestep. The value for the storage content at the beginning is set by the parameter *initial_storage_level* or not set if *initial_storage_level* is None. The variable of storage s and timestep t can be accessed by: *om.Storage.storage_content[s, t]*

The following constraints are created:

Set storage_content of last time step to one at t=0 if `balanced == True`

$$E(t_{last}) = E(-1)$$

Storage balance `om.Storage.balance[n, t]`

$$
\begin{aligned}
E(t) = & E(t-1) \cdot (1 - \beta(t))^{\tau(t)/(t_u)} \\
& - \gamma(t) \cdot E_{nom} \cdot \tau(t)/(t_u) \\
& - \delta(t) \cdot \tau(t)/(t_u) \\
& - \frac{\dot{E}_o(t)}{\eta_o(t)} \cdot \tau(t) + \dot{E}_i(t) \cdot \eta_i(t) \cdot \tau(t)
\end{aligned}
$$

Connect the invest variables of the input and the output flow.

$$
\begin{aligned}
InvestmentFlow.invest(source(n), n) + existing = \\
(InvestmentFlow.invest(n, target(n)) + existing) * \\
invest\_relation\_input\_output(n) \\
\forall n \in \text{INVEST\_REL\_IN\_OUT}
\end{aligned}
$$

symbol	explanation	attribute
$E(t)$	energy currently stored	`storage_content`
E_{nom}	nominal capacity of the energy storage	`nominal_storage_capacity`
$c(-1)$	state before initial time step	`initial_storage_level`
$c_{min}(t)$	minimum allowed storage	`min_storage_level[t]`
$c_{max}(t)$	maximum allowed storage	`max_storage_level[t]`
$\beta(t)$	fraction of lost energy as share of $E(t)$ per time unit	`loss_rate[t]`
$\gamma(t)$	fixed loss of energy relative to E_{nom} per time unit	`fixed_losses_relative[t]`
$\delta(t)$	absolute fixed loss of energy per time unit	`fixed_losses_absolute[t]`
$\dot{E}_i(t)$	energy flowing in	`inputs`
$\dot{E}_o(t)$	energy flowing out	`outputs`
$\eta_i(t)$	conversion factor (i.e. efficiency) when storing energy	`inflow_conversion_factor[t]`
$\eta_o(t)$	conversion factor when (i.e. efficiency) taking stored energy	`outflow_conversion_factor[t]`
$\tau(t)$	duration of time step	
t_u	time unit of losses $\beta(t)$, $\gamma(t)$ $\delta(t)$ and timeincrement $\tau(t)$	

The following parts of the objective function are created:

Nothing added to the objective function.

`CONSTRAINT_GROUP = True`

class `oemof.solph.components.`**`OffsetTransformer`**(**args*, ***kwargs*)

Bases: `oemof.network.network.Transformer`

An object with one input and one output.

Parameters **coefficients** (*tuple*) – Tuple containing the first two polynomial coefficients i.e. the y-intersection and slope of a linear equation. The tuple values can either be a scalar or a sequence with length of time horizon for simulation.

Notes

The sets, variables, constraints and objective parts are created

- *`OffsetTransformerBlock`*

Examples

```
>>> from oemof import solph
```

```
>>> bel = solph.Bus(label='bel')
>>> bth = solph.Bus(label='bth')
```

```
>>> ostf = solph.components.OffsetTransformer(
...     label='ostf',
...     inputs={bel: solph.Flow(
...         nominal_value=60, min=0.5, max=1.0,
...         nonconvex=solph.NonConvex())},
...     outputs={bth: solph.Flow()},
...     coefficients=(20, 0.5))
```

```
>>> type(ostf)
<class 'oemof.solph.components.OffsetTransformer'>
```

constraint_group()

class oemof.solph.components.**OffsetTransformerBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for the relation of nodes with type *OffsetTransformer*

The following constraints are created:

$$P_{out}(t) = C_1(t) \cdot P_{in}(t) + C_0(t) \cdot Y(t)$$

Table 5: Variables (V) and Parameters (P)

symbol	attribute	type	explanation
$P_{out}(t)$	`flow[n, o, t]`	V	Power of output
$P_{in}(t)$	`flow[i, n, t]`	V	Power of input
$Y(t)$	`status[i, n, t]`	V	binary status variable of nonconvex input flow
$C_1(t)$	`coefficients[1][n, t]`	P	linear coefficient 1 (slope)
$C_0(t)$	`coefficients[0][n, t]`	P	linear coefficient 0 (y-intersection)

CONSTRAINT_GROUP = True

3.1.4 oemof.solph.constraints module

Additional constraints to be used in an oemof energy model.

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

SPDX-License-Identifier: MIT

oemof.solph.constraints.**additional_investment_flow_limit**(*model*, *keyword*, *limit=None*)

Global limit for investment flows weighted by an attribute keyword.

This constraint is only valid for Flows not for components such as an investment storage.

The attribute named by keyword has to be added to every Investment attribute of the flow you want to take into account. Total value of keyword attributes after optimization can be retrieved calling the `oemof.solph.Model.invest_limit_${keyword}()`.

Parameters

- **model** (*oemof.solph.Model*) – Model to which constraints are added.
- **keyword** (*attribute to consider*) – All flows with Investment attribute containing the keyword will be used.
- **limit** (*numeric*) – Global limit of keyword attribute for the energy system.

$$\sum_{i \in IF} P_i \cdot w_i \leq limit$$

With *IF* being the set of InvestmentFlows considered for the integral limit.

The symbols used are defined as follows (with Variables (V) and Parameters (P)):

symbol	attribute	type	explanation
P_i	`InvestmentFlow.invest[i, o]`	V	installed capacity of investment flow
w_i	`keyword`	P	weight given to investment flow named according to *keyword*
$limit$	`limit`	P	global limit given by keyword *limit*

Note: The Investment attribute of the considered (Investment-)flows requires an attribute named like keyword!

Examples

```
>>> import pandas as pd
>>> from oemof import solph
>>> date_time_index = pd.date_range('1/1/2020', periods=5, freq='H')
>>> es = solph.EnergySystem(timeindex=date_time_index)
>>> bus = solph.Bus(label='bus_1')
>>> sink = solph.Sink(label="sink", inputs={bus:
...     solph.Flow(nominal_value=10, fix=[10, 20, 30, 40, 50])})
>>> src1 = solph.Source(label='source_0', outputs={bus: solph.Flow(
...     investment=solph.Investment(ep_costs=50, space=4))})
>>> src2 = solph.Source(label='source_1', outputs={bus: solph.Flow(
...     investment=solph.Investment(ep_costs=100, space=1))})
>>> es.add(bus, sink, src1, src2)
>>> model = solph.Model(es)
>>> model = solph.constraints.additional_investment_flow_limit(
...     model, "space", limit=1500)
>>> a = model.solve(solver="cbc")
>>> int(round(model.invest_limit_space()))
1500
```

oemof.solph.constraints.**emission_limit**(*om*, *flows=None*, *limit=None*)
Short handle for generic_integral_limit() with keyword="emission_factor".

Note: Flow objects required an attribute "emission_factor"!

oemof.solph.constraints.**equate_variables**(*model*, *var1*, *var2*, *factor1=1*, *name=None*)
Adds a constraint to the given model that set two variables to equal adaptable by a factor.

The following constraints are build:

$$var1 \cdot factor1 = var2$$

Parameters

- **var1** (*pyomo.environ.Var*) – First variable, to be set to equal with Var2 and multiplied with factor1.
- **var2** (*pyomo.environ.Var*) – Second variable, to be set equal to (Var1 * factor1).
- **factor1** (*float*) – Factor to define the proportion between the variables.
- **name** (*str*) – Optional name for the equation e.g. in the LP file. By default the name is: equate + string representation of var1 and var2.
- **model** (*oemof.solph.Model*) – Model to which the constraint is added.

Examples

The following example shows how to define a transmission line in the investment mode by connecting both investment variables. Note that the equivalent periodical costs (epc) of the line are 40. You could also add them to one line and set them to 0 for the other line.

```
>>> import pandas as pd
>>> from oemof import solph
>>> date_time_index = pd.date_range('1/1/2012', periods=5, freq='H')
>>> energysystem = solph.EnergySystem(timeindex=date_time_index)
>>> bel1 = solph.Bus(label='electricity1')
>>> bel2 = solph.Bus(label='electricity2')
>>> energysystem.add(bel1, bel2)
>>> energysystem.add(solph.Transformer(
...     label='powerline_1_2',
...     inputs={bel1: solph.Flow()},
...     outputs={bel2: solph.Flow(
...         investment=solph.Investment(ep_costs=20))}))
>>> energysystem.add(solph.Transformer(
...     label='powerline_2_1',
...     inputs={bel2: solph.Flow()},
...    outputs={bel1: solph.Flow(
...        investment=solph.Investment(ep_costs=20))}))
>>> om = solph.Model(energysystem)
>>> line12 = energysystem.groups['powerline_1_2']
>>> line21 = energysystem.groups['powerline_2_1']
>>> solph.constraints.equate_variables(
...     om,
...     om.InvestmentFlow.invest[line12, bel2],
...     om.InvestmentFlow.invest[line21, bel1])
```

oemof.solph.constraints.**generic_integral_limit**(*om*, *keyword*, *flows=None*, *limit=None*)

Set a global limit for flows weighted by attribute called keyword. The attribute named by keyword has to be added to every flow you want to take into account.

Total value of keyword attributes after optimization can be retrieved calling the `om.oemof.solph.Model.integral_limit_${keyword}()`.

Parameters

- **om** (*oemof.solph.Model*) – Model to which constraints are added.
- **flows** (*dict*) – Dictionary holding the flows that should be considered in constraint. Keys are (source, target) objects of the Flow. If no dictionary is given all flows containing the keyword attribute will be used.
- **keyword** (*attribute to consider*)
- **limit** (*numeric*) – Absolute limit of keyword attribute for the energy system.

Note: Flow objects required an attribute named like keyword!

Constraint:

$$\sum_{i \in F_E} \sum_{t \in T} P_i(t) \cdot w_i(t) \cdot \tau(t) \leq M$$

With *F_I* being the set of flows considered for the integral limit and *T* being the set of time steps.

The symbols used are defined as follows (with Variables (V) and Parameters (P)):

math. symbol	type	explanation
$P_n(t)$	V	power flow n at time step t
$w_N(t)$	P	weight given to Flow named according to *keyword*
$\tau(t)$	P	width of time step t
L	P	global limit given by keyword *limit*

Examples

```
>>> import pandas as pd
>>> from oemof import solph
>>> date_time_index = pd.date_range('1/1/2012', periods=5, freq='H')
>>> energysystem = solph.EnergySystem(timeindex=date_time_index)
>>> bel = solph.Bus(label='electricityBus')
>>> flow1 = solph.Flow(nominal_value=100, my_factor=0.8)
>>> flow2 = solph.Flow(nominal_value=50)
>>> src1 = solph.Source(label='source1', outputs={bel: flow1})
>>> src2 = solph.Source(label='source2', outputs={bel: flow2})
>>> energysystem.add(bel, src1, src2)
>>> model = solph.Model(energysystem)
>>> flow_with_keyword = {(src1, bel): flow1, }
>>> model = solph.constraints.generic_integral_limit(
...     model, "my_factor", flow_with_keyword, limit=777)
```

oemof.solph.constraints.**investment_limit**(*model*, *limit=None*)

Set an absolute limit for the total investment costs of an investment optimization problem:

$$\sum_{investment\_costs} \leq limit$$

Parameters

- **model** (*oemof.solph.Model*) – Model to which the constraint is added
- **limit** (*float*) – Absolute limit of the investment (i.e. RHS of constraint)

oemof.solph.constraints.**limit_active_flow_count**(*model*, *constraint_name*, *flows*, *lower_limit=0*, *upper_limit=None*)

Set limits (lower and/or upper) for the number of concurrently active NonConvex flows. The flows are given as a list.

