oemof Documentation

Release

oemof-Team

August 15, 2016

Contents

Contents:

CHAPTER 1

Getting started

Oemof stands for "Open Energy System Modelling Framework" and provides a free, open source and clearly documented model to analyse energy supply systems. It is developed in Python and designed as a framework with a modular structure containing several packages which communicate through well defined interfaces.

With oemof we provide base packages for energy system modelling and optimisation.

1.1 Documentation

Full documentation can be found at http://oemof.readthedocs.org.

1.2 Installing oemof

```
sudo pip3 install oemof
```

1.3 Developing oemof

We highly encourage you to contribute to further development of oemof. If you want to collaborate install the developer version as described below and pay attention to the developer notes described in the documentation: http://oemof.readthedocs.org/en/latest/developer_notes.html

To install the developer version two steps are necessary:

```
git clone git@github.com:oemof/oemof.git
sudo pip3 install -e /path/to/the/repository
```

See http://oemof.readthedocs.org/en/latest/installation_and_setup.html for further information on installation and setup.

See the developer version of the full documentation at: http://oemof.readthedocs.org/en/latest/.

1.4 Further packages within oemof

Feedinlib and oemof.db are part of the oemof framework. They can be used to create energy system models but are not a must.

1.5 Examples

The linkage of specific modules of the various packages is called an application (app) and depicts for example a concrete energy system model.

There is one executable example energy system in Storage optimization.

Further example apps in development can be found in Development examples.

1.6 License

CHAPTER 2

Overview

This overview was developed to make oemof easy to use and develop. It describes general ideas and structures of oemof and its modules.

2.1 The idea of an open framework

The Open Energy System Modeling Framework has been developed for the modeling and analysis of energy supply systems considering power and heat as well as prospectively mobility. Energy system models often do not have publicly accessible source code and freely available data and are poorly documented. The missing transparency slows down the scientific discussion on model quality with regard to certain problems such as grid extension. Besides, energy system models are often developed for a certain application and cannot be adjusted (or only with great effort) to other requirements.

The Center for Sustainable Energy Systems (ZNES) together with the Reiner Lemoine Institute (RLI) in Berlin and the Otto-von-Guericke-University of Magdeburg (OVGU) are developing an Open Energy System Modeling Framework (oemof) that addresses these problems by offering a free, open and clearly documented framework for energy system modeling. This transparent approach allows a sound scientific discourse on the underlying models and data. In this way the assessment of quality and significance of undertaken analyses is improved. Moreover, the modular composition of the framework supports the adjustment to a large number of application purposes. The open source approach allows a collaborative development of the framework that offers several advantages:

- **Synergies** - By developing collaboratively synergies between the participating institutes can be utilized.
- **Debugging** - Through the input of a larger group of users and developers bugs are identified and fixed at an earlier stage.
- **Advancement** - The oemof-based application profits from further development of the framework.

2.1.1 Open Energy System Modeling Framework (oemof)

oemof is programmed in Python and uses several Python packages for scientific applications (e.g. mathematical optimisation, network analysis, data analyses), optionally in combination with a PostgreSQL/PostGIS Database. It offers a toolbox of various functionalities needed to build energy system models in high temporal and spatial resolution. For instance, the wind energy feed-in in a model region based on weather data can be modeled, the CO2-minimal operation of biomass power plants can be calculated or the future energy supply of Europe can be simulated.

The framework consists of packages. For the communication between these packages interfaces are provided. A package again consists of modules that handle a defined task. A linkage of specific modules of the various packages is in oemof called an application (app) and depicts for example a concrete energy system model. The following image shows the underlying concept.

Besides other applications the apps “renpass-gis” and “reegis” are currently developed within the framework. “renpass-gis” enables the simulation of a future European energy system with a high spatial and temporal resolution. Different expansion pathways of conventional power plants, renewable energies and net infrastructure can be considered. The app “reegis” provides a simulation of a regional heat and power supply system. These two examples show that the modular approach of the framework allows applications with very different objectives.

2.1.2 An energy system within oemof

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

An *entity* is either a *bus* or a *component*. A bus is always connected with one or several components and characterised by an unique identifier (electricity, gas, heat). Components take resources from or feed resources to buses. Transfers from buses are inputs of components, transfers to buses are ouputs of components.