Total actual counts after optimization can be retrieved calling the `om.oemof.solph.Model.$(constraint_name)_count()`.

Parameters

- **model** (*oemof.solph.Model*) – Model to which constraints are added
- **constraint_name** (*string*) – name for the constraint
- **flows** (*list of flows*) – flows (have to be NonConvex) in the format [(in, out)]
- **lower_limit** (*integer*) – minimum number of active flows in the list
- **upper_limit** (*integer*) – maximum number of active flows in the list

Returns *the updated model*

Note: Flow objects required to be NonConvex

Constraint:

$$N_{X,min} \leq \sum_{n \in F} X_n(t) \leq N_{X,max} \forall t \in T$$

With *F* being the set of considered flows and *T* being the set of time steps.

The symbols used are defined as follows (with Variables (V) and Parameters (P)):

math. symbol	type	explanation
$X_n(t)$	V	status (0 or 1) of the flow n at time step t
$N_{X,min}$	P	lower_limit
$N_{X,max}$	P	lower_limit

---

oemof.solph.constraints.**limit_active_flow_count_by_keyword**(*model*, *keyword*, *lower_limit=0*, *upper_limit=None*)

This wrapper for limit_active_flow_count allows to set limits to the count of concurrently active flows by using a keyword instead of a list. The constraint will be named $(keyword)_count.

Parameters

- **model** (*oemof.solph.Model*) – Model to which constraints are added
- **keyword** (*string*) – keyword to consider (searches all NonConvexFlows)
- **lower_limit** (*integer*) – minimum number of active flows having the keyword
- **upper_limit** (*integer*) – maximum number of active flows having the keyword

Returns *the updated model*

See also:

limit_active_flow_count(), constraint_name(), flows()

oemof.solph.constraints.**shared_limit**(*model*, *quantity*, *limit_name*, *components*, *weights*, *lower_limit=0*, *upper_limit=None*)

Adds a constraint to the given model that restricts the weighted sum of variables to a corridor.

The following constraints are build:

$$l_{\text{low}} \leq \sum v_i(t) \times w_i(t) \leq l_{\text{up}} \forall t$$

Parameters

- **model** (*oemof.solph.Model*) – Model to which the constraint is added.
- **limit_name** (*string*) – Name of the constraint to create
- **quantity** (*pyomo.core.base.var.IndexedVar*) – Shared Pyomo variable for all components of a type.
- **components** (*list of components*) – list of components of the same type
- **weights** (*list of numeric values*) – has to have the same length as the list of components
- **lower_limit** (*numeric*) – the lower limit
- **upper_limit** (*numeric*) – the lower limit

Examples

The constraint can e.g. be used to define a common storage that is shared between parties but that do not exchange energy on balance sheet. Thus, every party has their own bus and storage, respectively, to model the energy flow. However, as the physical storage is shared, it has a common limit.

```
>>> import pandas as pd
>>> from oemof import solph
>>> date_time_index = pd.date_range('1/1/2012', periods=5, freq='H')
>>> energysystem = solph.EnergySystem(timeindex=date_time_index)
>>> b1 = solph.Bus(label="Party1Bus")
>>> b2 = solph.Bus(label="Party2Bus")
>>> storage1 = solph.components.GenericStorage(
...     label="Party1Storage",
...     nominal_storage_capacity=5,
...     inputs={b1: solph.Flow()},
...     outputs={b1: solph.Flow()})
>>> storage2 = solph.components.GenericStorage(
...     label="Party2Storage",
...     nominal_storage_capacity=5,
...     inputs={b1: solph.Flow()},
...     outputs={b1: solph.Flow()})
>>> energysystem.add(b1, b2, storage1, storage2)
>>> components = [storage1, storage2]
>>> model = solph.Model(energysystem)
>>> solph.constraints.shared_limit(
...    model,
...    model.GenericStorageBlock.storage_content,
...    "limit_storage", components,
...    [1, 1], upper_limit=5)
```

3.1.5 oemof.solph.console_scripts module

This module can be used to check the installation.

This is not an illustrated example.

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

SPDX-License-Identifier: MIT

oemof.solph.console_scripts.**check_oemof_installation**(*silent=False*)

3.1.6 oemof.solph.custom module

This module is designed to hold custom components with their classes and associated individual constraints (blocks) and groupings.

Therefore this module holds the class definition and the block directly located by each other.

SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de> SPDX-FileCopyrightText: Simon Hilpert SPDX-FileCopyrightText: Cord Kaldemeyer SPDX-FileCopyrightText: Patrik Schönfeldt SPDX-FileCopyrightText: Johannes Röder SPDX-FileCopyrightText: jakob-wo SPDX-FileCopyrightText: gplssm

SPDX-License-Identifier: MIT

class oemof.solph.custom.**ElectricalBus**(**args*, ***kwargs*)

Bases: *oemof.solph.network.Bus*

A electrical bus object. Every node has to be connected to Bus. This Bus is used in combination with ElectricalLine objects for linear optimal power flow (lopf) calculations.

Parameters

- **slack** (*boolean*) – If True Bus is slack bus for network
- **v_max** (*numeric*) – Maximum value of voltage angle at electrical bus
- **v_min** (*numeric*) – Mininum value of voltag angle at electrical bus
- **Note** (*This component is experimental. Use it with care.*)

Notes

The following sets, variables, constraints and objective parts are created

- *Bus*

The objects are also used inside:

- *ElectricalLine*

class oemof.solph.custom.**ElectricalLine**(**args*, ***kwargs*)

Bases: *oemof.solph.network.Flow*

An ElectricalLine to be used in linear optimal power flow calculations. based on angle formulation. Check out the Notes below before using this component!

Parameters

- **reactance** (*float or array of floats*) – Reactance of the line to be modelled
- **Note** (*This component is experimental. Use it with care.*)

Notes

- To use this object the connected buses need to be of the type *ElectricalBus*.
- It does not work together with flows that have set the attr.'nonconvex', i.e. unit commitment constraints are not possible
- Input and output of this component are set equal, therefore just use either only the input or the output to parameterize.
- Default attribute *min* of in/outflows is overwritten by -1 if not set differently by the user

The following sets, variables, constraints and objective parts are created

- *ElectricalLineBlock*

constraint_group()

class oemof.solph.custom.**ElectricalLineBlock**(**args*, ***kwargs*)
Bases: pyomo.core.base.block.SimpleBlock

Block for the linear relation of nodes with type class:*.ElectricalLine*

Note: This component is experimental. Use it with care.

The following constraints are created:

Linear relation om.ElectricalLine.electrical_flow[n,t]

$$flow(n, o, t) = 1/reactance(n, t)$$
$$cdot()voltage_angle(i(n), t) - volatage_angle(o(n), t),$$
$$\forall t$$
$$in$$
$$textrm TIMESTEPS,$$
$$\forall n$$
$$in$$
$$textrm ELECTRICAL\_LINES.$$

TODO: Add equate constraint of flows

The following variable are created:

TODO: Add voltage angle variable

TODO: Add fix slack bus voltage angle to zero constraint / bound

TODO: Add tests

CONSTRAINT_GROUP = True

class oemof.solph.custom.**GenericCAES**(**args*, ***kwargs*)
Bases: oemof.network.network.Transformer

Component *GenericCAES* to model arbitrary compressed air energy storages.

The full set of equations is described in: Kaldemeyer, C.; Boysen, C.; Tuschy, I. A Generic Formulation of Compressed Air Energy Storage as Mixed Integer Linear Program – Unit Commitment of Specific Technical Concepts in Arbitrary Market Environments Materials Today: Proceedings 00 (2018) 0000–0000 [currently in review]

Parameters

- **electrical_input** (*dict*) – Dictionary with key-value-pair of *oemof.Bus* and *oemof.Flow* object for the electrical input.
- **fuel_input** (*dict*) – Dictionary with key-value-pair of *oemof.Bus* and *oemof.Flow* object for the fuel input.
- **electrical_output** (*dict*) – Dictionary with key-value-pair of *oemof.Bus* and *oemof.Flow* object for the electrical output.
- **Note** (*This component is experimental. Use it with care.*)

Notes

The following sets, variables, constraints and objective parts are created

- `GenericCAES`

TODO: Add description for constraints. See referenced paper until then!

Examples

```
>>> from oemof import solph
>>> bel = solph.Bus(label='bel')
>>> bth = solph.Bus(label='bth')
>>> bgas = solph.Bus(label='bgas')
>>> # dictionary with parameters for a specific CAES plant
>>> concept = {
...     'cav_e_in_b': 0,
...     'cav_e_in_m': 0.6457267578,
...     'cav_e_out_b': 0,
...     'cav_e_out_m': 0.3739636077,
...     'cav_eta_temp': 1.0,
...     'cav_level_max': 211.11,
...     'cmp_p_max_b': 86.0918959849,
...     'cmp_p_max_m': 0.0679999932,
...     'cmp_p_min': 1,
...     'cmp_q_out_b': -19.3996965679,
...     'cmp_q_out_m': 1.1066036114,
...     'cmp_q_tes_share': 0,
...     'exp_p_max_b': 46.1294016678,
...     'exp_p_max_m': 0.2528340303,
...     'exp_p_min': 1,
...     'exp_q_in_b': -2.2073411014,
...     'exp_q_in_m': 1.129249765,
...     'exp_q_tes_share': 0,
...     'tes_eta_temp': 1.0,
...     'tes_level_max': 0.0}
>>> # generic compressed air energy storage (caes) plant
>>> caes = solph.custom.GenericCAES(
...     label='caes',
...     electrical_input={bel: solph.Flow()},
...     fuel_input={bgas: solph.Flow()},
...     electrical_output={bel: solph.Flow()},
...     params=concept, fixed_costs=0)
>>> type(caes)
<class 'oemof.solph.custom.GenericCAES'>
```

constraint_group()

class oemof.solph.custom.**GenericCAESBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for nodes of class:.*GenericCAES*.

Note: This component is experimental. Use it with care.

The following constraints are created:

(1) $P_{cmp}(t) = electrical\_input(t) \quad \forall t \in T$

(2) $P_{cmp\_max}(t) = m_{cmp\_max} \cdot CAS_{fil}(t-1) + b_{cmp\_max} \quad \forall t \in [1, t_{max}]$

(3) $P_{cmp\_max}(t) = b_{cmp\_max} \quad \forall t \notin [1, t_{max}]$

(4) $P_{cmp}(t) \leq P_{cmp\_max}(t) \quad \forall t \in T$

(5) $P_{cmp}(t) \geq P_{cmp\_min} \cdot ST_{cmp}(t) \quad \forall t \in T$

(6) $P_{cmp}(t) = m_{cmp\_max} \cdot CAS_{fil\_max} + b_{cmp\_max} \cdot ST_{cmp}(t) \quad \forall t \in T$

(7) $\dot{Q}_{cmp}(t) = m_{cmp\_q} \cdot P_{cmp}(t) + b_{cmp\_q} \cdot ST_{cmp}(t) \quad \forall t \in T$

(8) $\dot{Q}_{cmp}(t) = \dot{Q}_{cmp_out}(t) + \dot{Q}_{tes\_in}(t) \quad \forall t \in T$

(9) $r_{cmp\_tes} \cdot \dot{Q}_{cmp\_out}(t) = (1 - r_{cmp\_tes})\, \dot{Q}_{tes\_in}(t) \quad \forall t \in T$

(10) $P_{exp}(t) = electrical\_output(t) \quad \forall t \in T$

(11) $P_{exp\_max}(t) = m_{exp\_max} CAS_{fil}(t-1) + b_{exp\_max} \quad \forall t \in [1, t_{\max}]$

(12) $P_{exp\_max}(t) = b_{exp\_max} \quad \forall t \notin [1, t_{\max}]$

(13) $P_{exp}(t) \leq P_{exp\_max}(t) \quad \forall t \in T$

(14) $P_{exp}(t) \geq P_{exp\_min}(t) \cdot ST_{exp}(t) \quad \forall t \in T$

(15) $P_{exp}(t) \leq m_{exp\_max} \cdot CAS_{fil\_max} + b_{exp\_max} \cdot ST_{exp}(t) \quad \forall t \in T$

(16) $\dot{Q}_{exp}(t) = m_{exp\_q} \cdot P_{exp}(t) + b_{cxp\_q} \cdot ST_{cxp}(t) \quad \forall t \in T$

(17) $\dot{Q}_{exp\_in}(t) = fuel\_input(t) \quad \forall t \in T$

(18) $\dot{Q}_{exp}(t) = \dot{Q}_{exp\_in}(t) + \dot{Q}_{tes\_out}(t) + \dot{Q}_{cxp\_add}(t) \quad \forall t \in T$

(19) $r_{exp\_tes} \cdot \dot{Q}_{exp\_in}(t) = (1 - r_{exp\_tes})(\dot{Q}_{tes\_out}(t) + \dot{Q}_{exp\_add}(t)) \quad \forall t \in T$

(20) $\dot{E}_{cas\_in}(t) = m_{cas\_in} \cdot P_{cmp}(t) + b_{cas\_in} \cdot ST_{cmp}(t) \quad \forall t \in T$

(21) $\dot{E}_{cas\_out}(t) = m_{cas\_out} \cdot P_{cmp}(t) + b_{cas\_out} \cdot ST_{cmp}(t) \quad \forall t \in T$

(22) $\eta_{cas\_tmp} \cdot CAS_{fil}(t) = CAS_{fil}(t-1) + \tau \left(\dot{E}_{cas\_in}(t) - \dot{E}_{cas\_out}(t) \right) \quad \forall t \in [1, t_{max}]$

(23) $\eta_{cas\_tmp} \cdot CAS_{fil}(t) = \tau \left(\dot{E}_{cas\_in}(t) - \dot{E}_{cas\_out}(t) \right) \quad \forall t \notin [1, t_{max}]$

(24) $CAS_{fil}(t) \leq CAS_{fil\_max} \quad \forall t \in T$

(25) $TES_{fil}(t) = TES_{fil}(t-1) + \tau \left(\dot{Q}_{tes\_in}(t) - \dot{Q}_{tes\_out}(t) \right) \quad \forall t \in [1, t_{max}]$

(26) $TES_{fil}(t) = \tau \left(\dot{Q}_{tes\_in}(t) - \dot{Q}_{tes\_out}(t) \right) \quad \forall t \notin [1, t_{max}]$

(27) $TES_{fil}(t) \leq TES_{fil\_max} \quad \forall t \in T$

Table: Symbols and attribute names of variables and parameters

Table 6: Variables (V) and Parameters (P)

symbol	attribute	type	explanation
ST_{cmp}	`cmp_st[n,t]`	V	Status of compression
P_{cmp}	`cmp_p[n,t]`	V	Compression power
P_{cmp_max}	`cmp_p_max[n,t]`	V	Max. compression power
$\dot{Q}_{cmp}$	`cmp_q_out_sum[n, t]`	V	**Summed** heat flow in compression
$\dot{Q}_{cmp\_out}$	`cmp_q_waste[n,t]`	V	Waste heat flow from compression
$ST_{exp}(t)$	`exp_st[n,t]`	V	Status of expansion (binary)
$P_{exp}(t)$	`exp_p[n,t]`	V	Expansion power
$P_{exp\_max}(t)$	`exp_p_max[n,t]`	V	Max. expansion power
$\dot{Q}_{exp}(t)$	`exp_q_in_sum[n, t]`	V	Summed heat flow in expansion
$\dot{Q}_{exp\_in}(t)$	`exp_q_fuel_in[n, t]`	V	Heat (external) flow into expansion
$\dot{Q}_{exp\_add}(t)$	`exp_q_add_in[n, t]`	V	Additional heat flow into expansion
$CAV_{fil}(t)$	`cav_level[n,t]`	V	**Filling level** if CAE
$\dot{E}_{cas\_in}(t)$	`cav_e_in[n,t]`	V	Exergy flow into CAS
$\dot{E}_{cas\_out}(t)$	`cav_e_out[n,t]`	V	Exergy flow from CAS
$TES_{fil}(t)$	`tes_level[n,t]`	V	Filling level of Thermal Energy Storage (TES)
$\dot{Q}_{tes\_in}(t)$	`tes_e_in[n,t]`	V	**Heat** flow into TES
$\dot{Q}_{tes\_out}(t)$	`tes_e_out[n,t]`	V	**Heat** flow from TES
b_{cmp_max}	`cmp_p_max_b[n,t]`	P	**Specific** y-intersection
b_{cmp_q}	`cmp_q_out_b[n,t]`	P	Specific y-intersection
b_{exp_max}	`exp_p_max_b[n,t]`	P	Specific y-intersection
b_{exp_q}	`exp_q_in_b[n,t]`	P	Specific y-intersection
b_{cas_in}	`cav_e_in_b[n,t]`	P	Specific y-intersection
b_{cas_out}	`cav_e_out_b[n,t]`	P	Specific y-intersection
m_{cmp_max}	`cmp_p_max_m[n,t]`	P	**Specific** slope
m_{cmp_q}	`cmp_q_out_m[n,t]`	P	**Specific** slope

Continued on next page

Table 6 – continued from previous page

symbol	attribute	type	explanation
m_{exp_max}	`exp_p_max_m[n,t]`	P	**Specific** slope
m_{exp_q}	`exp_q_in_m[n,t]`	P	**Specific** slope
m_{cas_in}	`cav_e_in_m[n,t]`	P	**Specific** slope
m_{cas_out}	`cav_e_out_m[n,t]`	P	**Specific** slope
P_{cmp_min}	`cmp_p_min[n,t]`	P	Min. compression power
r_{cmp_tes}	`cmp_q_tes_share[n, t]`	P	**Ratio** between waste heat flow and heat flow into TES
r_{exp_tes}	`exp_q_tes_share[n, t]`	P	**Ratio** **between external heat flow into expansi** additional source
τ	`m. timeincrement[n, t]`	P	**Time interval** length
TES_{fil_max}	`tes_level_max[n, t]`	P	Max. filling level of TES
CAS_{fil_max}	`cav_level_max[n, t]`	P	**Max.** filling level of TES
τ	`cav_eta_tmp[n,t]`	P	**Temporal efficiency** (loss factor to take intertemporal losses into account)
electrical_input	`flow[list(n. electrical_input. keys())[0], n, t]`	P	Electr. power input into compression
electrical_output	`flow[n, list(n. electrical_output. keys())[0], t]`	P	Electr. power output of expansion

Continued on next page

Table 6 – continued from previous page

symbol	attribute	type	explanation
$fuel\_input$	`flow[list(n.fuel_input.keys())[0], n, t]`	P	**Heat input** (external) into Expansion

CONSTRAINT_GROUP = True

class oemof.solph.custom.**Link**(**args*, ***kwargs*)

Bases: *oemof.solph.network.Transformer*

A Link object with 1...2 inputs and 1...2 outputs.

Parameters

- **conversion_factors** (*dict*) – Dictionary containing conversion factors for conversion of each flow. Keys are the connected tuples (input, output) bus objects. The dictionary values can either be a scalar or an iterable with length of time horizon for simulation.
- **Note** (*This component is experimental. Use it with care.*)

Notes

The sets, variables, constraints and objective parts are created

- *LinkBlock*

Examples

```
>>> from oemof import solph
>>> bel0 = solph.Bus(label="el0")
>>> bel1 = solph.Bus(label="el1")
```

```
>>> link = solph.custom.Link(
...     label="transshipment_link",
...     inputs={bel0: solph.Flow(), bel1: solph.Flow()},
...     outputs={bel0: solph.Flow(), bel1: solph.Flow()},
...     conversion_factors={(bel0, bel1): 0.92, (bel1, bel0): 0.99})
>>> print(sorted([x[1][5] for x in link.conversion_factors.items()]))
[0.92, 0.99]
```

```
>>> type(link)
<class 'oemof.solph.custom.Link'>
```

```
>>> sorted([str(i) for i in link.inputs])
['el0', 'el1']
```

```
>>> link.conversion_factors[(bel0, bel1)][3]
0.92
```

constraint_group()

class oemof.solph.custom.**LinkBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Block for the relation of nodes with type *Link*

Note: This component is experimental. Use it with care.

The following constraints are created:

TODO: Add description for constraints TODO: Add tests

CONSTRAINT_GROUP = True

class oemof.solph.custom.**SinkDSM**(*demand*, *capacity_up*, *capacity_down*, *method*, *shift_interval=None*, *delay_time=None*, *cost_dsm_up=0*, *cost_dsm_down=0*, ***kwargs*)

Bases: *oemof.solph.network.Sink*

Demand Side Management implemented as Sink with flexibility potential.

Based on the paper by Zerrahn, Alexander and Schill, Wolf-Peter (2015): On the representation of demand-side management in power system models, in: Energy (84), pp. 840-845, 10.1016/j.energy.2015.03.037, accessed 17.09.2019, pp. 842-843.

SinkDSM adds additional constraints that allow to shift energy in certain time window constrained by `capacity_up` and `capacity_down`.

Parameters

- **demand** (*numeric*) – original electrical demand
- **capacity_up** (*int or array*) – maximum DSM capacity that may be increased
- **capacity_down** (*int or array*) – maximum DSM capacity that may be reduced
- **method** (*'interval' , 'delay'*) – Choose one of the DSM modelling approaches. Read notes about which parameters to be applied for which approach.

  interval :

  Simple model in which the load shift must be compensated in a predefined fixed interval (`shift_interval` is mandatory). Within time windows of the length `shift_interval` DSM up and down shifts are balanced. See *SinkDSMIntervalBlock* for details.

  delay :

  Sophisticated model based on the formulation by Zerrahn & Schill (2015). The load-shift of the component must be compensated in a predefined delay-time (`delay_time` is mandatory). For details see *SinkDSMDelayBlock*.
- **shift_interval** (*int*) – Only used when `method` is set to 'interval'. Otherwise, can be None. It's the interval in which between DSM_t^{up} and DSM_t^{down} have to be compensated.
- **delay_time** (*int*) – Only used when `method` is set to 'delay'. Otherwise, can be None. Length of symmetrical time windows around t in which DSM_t^{up} and $DSM_{t,tt}^{down}$ have to be compensated.
- **cost_dsm_up** (`int`) – Cost per unit of DSM activity that increases the demand
- **cost_dsm_down** (`int`) – Cost per unit of DSM activity that decreases the demand

---

Note:

- This component is a candidate component. It's implemented as a custom component for users that like to use and test the component at early stage. Please report issues to improve the component.

- As many constraints and dependencies are created in method 'delay', computational cost might be high with a large 'delay_time' and with model of high temporal resolution
- Using `method` 'delay' might result in demand shifts that exceed the specified delay time by activating up and down simultaneously in the time steps between to DSM events.
- It's not recommended to assign cost to the flow that connects *SinkDSM* with a bus. Instead, use `cost_dsm_up` or `cost_dsm_down`

---

constraint_group()

class oemof.solph.custom.**SinkDSMDelayBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Constraints for SinkDSM with "delay" method

The following constraints are created for method = 'delay':

$$(1) \quad \dot{E}_t = demand_t + DSM_t^{up} - \sum_{tt=t-L}^{t+L} DSM_{t,tt}^{do} \quad \forall t \in \mathbb{T}$$

$$(2) \quad DSM_t^{up} = \sum_{tt=t-L}^{t+L} DSM_{t,tt}^{do} \quad \forall t \in \mathbb{T}$$

$$(3) \quad DSM_t^{up} \leq E_t^{up} \quad \forall t \in \mathbb{T}$$

$$(4) \quad \sum_{tt=t-L}^{t+L} DSM_{t,tt}^{do} \leq E_t^{do} \quad \forall t \in \mathbb{T}$$

$$(5) \quad DSM_t^{up} + \sum_{tt=t-L}^{t+L} DSM_{t,tt}^{do} \leq max\{E_t^{up}, E_t^{do}\} \quad \forall t \in \mathbb{T}$$

Table: Symbols and attribute names of variables and parameters

Table 7: Variables (V) and Parameters (P)

symbol	attribute	type	explanation
DSM_t^{up}	`dsm_do[g,t,tt]`	V	DSM up shift (additional load)
$DSM_{t,tt}^{do}$	`dsm_up[g,t]`	V	DSM down shift (less load)
$\dot{E}_t$	`flow[g,t]`	V	Energy flowing in from electrical bus
L	`delay_time`	P	Delay time for load shift
$demand_t$	`demand[t]`	P	Electrical demand series
E_t^{do}	`capacity_down[tt]`	P	Capacity DSM down shift
E_t^{up}	`capacity_up[tt]`	P	Capacity DSM up shift
$\mathbb{T}$		P	Time steps

CONSTRAINT_GROUP = True

class oemof.solph.custom.**SinkDSMIntervalBlock**(**args*, ***kwargs*)

Bases: pyomo.core.base.block.SimpleBlock

Constraints for SinkDSM with "interval" method

The following constraints are created for method = 'interval':

$$
\begin{aligned}
&(1) \quad \dot{E}_t = demand_t + DSM_t^{up} - DSM_t^{do} \quad \forall t \in \mathbb{T} \\
&(2) \quad DSM_t^{up} \leq E_t^{up} \quad \forall t \in \mathbb{T} \\
&(3) \quad DSM_t^{do} \leq E_t^{do} \quad \forall t \in \mathbb{T} \\
&(4) \quad \sum_{t=t_s}^{t_s+\tau} DSM_t^{up} = \sum_{t=t_s}^{t_s+\tau} DSM_t^{do} \quad \forall t_s \in \{k \in \mathbb{T} \mid k \mod \tau = 0\}
\end{aligned}
$$

Table: Symbols and attribute names of variables and parameters

Table 8: Variables (V) and Parameters (P)

symbol	attribute	type	explanation
DSM_t^{up}	capacity_up	V	DSM up shift
DSM_t^{do}	capacity_down	V	DSM down shift
$\dot{E}_t$	inputs	V	Energy flowing in from electrical bus
$demand_t$	demand[t]	P	Electrical demand series
E_t^{do}	capacity_down[tt]	P	Capacity DSM down shift capacity
E_t^{up}	capacity_up[tt]	P	Capacity DSM up shift
τ	shift_interval	P	Shift interval
$\mathbb{T}$		P	Time steps

CONSTRAINT_GROUP = True

3.1.7 oemof.solph.groupings module

Groupings needed on an energy system for it to work with solph.

If you want to use solph on an energy system, you need to create it with these groupings specified like this:

from oemof.network import EnergySystem import solph

energy_system = EnergySystem(groupings=solph.GROUPINGS)

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

SPDX-License-Identifier: MIT

oemof.solph.groupings.**constraint_grouping**(*node*, *fallback=<function <lambda>>*)

Grouping function for constraints.

This function can be passed in a list to groupings of *oemof.solph.network.EnergySystem*.

Parameters

- **node** (`Node <oemof.network.Node>`) – The node for which the figure out a constraint group.
- **fallback** (*callable, optional*) – A function of one argument. If *node* doesn't have a *constraint_group* attribute, this is used to group the node instead. Defaults to not group the node at all.

3.1.8 oemof.solph.helpers module

This is a collection of helper functions which work on their own and can be used by various classes. If there are too many helper-functions, they will be sorted in different modules.

SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de> SPDX-FileCopyrightText: Caroline Möller SPDX-FileCopyrightText: henhuy SPDX-FileCopyrightText: gplssm SPDX-FileCopyrightText: Stephan Günther SPDX-FileCopyrightText: elisapap

SPDX-License-Identifier: MIT

`oemof.solph.helpers.`**`calculate_timeincrement`**(*timeindex*, *fill_value=None*)
Calculates timeincrement for *timeindex*

Parameters

- **timeindex** (*pd.DatetimeIndex*) – timeindex of energysystem
- **fill_value** (*numerical*) – timeincrement for first timestep in hours

`oemof.solph.helpers.`**`extend_basic_path`**(*subfolder*)
Returns a path based on the basic oemof path and creates it if necessary. The subfolder is the name of the path extension.

`oemof.solph.helpers.`**`flatten`**(*d*, *parent_key=''*, *sep='_'*)
Flatten dictionary by compressing keys.

See: **https://stackoverflow.com/questions/6027558/** flatten-nested-python-dictionaries-compressing-keys

d : dictionary sep : separator for flattening keys

Returns *dict*

`oemof.solph.helpers.`**`get_basic_path`**()
Returns the basic oemof path and creates it if necessary. The basic path is the '.oemof' folder in the $HOME directory.

3.1.9 oemof.solph.models module

Solph Optimization Models.

SPDX-FileCopyrightText: Uwe Krien <krien@uni-bremen.de> SPDX-FileCopyrightText: Simon Hilpert SPDX-FileCopyrightText: Cord Kaldemeyer SPDX-FileCopyrightText: gplssm SPDX-FileCopyrightText: Patrik Schönfeldt

SPDX-License-Identifier: MIT

`class` `oemof.solph.models.`**`BaseModel`**(*energysystem*, ***kwargs*)
Bases: `pyomo.core.base.PyomoModel.ConcreteModel`

The BaseModel for other solph-models (Model, MultiPeriodModel, etc.)

Parameters

- **energysystem** (*EnergySystem object*) – Object that holds the nodes of an oemof energy system graph

- **constraint_groups** (*list (optional)*) – Solph looks for these groups in the given energy system and uses them to create the constraints of the optimization problem. Defaults to *Model.CONSTRAINTS*
- **objective_weighting** (*array like (optional)*) – Weights used for temporal objective function expressions. If nothing is passed *timeincrement* will be used which is calculated from the freq length of the energy system timeindex .
- **auto_construct** (*boolean*) – If this value is true, the set, variables, constraints, etc. are added, automatically when instantiating the model. For sequential model building process set this value to False and use methods *_add_parent_block_sets*, *_add_parent_block_variables*, *_add_blocks*, *_add_objective*
- **Attributes**
- **———**
- **timeincrement** (*sequence*) – Time increments.
- **flows** (*dict*) – Flows of the model.
- **name** (*str*) – Name of the model.
- **es** (*solph.EnergySystem*) – Energy system of the model.
- **meta** (*pyomo.opt.results.results_.SolverResults* or None) – Solver results.
- **dual** (*... or None*)
- **rc** (*... or None*)

CONSTRAINT_GROUPS = []

receive_duals()
Method sets solver suffix to extract information about dual variables from solver. Shadow prices (duals) and reduced costs (rc) are set as attributes of the model.

relax_problem()
Relaxes integer variables to reals of optimization model self.

results()
Returns a nested dictionary of the results of this optimization

solve(*solver='cbc'*, *solver_io='lp'*, ***kwargs*)
Takes care of communication with solver to solve the model.

Parameters

- **solver** (*string*) – solver to be used e.g. "glpk","gurobi","cplex"
- **solver_io** (*string*) – pyomo solver interface file format: "lp","python","nl", etc.
- ****kwargs** (*keyword arguments*) – Possible keys can be set see below:

Other Parameters

- **solve_kwargs** (*dict*) – Other arguments for the pyomo.opt.SolverFactory.solve() method Example : {"tee":True}
- **cmdline_options** (*dict*) – Dictionary with command line options for solver e.g. {"mipgap":"0.01"} results in "–mipgap 0.01" {"interior":""} results in "–interior" Gurobi solver takes numeric parameter values such as {"method": 2}

class oemof.solph.models.**Model**(*energysystem*, ***kwargs*)
Bases: *oemof.solph.models.BaseModel*

An energy system model for operational and investment optimization.

Parameters

- **energysystem** (*EnergySystem object*) – Object that holds the nodes of an oemof energy system graph
- **constraint_groups** (*list*) – Solph looks for these groups in the given energy system and uses them to create the constraints of the optimization problem. Defaults to *Model.CONSTRAINTS*
- ****The following basic sets are created****
- **NODES** – A set with all nodes of the given energy system.
- **TIMESTEPS** – A set with all timesteps of the given time horizon.
- **FLOWS** – A 2 dimensional set with all flows. Index: *(source, target)*
- ****The following basic variables are created****
- **flow** – Flow from source to target indexed by FLOWS, TIMESTEPS. Note: Bounds of this variable are set depending on attributes of the corresponding flow object.

CONSTRAINT_GROUPS = [<class 'oemof.solph.blocks.Bus'>, <class 'oemof.solph.blocks.Tran

3.1.10 oemof.solph.network module

Classes used to model energy supply systems within solph.

Classes are derived from oemof core network classes and adapted for specific optimization tasks. An energy system is modelled as a graph/network of nodes with very specific constraints on which types of nodes are allowed to be connected.

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

SPDX-License-Identifier: MIT

class oemof.solph.network.**Bus**(**args*, ***kwargs*)
Bases: oemof.network.network.Bus

A balance object. Every node has to be connected to Bus.

Notes

The following sets, variables, constraints and objective parts are created

- *Bus*

constraint_group()

class oemof.solph.network.**EnergySystem**(***kwargs*)
Bases: oemof.network.energy_system.EnergySystem

A variant of EnergySystem specially tailored to solph.

In order to work in tandem with solph, instances of this class always use *solph.GROUPINGS <oemof.solph.GROUPINGS>*. If custom groupings are supplied via the *groupings* keyword argument, *solph.GROUPINGS <oemof.solph.GROUPINGS>* is prepended to those.

If you know what you are doing and want to use solph without *solph.GROUPINGS <oemof.solph.GROUPINGS>*, you can just use `core's EnergySystem` directly.

class `oemof.solph.network.`**Flow**(***kwargs*)

Bases: `oemof.network.network.Edge`

Defines a flow between two nodes.

Keyword arguments are used to set the attributes of this flow. Parameters which are handled specially are noted below. For the case where a parameter can be either a scalar or an iterable, a scalar value will be converted to a sequence containing the scalar value at every index. This sequence is then stored under the paramter's key.

Parameters

- **nominal_value** (numeric, P_{nom}) – The nominal value of the flow. If this value is set the corresponding optimization variable of the flow object will be bounded by this value multiplied with min(lower bound)/max(upper bound).
- **max** (numeric (iterable or scalar), f_{max}) – Normed maximum value of the flow. The flow absolute maximum will be calculated by multiplying `nominal_value` with `max`
- **min** (numeric (iterable or scalar), f_{min}) – Normed minimum value of the flow (see `max`).
- **fix** (numeric (iterable or scalar), f_{actual}) – Normed fixed value for the flow variable. Will be multiplied with the `nominal_value` to get the absolute value. If `fixed` is set to `True` the flow variable will be fixed to *fix * nominal_value*, i.e. this value is set exogenous.
- **positive_gradient** (`dict`, default: *{'ub': None, 'costs': 0}*) –

  A dictionary containing the following two keys:

  – *'ub'*: numeric (iterable, scalar or None), the normed *upper bound* on the positive difference (*flow[t-1] < flow[t]*) of two consecutive flow values.

  – *'costs'*: numeric (scalar or None), the gradient cost per unit.
- **negative_gradient** (`dict`, default: *{'ub': None, 'costs': 0}*) –

  A dictionary containing the following two keys:

  – *'ub'*: numeric (iterable, scalar or None), the normed *upper bound* on the negative difference (*flow[t-1] > flow[t]*) of two consecutive flow values.

  – *'costs'*: numeric (scalar or None), the gradient cost per unit.
- **summed_max** (numeric, $f_{sum,max}$) – Specific maximum value summed over all timesteps. Will be multiplied with the nominal_value to get the absolute limit.
- **summed_min** (numeric, $f_{sum,min}$) – see above
- **variable_costs** (*numeric (iterable or scalar)*) – The costs associated with one unit of the flow. If this is set the costs will be added to the objective expression of the optimization problem.
- **fixed** (*boolean*) – Boolean value indicating if a flow is fixed during the optimization problem to its ex-ante set value. Used in combination with the `fix`.
- **investment** (*Investment*) – Object indicating if a nominal_value of the flow is determined by the optimization problem. Note: This will refer all attributes to an investment variable instead of to the nominal_value. The nominal_value should not be set (or set to None) if an investment object is used.
- **nonconvex** (*NonConvex*) – If a nonconvex flow object is added here, the flow constraints will be altered significantly as the mathematical model for the flow will be different, i.e.

constraint etc. from *NonConvexFlow* will be used instead of *Flow*. Note: at the moment this does not work if the investment attribute is set .

Notes

The following sets, variables, constraints and objective parts are created

- *Flow*
- *InvestmentFlow* (additionally if Investment object is present)
- ***NonConvexFlow*** **(If** nonconvex object is present, CAUTION: replaces *Flow* class and a MILP will be build)

Examples

Creating a fixed flow object:

```
>>> f = Flow(fix=[10, 4, 4], variable_costs=5)
>>> f.variable_costs[2]
5
>>> f.fix[2]
4
```

Creating a flow object with time-depended lower and upper bounds:

```
>>> f1 = Flow(min=[0.2, 0.3], max=0.99, nominal_value=100)
>>> f1.max[1]
0.99
```

class oemof.solph.network.**Sink**(**args*, ***kwargs*)
Bases: oemof.network.network.Sink

An object with one input flow.

constraint_group()

class oemof.solph.network.**Source**(**args*, ***kwargs*)
Bases: oemof.network.network.Source

An object with one output flow.

constraint_group()

class oemof.solph.network.**Transformer**(**args*, ***kwargs*)
Bases: oemof.network.network.Transformer

A linear Transformer object with n inputs and n outputs.

Parameters **conversion_factors** (*dict*) – Dictionary containing conversion factors for conversion of each flow. Keys are the connected bus objects. The dictionary values can either be a scalar or an iterable with length of time horizon for simulation.

Examples

Defining an linear transformer:

```
>>> from oemof import solph
>>> bgas = solph.Bus(label='natural_gas')
>>> bcoal = solph.Bus(label='hard_coal')
>>> bel = solph.Bus(label='electricity')
>>> bheat = solph.Bus(label='heat')
```

```
>>> trsf = solph.Transformer(
...     label='pp_gas_1',
...     inputs={bgas: solph.Flow(), bcoal: solph.Flow()},
...     outputs={bel: solph.Flow(), bheat: solph.Flow()},
...     conversion_factors={bel: 0.3, bheat: 0.5,
...                         bgas: 0.8, bcoal: 0.2})
>>> print(sorted([x[1][5] for x in trsf.conversion_factors.items()]))
[0.2, 0.3, 0.5, 0.8]
```

```
>>> type(trsf)
<class 'oemof.solph.network.Transformer'>
```

```
>>> sorted([str(i) for i in trsf.inputs])
['hard_coal', 'natural_gas']
```

```
>>> trsf_new = solph.Transformer(
...     label='pp_gas_2',
...     inputs={bgas: solph.Flow()},
...     outputs={bel: solph.Flow(), bheat: solph.Flow()},
...     conversion_factors={bel: 0.3, bheat: 0.5})
>>> trsf_new.conversion_factors[bgas][3]
1
```

Notes

The following sets, variables, constraints and objective parts are created

- *Transformer*

constraint_group()

3.1.11 oemof.solph.options module

Optional classes to be added to a network class.

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

SPDX-License-Identifier: MIT

class oemof.solph.options.**Investment**(*maximum=inf*, *minimum=0*, *ep_costs=0*, *existing=0*, *nonconvex=False*, *offset=0*, ***kwargs*)

Bases: object

Parameters

- **maximum** (float, $P_{invest,max}$ or $E_{invest,max}$) – Maximum of the additional invested capacity

- **minimum** (float, $P_{invest,min}$ or $E_{invest,min}$) – Minimum of the additional invested capacity. If *nonconvex* is *True*, *minimum* defines the threshold for the invested capacity.
- **ep_costs** (float, $c_{invest,var}$) – Equivalent periodical costs for the investment per flow capacity.
- **existing** (float, P_{exist} or E_{exist}) – Existing / installed capacity. The invested capacity is added on top of this value. Not applicable if *nonconvex* is set to *True*.
- **nonconvex** (*bool*) – If *True*, a binary variable for the status of the investment is created. This enables additional fix investment costs (*offset*) independent of the invested flow capacity. Therefore, use the *offset* parameter.
- **offset** (float, $c_{invest,fix}$) – Additional fix investment costs. Only applicable if *nonconvex* is set to *True*.

For the variables, constraints and parts of the objective function, which are created, see *oemof.solph.blocks.InvestmentFlow* and *oemof.solph.components.GenericInvestmentStorageBlock*.

class oemof.solph.options.**NonConvex**(***kwargs*)
Bases: object

Parameters

- **startup_costs** (*numeric (iterable or scalar)*) – Costs associated with a start of the flow (representing a unit).
- **shutdown_costs** (*numeric (iterable or scalar)*) – Costs associated with the shutdown of the flow (representing a unit).
- **activity_costs** (*numeric (iterable or scalar)*) – Costs associated with the active operation of the flow, independently from the actual output.
- **minimum_uptime** (*numeric (1 or positive integer)*) – Minimum time that a flow must be greater then its minimum flow after startup. Be aware that minimum up and downtimes can contradict each other and may lead to infeasible problems.
- **minimum_downtime** (*numeric (1 or positive integer)*) – Minimum time a flow is forced to zero after shutting down. Be aware that minimum up and downtimes can contradict each other and may to infeasible problems.
- **maximum_startups** (*numeric (0 or positive integer)*) – Maximum number of start-ups.
- **maximum_shutdowns** (*numeric (0 or positive integer)*) – Maximum number of shutdowns.
- **initial_status** (*numeric (0 or 1)*) – Integer value indicating the status of the flow in the first time step (0 = off, 1 = on). For minimum up and downtimes, the initial status is set for the respective values in the edge regions e.g. if a minimum uptime of four timesteps is defined, the initial status is fixed for the four first and last timesteps of the optimization period. If both, up and downtimes are defined, the initial status is set for the maximum of both e.g. for six timesteps if a minimum downtime of six timesteps is defined in addition to a four timestep minimum uptime.

max_up_down
Compute or return the _max_up_down attribute.

3.1.12 oemof.solph.plumbing module

Plumbing stuff.

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

SPDX-License-Identifier: MIT

oemof.solph.plumbing.**sequence**(*iterable_or_scalar*)

Tests if an object is iterable (except string) or scalar and returns a the original sequence if object is an iterable and a 'emulated' sequence object of class _Sequence if object is a scalar or string.

Parameters **iterable_or_scalar** (*iterable or None or int or float*)

Examples

```
>>> sequence([1,2])
[1, 2]
```

```
>>> x = sequence(10)
>>> x[0]
10
```

```
>>> x[10]
10
>>> print(x)
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

3.1.13 oemof.solph.processing module

Modules for providing a convenient data structure for solph results.

Information about the possible usage is provided within the examples.

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

SPDX-License-Identifier: MIT

oemof.solph.processing.**convert_keys_to_strings**(*result*, *keep_none_type=False*)

Convert the dictionary keys to strings.

All (tuple) keys of the result object e.g. results[(pp1, bus1)] are converted into strings that represent the object labels e.g. results[('pp1','bus1')].

oemof.solph.processing.**create_dataframe**(*om*)

Create a result dataframe with all optimization data.

Results from Pyomo are written into pandas DataFrame where separate columns are created for the variable index e.g. for tuples of the flows and components or the timesteps.

oemof.solph.processing.**get_timestep**(*x*)

Get the timestep from oemof tuples.

The timestep from tuples *(n, n, int)*, *(n, n)*, *(n, int)* and (n,) is fetched as the last element. For time-independent data (scalars) zero ist returned.

oemof.solph.processing.**get_tuple**(*x*)

Get oemof tuple within iterable or create it.

Tuples from Pyomo are of type *(n, n, int)*, *(n, n)* and *(n, int)*. For single nodes *n* a tuple with one object *(n,)* is created.

oemof.solph.processing.**meta_results**(*om*, *undefined=False*)
Fetch some meta data from the Solver. Feel free to add more keys.

Valid keys of the resulting dictionary are: 'objective', 'problem', 'solver'.

om [oemof.solph.Model] A solved Model.

undefined [bool] By default (False) only defined keys can be found in the dictionary. Set to True to get also the undefined keys.

Returns *dict*

oemof.solph.processing.**parameter_as_dict**(*system*, *exclude_none=True*)
Create a result dictionary containing node parameters.

Results are written into a dictionary of pandas objects where a Series holds all scalar values and a dataframe all sequences for nodes and flows. The dictionary is keyed by flows (n, n) and nodes (n, None), e.g. *parameter[(n, n)]['sequences']* or *parameter[(n, n)]['scalars']*.

Parameters

- **system** (*energy_system.EnergySystem*) – A populated energy system.
- **exclude_none** (*bool*) – If True, all scalars and sequences containing None values are excluded

Returns **dict** (*Parameters for all nodes and flows*)

oemof.solph.processing.**remove_timestep**(*x*)
Remove the timestep from oemof tuples.

The timestep is removed from tuples of type *(n, n, int)* and *(n, int)*.

oemof.solph.processing.**results**(*om*)
Create a result dictionary from the result DataFrame.

Results from Pyomo are written into a dictionary of pandas objects where a Series holds all scalar values and a dataframe all sequences for nodes and flows. The dictionary is keyed by the nodes e.g. *results[idx]['scalars']* and flows e.g. *results[n, n]['sequences']*.

3.1.14 oemof.solph.views module

Modules for providing convenient views for solph results.

Information about the possible usage is provided within the examples.

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

SPDX-License-Identifier: MIT

class oemof.solph.views.**NodeOption**
Bases: str, enum.Enum

An enumeration.

All = 'all'

HasInputs = 'has_inputs'

HasOnlyInputs = 'has_only_inputs'

HasOnlyOutputs = 'has_only_outputs'

HasOutputs = 'has_outputs'

oemof.solph.views.**convert_to_multiindex**(*group*, *index_names=None*, *droplevel=None*)
Convert dict to pandas DataFrame with multiindex

Parameters

- **group** (*dict*) – Sequences of the oemof.solph.Model.results dictionary
- **index_names** (*arraylike*) – Array with names of the MultiIndex
- **droplevel** (*arraylike*) – List containing levels to be dropped from the dataframe

oemof.solph.views.**filter_nodes**(*results*, *option=<NodeOption.All: 'all'>*, *exclude_busses=False*)
Get set of nodes from results-dict for given node option.

This function filters nodes from results for special needs. At the moment, the following options are available:

- *NodeOption.All*: *'all'*: Returns all nodes
- ***NodeOption.HasOutputs*: *'has_outputs'*:** Returns nodes with an output flow (eg. Transformer, Source)
- ***NodeOption.HasInputs*: *'has_inputs'*:** Returns nodes with an input flow (eg. Transformer, Sink)
- ***NodeOption.HasOnlyOutputs*: *'has_only_outputs'*:** Returns nodes having only output flows (eg. Source)
- ***NodeOption.HasOnlyInputs*: *'has_only_inputs'*:** Returns nodes having only input flows (eg. Sink)

Additionally, busses can be excluded by setting *exclude_busses* to *True*.

Parameters

- **results** (*dict*)
- **option** (*NodeOption*)
- **exclude_busses** (*bool*) – If set, all bus nodes are excluded from the resulting node set.

Returns set – A set of Nodes.

oemof.solph.views.**get_node_by_name**(*results*, **names*)
Searches results for nodes

Names are looked up in nodes from results and either returned single node (in case only one name is given) or as list of nodes. If name is not found, None is returned.

oemof.solph.views.**net_storage_flow**(*results*, *node_type*)
Calculates the net storage flow for storage models that have one input edge and one output edge both with flows within the domain of non-negative reals.

Parameters

- **results** (*dict*) – A result dictionary from a solved oemof.solph.Model object
- **node_type** (*oemof.solph class*) – Specifies the type for which (storage) type net flows are calculated

Returns

- *pandas.DataFrame object with multiindex colums. Names of levels of columns*
- **are** (*from, to, net_flow.*)

Examples

import oemof.solph as solph from oemof.outputlib import views

solve oemof solph model 'm' # Then collect node weights views.net_storage_flow(m.results(), node_type=solph.GenericStorage)

oemof.solph.views.**node**(*results*, *node*, *multiindex=False*, *keep_none_type=False*)
Obtain results for a single node e.g. a Bus or Component.

Either a node or its label string can be passed. Results are written into a dictionary which is keyed by 'scalars' and 'sequences' holding respective data in a pandas Series and DataFrame.

oemof.solph.views.**node_input_by_type**(*results*, *node_type*, *droplevel=None*)
Gets all inputs for all nodes of the type *node_type* and returns a dataframe.

Parameters

- **results** (*dict*) – A result dictionary from a solved oemof.solph.Model object
- **node_type** (*oemof.solph class*) – Specifies the type of the node for that inputs are selected
- **droplevel** (*list*)

Notes

from oemof import solph from oemof.outputlib import views

solve oemof solph model 'm' # Then collect node weights views.node_input_by_type(m.results(), node_type=solph.Sink)

oemof.solph.views.**node_output_by_type**(*results*, *node_type*, *droplevel=None*)
Gets all outputs for all nodes of the type *node_type* and returns a dataframe.

Parameters

- **results** (*dict*) – A result dictionary from a solved oemof.solph.Model object
- **node_type** (*oemof.solph class*) – Specifies the type of the node for that outputs are selected
- **droplevel** (*list*)

Notes

import oemof.solph as solph from oemof.outputlib import views

solve oemof solph model 'm' # Then collect node weights views.node_output_by_type(m.results(), node_type=solph.Transformer)

oemof.solph.views.**node_weight_by_type**(*results*, *node_type*)
Extracts node weights (if exist) of all components of the specified *node_type*.

Node weight are endogenous optimzation variables associated with the node and not the edge between two node, foxample the variable representing the storage level.

Parameters

- **results** (*dict*) – A result dictionary from a solved oemof.solph.Model object
- **node_type** (*oemof.solph class*) – Specifies the type for which node weights should be collected

Example

from oemof.outputlib import views

solve oemof model 'm' # Then collect node weights views.node_weight_by_type(m.results(), node_type=solph.GenericStorage)

CHAPTER 4

Contributing

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

4.1 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.

4.2 Documentation improvements

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

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/oemof/oemof-solph/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 :)

4.4 Development

To set up *oemof-solph* for local development:

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

```
git clone git@github.com:oemof/oemof-solph.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.

4.4.1 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`.

4.4.2 Tests

To run the all tests run:

```
tox
```

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

[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 ...

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

4.4.3 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
```

CHAPTER 5

Authors

–**alphabetic order**–

(see full list on github)

- Birgit Schachler
- Brian Michael Lancien
- Caroline Möller
- Caterina Köhl
- Clemens Wingenbach
- Cord Kaldemeyer
- Daniel Rank
- Elisa Gaudchau
- Elisa Papdis
- Fabian Büllesbach
- Francesco Witte
- Guido Plessmann
- Hendrik Hyskens
- Jakob Wolf
- Jann Launer
- Jens-Olaf Delfs
- Johannes Röder
- Jonathan Amme
- Julian Endres
- Lluis Millet

- Martin Soethe
- Patrik Schönfeldt
- Simon Hilpert
- Stephan Günther
- Uwe Krien

CHAPTER 6

Changelog

These are new features and improvements of note in each release

Releases

6.1 v0.4.1 (June 24, 2020)

6.1.1 Bug fixes

- Fixed incompatibility with recent Pyomo release (5.7)

6.1.2 Known issues

- **Results of one-time-step optimisation counterintuitive** If an optimisation with one time-step is performed, at the processing of the results, the scalars of the results is stored in the dict of the sequences. (See Issue #693)

6.1.3 Contributors

- Uwe Krien
- Patrik Schönfeldt

6.2 v0.4.0 (June 6, 2020)

6.2.1 API changes

- **New package name** For installation via pypi use `pip install oemof.solph`.
- **Change the import of oemof-solph due to unbundling oemof solph** The import statements have changed, for example `from outputlib.views import processing` -> `from oemof.solph import processing`. There are further changes for the modules `views`, `helpers`, `economics`, `logger`, `network`.
- **Rename GenericStorage attributes** The attribute `capacity` of the *GenericStorage* describing the current absolute stored energy/material/water etc. has been renamed to `storage_content`. In the *GenericStorageBlock* and *GenericInvestmentStorageBlock*, the attribute `init_cap` has been renamed `init_content`. This change is intended to avoid confusion with `nominal_storage_capacity` or capacity in terms of installed capacity.
- **Rename the flow attribute ‘‘actual_value‘‘ to ‘‘fix‘‘ and remove ‘‘fixed‘‘**

6.2.2 New features

- **Allows having a non equidistant timeindex** By adding the calculate_timeincrement function to tools/helpers.py a non equidistant timeincrement can be calculated. The EnergySystem will now be defined by the timeindex and the calculated timeincrement.
- **Allows non-convex investments for flows and storages.** With this feature, fix investment costs, which do not dependent on the nominal capacity, can be considered.

- **Add user warnings for debugging.** A UserWarning is raised for untypical uses even though this kind of usage is valid if you really know what you are doing. This will help users to debug their code but can be turned of for experienced users.
- **Add fixed losses to GenericStorage** *~oemof.solph.GenericStorage* can now have `fixed_losses`, that are independent from storage content.

6.2.3 New components/constraints

- **Allows a generic limit for attribute weighted investment flows** InvestmentFlow, which share other limited resources (e.g. space), can be considered.
- **Allow to limit count of concurrently active flows in group of flows** Flows have to be NonConvex, the limit can be an upper or lower one.
- **New constraint *shared_limit*** *Shared limit* allows to restrict the weighted sum of arbitrary variables to a corridor. This can be used, e.g. to model shared space is used to store wood pallets and logs with their respective energy density.

6.2.4 Documentation

- **Restructure and clean-up documentation due to the unbundling**
- **Improved documentation of ExtractionTurbineCHP**

6.2.5 Known issues

- **Results of one-time-step optimisation counterintuitive** If an optimisation with one time-step is performed, at the processing of the results, the scalars of the results is stored in the dict of the sequences. (See Issue #693)

6.2.6 Testing

- **Use `tox` for testing** Now, pep8 tests and build of documentation are tested.
- **Skip github link checks when testing locally**

6.2.7 Other changes

- **Redefine `loss_rate` of *GenericStorage*** The `loss_rate` of *~oemof.solph.components.GenericStorage* is now defined per time increment.
- **Change parameters' data type in the docstrings** The parameters' data type is changed from *numeric (sequence or scalar)* to *numeric (iterable or scalar)* (Issue #673).
- **Add python 3.8 support, remove python 3.5 support**

6.2.8 Contributors

- Caterina Köhl
- Jonathan Amme

- Uwe Krien
- Johannes Röder
- Jann Launer
- Daniel Rank
- Patrik Schönfeldt
- Stephan Günther

6.3 v0.3.2 (November 29, 2019)

6.3.1 New features

- Allow generic limits for integral over weighted flows. (This is a generalised version of <solph.constraints.emission_limit>.)
- Allow time-dependent weights for integrated weighted limit.

6.3.2 New components

- Custom component: *~oemof.solph.custom.SinkDSM*. Demand Side Management component that allows to represent flexibile demand. How the component is used is shown in *SinkDSM (custom)*.

6.3.3 Documentation

- Revision of the outputlib documentation.

6.3.4 Other changes

- The license hase been changed from GPLv3 to the MIT license
- The BaseModel has been revised (test, docstring, warnings, internal naming) (PR #605)
- Style revision to meet pep8 and other pep rules.

6.3.5 Contributors

- Guido Plessmann
- Johannes Röder
- Julian Endres
- Patrik Schönfeldt
- Uwe Krien

6.4 v0.3.1 (June 11, 2019)

6.4.1 Other changes

- The API of the GenericStorage changed. Due to the open structure of solph the old parameters are still accepted. Therefore users may not notice that the default value is used instead of the given value after an update from v0.2.x to v0.3.x. With this version an error is raised. We work on a structure to avoid such problems in the future.

6.4.2 Contributors

- Patrik Schönfeldt
- Stephan Günther
- Uwe Krien

6.5 v0.3.0 (June 5, 2019)

6.5.1 API changes

- The `param_results` function does not exist anymore. It has been renamed to `parameter_as_dict` (Issue #537).
- The storage API has been revised. Please check the API documentation for all details.
- The OffsetTransformer is now a regular oemof.solph component. It has been tested and the documentation has been improved. So it has been move from *custom* to *components*. Use oemof.solph.components.OffsetTransformer (Issue #522).

6.5.2 New features

- Now it is possible to model just one time step. This is important for time step based models and all other models with an outer loop (Issue #519).
- The storage can be used unbalanced now, which means that the level at the end could be different to the level at the beginning of the modeled time period. See the storage documentation for more details.
- *NonConvexFlow <oemof.solph.blocks.NonConvexFlow>* can now have *activity_costs*, *maximum_startups*, and *maximum_shutdowns*. This helps, to model e.g. terms of maintannace contracts for small CHP plants.
- Namedtuples and tuples as labels work now without problems. This makes it much easier to find objects and results in large energy systems (Issue #507).
- Groups are now fully lazy. This means that groups are only computed when they are accessed. Previously, whenever nodes where added to an energy system, groups where computed for all but the most recently added node. This node was then only grouped upon addition of another node or upon access of the groups property.
- There is now an explicit *Edge <oemof.network.Edge>* class. This means that an energy system now consists of *Buses <oemof.network.Bus>*, *Components <oemof.network.Component>* and *Edges <oemof.network.Edge>*. For implementation reasons, *Edges <oemof.network.Edge>* are still *Nodes <oemof.network.Node>*. If you know a bit of graph theory and this seems strange to you, think of these *Edges <oemof.network.Edge>* as classical graph theoretic edges, reified as nodes with an in- and outdegree of one.

- *Energy systems <oemof.energy_system.EnergySystem>* now support blinker signals. The first supported signal gets emitted, whenever a *node <oemof.network.node>* is *added <oemof.energy_system.EnergySystem.add>* to an *energy system <oemof.energy_system.EnergySystem>*. (blinker)

6.5.3 Documentation

- The template for docstrings with equations (docstring of block classes) has been improved.
- A lot of improvements on the documentation

6.5.4 Bug fixes

- The timeincrement attribute of the model is not set to one anymore. In earlier versions the timeincrement was set to one by default. This was a problem if a wrong time index was passed. In that case the timeincrement was set to one without a warning. Now an error is raised if no timeincrement or valid time index is found (Issue #549).

6.5.5 Testing

- Automatic test coverage control was implemented. Now a PR will not be accepted if it decreases the test coverage.
- Test coverage was increased to over 90%. A badge was added to the oemof github page that shows the actual test coverage.
- Test coverage on the *groupings <oemof.groupings>* and *network <oemof.network>* modules has significantly increased. These modules where historically very weakly tested and are now approaching 90% and 95% respectively with more tests being planned.

6.5.6 Contributors

(alphabetical order)

- ajimenezUCLA
- FranziPl
- Johannes Röder
- Jakob Wolf
- Jann Launer
- Lluis Millet
- Patrik Schönfeldt
- Simon Hilpert
- Stephan Günther
- Uwe Krien

6.6 v0.2.3 (November 21, 2018)

6.6.1 Bug fixes

- Some functions did not work with tuples as labels. It has been fixed for the ExtractionTurbineCHP, the graph module and the parameter_as_dict function. (Issue #507)

6.6.2 Contributors

- Simon Hilpert
- Stephan Günther
- Uwe Krien

6.7 v0.2.2 (July 1, 2018)

6.7.1 API changes

- The storage API has been revised, even though it is still possible to use the old API. In that case a warning is raised (Issue #491).
- The newly introduced parm_results are not results and therefore renamed to parameter_as_dict. The old name is still valid but raises a warning.

6.7.2 New features

- We added a new attribute *existing* to the *solph.options.Investement* class. It will now be possible to run investment optimization based on already installed capacity of a component. Take a look on *Using the investment mode* for usage of this option. (Issue #489).
- Investement variables for the capacity and the flows are now decoupled to enable more flexibility. It is possible to couple the flows to the capacity, the flows to itself or to not couple anything. The newly added attributes *invest_relation_input_output*, *invest_relation_input_capacity* and *invest_relation_output_capacity* replace the existing attributes *nominal_input_capacity_ratio* and *nominal_input_capacity_ratio* for the investment mode. In case of the dispatch mode one should use the *nominal_value* of the Flow classes. The attributes *nominal_input_capacity_ratio* and *nominal_input_capacity_ratio* will be removed in v0.3.0. Please adapt your application to avoid problems in the future (Issue #480).
- We now have experimental support for deserializing an energy system from a tabular data package. Since we have to extend the datapackage format a bit, the specification is not yet finalized and documentation as well as tests range from sparse to nonexistent. But anyone who wishes to help with the code is welcome to check it out in the *datapackage <oemof.tools.datapackage>* module.

6.7.3 New components

6.7.4 Documentation

- The documentation of the storage storage component has been improved.
- The documentation of the Extraction Turbine has been improved.

6.7.5 Known issues

- It is not possible to model one time step with oemof.solph. You have to model at least two timesteps (Issue #306). Please leave a comment if this bug affects you.

6.7.6 Bug fixes

- Fix file extension check to dump a graph correctly as .graphml-file
- The full constraint set of the ExtractionTurbineCHP class was only build for one object. If more than one object was present the input/output constraint was missing. This lead to wrong results.
- In the solph constraints module the emission constraint did not include the timeincrement from the model which has now be fixed.
- The parameter_as_dict (former: param_results) do work with the views functions now (Issue #494).

6.7.7 Testing

- The test coverage has been increased (>80%). oemof has experimental areas to test new functions. These functions are marked as experimental and will not be tested. Therefore the real coverage is even higher.

6.7.8 Other changes

- Subclasses of *Node <oemof.network.Node>* are no longer optimized using __slots__. The abstract parent class still defines *__slots__ <oemof.network.Node.__slots__>* so that custom subclasses still have the option of using it.

6.7.9 Contributors

- Fabian Büllesbach
- Guido Plessmann
- Simon Hilpert
- Stephan Günther
- Uwe Krien

6.8 v0.2.1 (March 19, 2018)

6.8.1 API changes

- The function create_nx_graph only takes an energysystem as argument, not a solph model. As it is not a major release you can still pass a Model but you should adapt your application as soon as possible. (Issue #439)

6.8.2 New features

- It is now possible determine minimum up and downtimes for nonconvex flows. Check the oemof_examples repository for an exemplary usage.
- Startup and shutdown costs can now be defined time-dependent.
- The graph module has been revised. (Issue #439)
  - You can now store a graph to disc as *.graphml* file to open it in yEd with labels.
  - You can add weight to edges.
  - Labels are attached to the nodes.
- Two functions *get_node_by_name* and *filter_nodes* have been added that allow to get specified nodes or nodes of one kind from the results dictionary. (Issue #426)
- A new function *param_results()* collects all parameters of nodes and flows in a dictionary similar to the *results* dictionary. (Issue #445)
- In *outputlib.views.node()*, an option for multiindex dataframe has been added.

6.8.3 Documentation

- Some small fixes and corrected typos.

6.8.4 Known issues

- It is not possible to model one time step with oemof.solph. You have to model at least two timesteps (Issue #306). Please leave a comment if this bug affects you.

6.8.5 Bug fixes

- Shutdown costs for nonconvex flows are now accounted within the objective which was not the case before due to a naming error.
- Console script *oemof_test_installation* has been fixed. (Issue #452)
- Adapt solph to API change in the Pyomo package.
- Deserializing a *Node <oemof.network.Node>* leads to an object which was no longer serializable. This is fixed now. *Node <oemof.network.Node>* instances should be able to be dumped and restored an arbitraty amount of times.
- Adding timesteps to index of constraint for component el-line fixes an issue with pyomo.

6.8.6 Testing

- New console script *test_oemof* has been added (experimental). (Issue #453)

6.8.7 Other changes

- Internal change: Unnecessary list extensions while creating a model are avoided, which leads to a decrease in runtime. (Issue #438)
- The negative/positive gradient attributes are dictionaries. In the constructor they moved from sequences to a new *dictionaries* argument. (Issue #437)
- License agreement was adapted according to the reuse project (Issue #442)
- Code of conduct was added. (Issue #440)
- Version of required packages is now limited to the most actual version (Issue #464)

6.8.8 Contributors

- Cord Kaldemeyer
- Jann Launer
- Simon Hilpert
- Stephan Günther
- Uwe Krien

6.9 v0.2.0 (January 12, 2018)

6.9.1 API changes

- The *NodesFromCSV* has been removed from the code base. An alternative excel (spreadsheet) reader is provided in the newly created excel example in the oemof_examples repository.
- LinearTransformer and LinearN1Transformer classes are now combined within one Transformer class. The new class has n inputs and n outputs. Please note that the definition of the conversion factors (for N1) has changed. See the new docstring of *~oemof.solph.network.Transformer* class to avoid errors (Issue #351).
- The *oemof.solph.network.Storage* class has been renamed and moved to *oemof.solph.components.GenericStorage*.
- As the example section has been moved to a new repository the *oemof_example* command was removed. Use *oemof_installation_test* to check your installation and the installed solvers.
- OperationalModel has been renamed to Model. The *es* parameter was renamed to *energysystem* parameter.
- *Nodes <oemof.network.Node>* are no longer automatically added to the most recently created *energy system <oemof.energy_system.EnergySystem>*. You can still restore the old automatic registration by manually assigning an *energy system <oemof.energy_system.EnergySystem>* to *Node.registry <oemof.network.Node.registry>*. On the other hand you can still explicitly *add <oemof.energy_system.EnergySystem.add> nodes <oemof.network.Node>* to an *energy system <oemof.energy_system.EnergySystem>*. This option has been made a bit more feature rich by the way, but see below for more on this. Also check the oemof_examples repository for more information on the usage.
- The *fixed_costs* attribute of the *nodes <oemof.solph.network.Flow>* has been removed. See (Issue #407) for more information.
- The classes *DataFramePlot <outputlib.DataFramePlot>* and *ResultsDataFrame <outputlib.ResultsDataFrame>* have been removed due to the redesign of the outputlib module.

6.9.2 New features

- A new oemof_examples repository with some new examples was created.
- A new outputlib module has been created to provide a convenient data structure for optimization results and enable quick analyses. All decision variables of a Node are now collected automatically which enables an easier development of custom components. See the revised *Handling Results* documentation for more details or have a look at the oemof_examples repository for information on the usage. Keep your eyes open, some new functions will come soon that make the processing of the results easier. See the actual pull request or issues for detailed information.
- The transformer class can now be used with n inputs and n outputs (*~oemof.solph.network.Transformer*)
- A new module with useful additional constraints were created with these constraints global emission or investment limits can be set. Furthermore it is possible to connect investment variables. Please add your own additional constraints or let us know what is needed in the future.
- A module to create a networkx graph from your energy system or your optimisation model was added. You can use networkx to plot and analyse graphs. See the graph module in the documentation of oemof-network for more information.
- It's now possible to modify a *node's <oemof.network.Node> inputs <oemof.network.Node.inputs>* and *outputs <oemof.network.Node.outputs>* by inserting and removing *nodes <oemof.network.Node>* to and from the corresponding dictionaries. *Outputs <oemof.network.Node.outputs>* where already working previously, but due to an implementation quirk, *inputs <oemof.network.Node.inputs>* did not behave as expected. This is now fixed.
- One can now explicitly *add <oemof.energy_system.EnergySystem.add> nodes <oemof.network.Node>* to an *energy system <oemof.energy_system.EnergySystem>* in bulk using * and ** syntax. For the latter case, the *values <dict.values>* of the dictionary passed in will be added.
- New components can now be added to the custom.py module. Components in this module are indicated as in a testing state. Use them with care. This lowers the entry barriers to test new components within the solph structure and find other testers.

6.9.3 New components

- The nodes *ElectricalLine <oemof.solph.custom.ElectricalLine>* and *ElectricalBus <oemof.solph.custom.ElectricalBus>* can be used for Linear Optimal Powerflow calculation based on angle formulations. These components have been added to the *solph.custom* module. Though it should work correctly, it is in a preliminary stage. Please check your results. Feedback is welcome!
- The custom component *Link <oemof.solph.custom.Link>* can now be used to model a bidirectional connection within one component. Check out the example in the oemof_examples repository.
- The component *GenericCHP <oemof.solph.components.GenericCHP>* can be used to model different CHP types such as extraction or back-pressure turbines and motoric plants. The component uses a mixed-integer linear formulation and can be adapted to different technical layouts with a high level of detail. Check out the example in the oemof_examples repository.
- The component *GenericCAES <oemof.solph.custom.GenericCAES>* can be used to model different concepts of compressed air energy storage. Technical concepts such as diabatic or adiabatic layouts can be modelled at a high level of detail. The component uses a mixed-integer linear formulation.
- The custom component *GenericOffsetTransformer <oemof.solph.custom.GenericOffsetTransformer>* can be used to model components with load ranges such as heat pumps and also uses a mixed-integer linear formulation.

6.9.4 Documentation

- Large parts of the documentation have been proofread and improved since the last developer meeting in Flensburg.
- The solph documentation has got an extra section with all existing components (*Solph components*).
- The developer documentation has been developed to lower the barriers for new developers. Furthermore, a template for pull request was created.

6.9.5 Known issues

- It is not possible to model one time step with oemof.solph. You have to model at least two timesteps (Issue #306). Please leave a comment if this bug affects you.

6.9.6 Bug fixes

- LP-file tests are now invariant against sign changes in equations, because the equations are now normalized to always have non-negative right hand sides.

6.9.7 Testing

- All known and newly created components are now tested within an independent testing environment which can be found in */tests/*.
- Other testing routines have been streamlined and reviewed and example tests have been integrated in the nosetest environment.

6.9.8 Other changes

- The plot functionalities have been removed completely from the outputlib as they are less a necessary part but more an optional tool . Basic plotting examples that show how to quickly create plots from optimization results can now be found in the oemof_examples repository. You can still find the "old" standard plots within the oemof_visio repository as they are now maintained separately.
- A user forum has been created to answer use questions.

6.9.9 Contributors

- Cord Kaldemeyer
- Jens-Olaf Delfs
- Stephan Günther
- Simon Hilpert
- Uwe Krien

6.10 v0.1.4 (March 28, 2017)

6.10.1 Bug fixes

- fix examples (issue #298)

6.10.2 Documentation

- Adapt installation guide.

6.10.3 Contributors

- Uwe Krien
- Stephan Günther

6.11 v0.1.2 (March 27, 2017)

6.11.1 New features

- Revise examples - clearer naming, cleaner code, all examples work with cbc solver (issue #238, issue #247).
- Add option to choose solver when executing the examples (issue #247).
- Add new transformer class: VariableFractionTransformer (child class of LinearTransformer). This class represents transformers with a variable fraction between its output flows. In contrast to the LinearTransformer by now it is restricted to two output flows.(issue #248)
- Add new transformer class: N1Transformer (counterpart of LinearTransformer). This class allows to have multiple inputflows that are converted into one output flow e.g. heat pumps or mixing-components.
- Allow to set addtional flow attributes inside NodesFromCSV in solph inputlib
- Add economics module to calculate investment annuities (more to come in future versions)
- Add module to store input data in multiple csv files and merge by preprocessing
- Allow to slice all information around busses via a new method of the ResultsDataFrame
- Add the option to save formatted balances around busses as single csv files via a new method of the ResultsDataFrame

6.11.2 Documentation

- Improve the installation guide.

6.11.3 Bug fixes

- Allow conversion factors as a sequence in the CSV reader

6.11.4 Other changes

- Speed up constraint-building process by removing unnecessary method call
- Clean up the code according to pep8 and pylint

6.11.5 Contributors

- Cord Kaldemeyer
- Guido Plessmann
- Uwe Krien
- Simon Hilpert
- Stephan Günther

6.12 v0.1.1 (November 2, 2016)

Hot fix release to make examples executable.

6.12.1 Bug fixes

- Fix copy of default logging.ini (issue #235)
- Add matplotlib to requirements to make examples executable after installation (issue #236)

6.12.2 Contributors

- Guido Plessmann
- Uwe Krien

6.13 v0.1.0 (November 1, 2016)

The framework provides the basis for a great range of different energy system model types, ranging from LP bottom-up (power and heat) economic dispatch models with optional investment to MILP operational unit commitment models.

With v0.1.0 we refactored oemof (not backward compatible!) to bring the implementation in line with the general concept. Hence, the API of components has changed significantly and we introduced the new 'Flow' component. Besides an extensive grouping functionality for automatic creation of constraints based on component input data the documentation has been revised.

We provide examples to show the broad range of possible applications and the frameworks flexibility.

6.13.1 API changes

- The demandlib is no longer part of the oemof package. It has its own package now: (demandlib)

6.13.2 New features

- Solph's *EnergySystem <oemof.solph.network.EnergySystem>* now automatically uses solph's *GROUPINGS <oemof.solph.groupings.GROUPINGS>* in addition to any user supplied ones. See the API documentation for more information.
- The *groupings <oemof.groupings.Grouping>* introduced in version 0.0.5 now have more features, more documentation and should generally be pretty usable:
  - They moved to their own module: *oemof.groupings* and deprecated constructs ensuring compatibility with prior versions have been removed.
  - It's possible to assign a node to multiple groups from one *Grouping <oemof.groupings.Grouping>* by returning a list of group keys from *key <oemof.groupings.Grouping.key>*.
  - If you use a non callable object as the *key <oemof.groupings.Grouping.key>* parameter to *Groupings <oemof.groupings.Grouping>*, the constructor will not make an attempt to call them, but use the object directly as a key.
  - There's now a *filter <oemof.groupings.Grouping.filter>* parameter, enabling a more concise way of filtering group contents than using *value <oemof.groupings.Grouping.value>*.

6.13.3 Documentation

- Complete revision of the documentation. We hope it is now more intuitive and easier to understand.

6.13.4 Testing

- Create a structure to use examples as system tests (issue #160)

6.13.5 Bug fixes

- Fix relative path of logger (issue #201)
- More path fixes regarding installation via pip

6.13.6 Other changes

- Travis CI will now check PR's automatically
- Examples executable from command-line (issue #227)

6.13.7 Contributors

- Stephan Günther
- Simon Hilpert
- Uwe Krien
- Guido Pleßmann
- Cord Kaldemeyer

6.14 v0.0.7 (May 4, 2016)

6.14.1 Bug fixes

- Exclude non working pyomo version

6.15 v0.0.6 (April 29, 2016)

6.15.1 New features

- It is now possible to choose whether or not the heat load profile generated with the BDEW heat load profile method should only include space heating or space heating and warm water combined. (Issue #130)
- Add possibility to change the order of the columns of a DataFrame subset. This is useful to change the order of stacked plots. (Issue #148)

6.15.2 Documentation

6.15.3 Testing

- Fix constraint tests (Issue #137)

6.15.4 Bug fixes

- Use of wrong columns in generation of SF vector in BDEW heat load profile generation (Issue #129)
- Use of wrong temperature vector in generation of h vector in BDEW heat load profile generation.

6.15.5 Other changes

6.15.6 Contributors

- Uwe Krien
- Stephan Günther
- Simon Hilpert
- Cord Kaldemeyer
- Birgit Schachler

6.16 v0.0.5 (April 1, 2016)

6.16.1 New features

- There's now a *flexible transformer <oemof.core.network.entities.components.transformers.TwoInputsOneOutput>* with two inputs and one output. (Issue #116)

- You now have the option create special groups of entities in your energy system. The feature is not yet fully implemented, but simple use cases are usable already. (Issue #60)

6.16.2 Documentation

- The documentation of the *electrical demand <oemof.demandlib.demand.electrical_demand>* class has been cleaned up.
- The API documentation now has its own section so it doesn't clutter up the main navigation sidebar so much anymore.

6.16.3 Testing

- There's now a dedicated module/suite testing solph constraints.
- This suite now has proper fixtures (i.e. *setup/teardown* methods) making them (hopefully) independent of the order in which they are run (which, previously, they where not).

6.16.4 Bug fixes

- Searching for oemof's configuration directory is now done in a platform independent manner. (Issue #122)
- Weeks no longer have more than seven days. (Issue #126)

6.16.5 Other changes

- Oemof has a new dependency: dill. It enables serialization of less common types and acts as a drop in replacement for pickle.
- Demandlib's API has been simplified.

6.16.6 Contributors

- Uwe Krien
- Stephan Günther
- Guido Pleßmann

6.17 v0.0.4 (March 03, 2016)

6.17.1 New features

- Revise the outputlib according to (issue #54)
- Add postheating device to transport energy between two buses with different temperature levels (issue #97)
- Better integration with pandas

6.17.2 Documentation

- Update developer notes

6.17.3 Testing

- Described testing procedures in developer notes
- New constraint tests for heating buses

6.17.4 Bug fixes

- Use of pyomo fast build
- Broken result-DataFrame in outputlib
- Dumping of EnergySystem

6.17.5 Other changes

- PEP8

6.17.6 Contributors

- Cord Kaldemeyer
- Uwe Krien
- Simon Hilpert
- Stephan Günther
- Clemens Wingenbach
- Elisa Papdis
- Martin Soethe
- Guido Plessmann

6.18 v0.0.3 (January 29, 2016)

6.18.1 New features

- Added a class to convert the results dictionary to a multiindex DataFrame (issue #36)
- Added a basic plot library (issue #36)
- Add logging functionalities (issue #28)
- Add *entities_from_csv* functionality for creating of entities from csv-files
- Add a time-depended upper bound for the output of a component (issue #65)
- Add fast_build functionlity for pyomo models in solph module (issue #68)

- The package is no longer named *oemof_base* but is now just called *oemof*.
- The *results* dictionary stored in the energy system now contains an attribute for the objective function and for objects which have special result attributes, those are now accessible under the object keys, too. (issue #58)

6.18.2 Documentation

- Added the Readme.rst as "Getting started" to the documentation.
- Fixed installation description (issue #38)
- Improved the developer notes.

6.18.3 Testing

- With this release we start implementing nosetests (issue #47
- Tests added to test constraints and the registration process (issue #73).

6.18.4 Bug fixes

- Fix contraints in solph
- Fix pep8

6.18.5 Other changes

6.18.6 Contributors

- Cord Kaldemeyer
- Uwe Krien
- Clemens Wingenbach
- Simon Hilpert
- Stephan Günther

6.19 v0.0.2 (December 22, 2015)

6.19.1 New features

- Adding a definition of a default oemof logger (issue #28)
- Revise the EnergySystem class according to the oemof developing meeting (issue #25)
- Add a dump and restore method to the EnergySystem class to dump/restore its attributes (issue #31)
- Functionality for minimum up- and downtime constraints (oemof.solph.linear_mixed_integer_constraints module)
- Add *relax* option to simulation class for calculation of linear relaxed mixed integer problems

- Instances of *EnergySystem <oemof.core.energy_system.EnergySystem>* now keep track of *Entities <oemof.core.network.Entity>* via the *entities <oemof.core.energy_system.EnergySystem.entities>* attribute. (issue #20)
- There's now a standard way of working with the results obtained via a call to *OptimizationModel#results <oemof.solph.optimization_model.OptimizationModel.results>*. See its documentation, the documentation of *EnergySystem#optimize <oemof.core.energy_system.EnergySystem.optimize>* and finally the discussion at issue #33 for more information.
- New class *VariableEfficiencyCHP <oemof.core.network.entities.components.transformers.VariableEfficiencyCHP>* to model combined heat and power units with variable electrical efficiency.
- New methods for *VariableEfficiencyCHP <oemof.core.network.entities.components.transformers.VariableEfficiencyCHP>* inside the solph-module:
- *MILP-constraint <oemof.solph.linear_mixed_integer_constraints.add_variable_linear_eta_relation>*
- *Linear-constraint <oemof.solph.linear_constraints.add_eta_total_chp_relation>*

6.19.2 Documentation

- missing docstrings of the core subpackage added (issue #9)
- missing figures of the meta-documentation added
- missing content in developer notes (issue #34)

6.19.3 Testing

6.19.4 Bug fixes

- now the api-docs can be read on readthedocs.org
- a storage automically calculates its maximum output/input if the capacity and the c-rate is given (issue #27)
- Fix error in accessing dual variables in oemof.solph.postprocessing

6.19.5 Other changes

6.19.6 Contributors

- Uwe Krien
- Simon Hilpert
- Cord Kaldemeyer
- Guido Pleßmann
- Stephan Günther

6.20 v0.0.1 (November 25, 2015)

First release by the oemof developing group.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

P

R

S

T