Components are likewise always connected with one or several buses. Based on their characteristics they are divided into several sub types. *Transformers* have input and output, e.g. a gas turbine takes from a bus of type ‘gas’ and feeds into a bus of type ‘electricity’. With additional information like parameters and transfer functions input and output can be specified. Using the example of a gas turbine the resource consumption (input) is related to the provided end energy (output) by means of an efficiency factor. A *sink* has only an input but no output. With *sink* consumers like households can be modeled. A *source* has exactly one output but no input. Thus for example, wind energy and photovoltaic plants can be modeled. Components of type *transport* have like transformers input and output. However, corresponding buses are always of the same type, e.g. electricity. With components of type transport transmission lines can be modeled for example.

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

Besides the use of the basic components one has the possibility to develop more specified components on the base of the basic components. The following figure illustrates the setup of a simple energy system and the basic structure explained before.

2.1.3 Mathematical description (generic formulation as graph without timesteps)

Entities are connected in such a way that buses are only connected to components and vice versa. In this way the energy system can be interpreted as a bipartite graph. In this graph the entities represent vertices. The inputs and the ouputs can be interpreted as directed edges. For every edge in this graph there will be a value which we define as the weight of the edge.

Set of entities E as a union of sets of buses (B), transformers(F), sources (O), sinks (I) and transports (P) respectively, which are the vertices:

$$E := \{E_B, E_F, E_O, E_I, E_P\}$$

Set of Components:

$$E_C := E \setminus E_B$$

Set of directed edges...:

$$\vec{E} := \{(e_i, e_j), ...\}$$

Function f as "Uebertragunsfunktion" for each component used in constraints:

$$f(I_e, O_e) \leq \vec{0}, \quad \forall e \in E_C$$

I_e and O_e as subsets of E:

$$I_e := \{i \in E | (i, e) \in \vec{E}\}$$
$$O_e := \{o \in E | (e, o) \in \vec{E}\}$$

And additional constraint for outflow o and inflow i for each edge:

$$o_{e_1} - i_{e_2} = 0, \quad \forall (e_1, e_2) \in \vec{E}$$

2.1.4 Example

An example of a simple energy system shows the usage of the entities for real world representations.

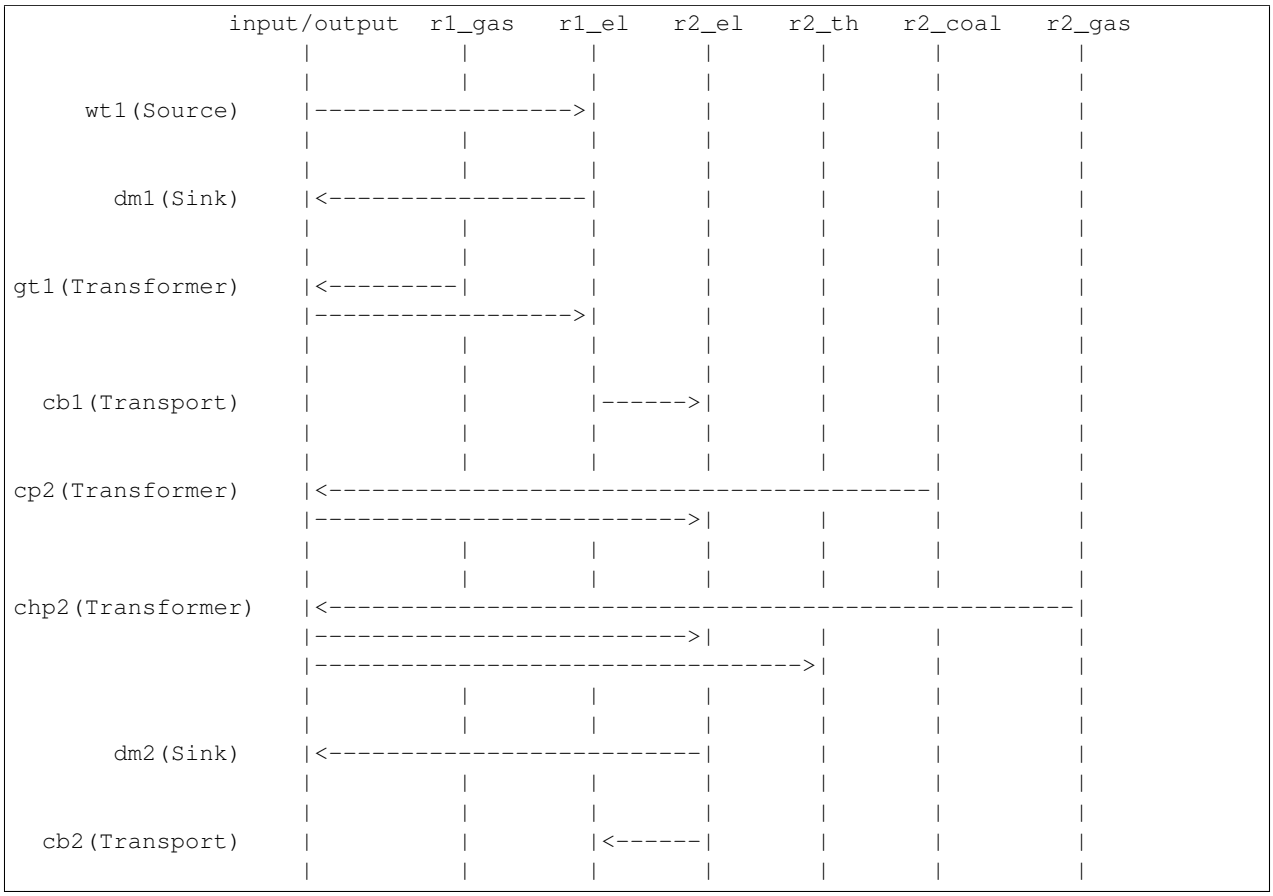
Region1:

components: wind turbine (wt1), electrical demand (dm1), gas turbine (gt1), cable to region2 (cb1) busses: gas pipeline (r1_gas), electrical grid (r1_el)

Region2:

components: coal plant (cp2), chp plant (chp2), electrical demand (dm2), cable to region2 (cb2), p2g-facility (ptg2) busses: electrical grid (r2_el), local heat network (r2_th), coal reservoir (r2_coal), gas pipeline (r2_gas)

In oemof this would look as follows:



```
                    |          |        |       |       |       |          |
ptg2(Transformer)   |<-------------------------|       |       |          |
                    |-------------------------------------------------->|
```

2.1.5 Classes and packages

All energy system entities (busses and components) are represented in a class hierarchy that can be easily extended. These classes form the basis for so so-called framework packages, that operate on top of them.

The framework consists of various packages that provide different functionalities. Currently, there are three modules but in future further extensions will be made.

oemof's current packages:

- *feedinlib* generates wind and solar feedin timeseries for different plants and geographical locations
- *demandlib* generates electrical and thermal demands for different objects
- *solph* creates and solves a (mixed-integer) linear optimization problem for a given energy system

All packages may interact with each other but can also be used stand-alone. A detailed description can be found in the following sections.

2.2 Documentation

The framework is documented on three different levels:

- Code commenting
- Code documentation
- General documentation

2.2.1 Code commenting

Code comments are block and inline comments in the source code. They can help to understand the code and should be utilized "as much as necessary, as little as possible". When writing comments follow the PEP 0008 style guide: https://www.python.org/dev/peps/pep-0008/#comments.

2.2.2 Code documentation

Code documentation is done via documentation strings, a.k.a. "docstrings", and used for all public modules, functions, classes, and methods.

We are using the numpydoc extension of sphinx and thus the numpydoc docstring notation. PEP 0257 (https://www.python.org/dev/peps/pep-0257/) lays down a few, very general conventions for docstrings. Following is an example of a numpydoc docstring:

```
def docstring():
    r"""A one-line summary that does not use variable names or the
    function name.

    Several sentences providing an extended description. Refer to
    variables using back-ticks, e.g. `var`.
```

```
    Parameters
    ----------
    var1 : array_like
        Array_like means all those objects -- lists, nested lists, etc. --
        that can be converted to an array.  We can also refer to
        variables like `var1`.
    var2 : int
        The type above can either refer to an actual Python type
        (e.g. ``int``), or describe the type of the variable in more
        detail, e.g. ``(N,) ndarray`` or ``array_like``.
    Long_variable_name : {'hi', 'ho'}, optional
        Choices in brackets, default first when optional.
    main_dt : dictionary
        Main dictionary as described below [1]_
    prob : pulp.lp-problem
        LP-Problem-Variable, which contains the linear problem [2]_

    Returns
    -------
    type
        Explanation of anonymous return value of type ``type``.
    describe : type
        Explanation of return value named `describe`.
    out : type
        Explanation of `out`.
    prob : pulp.lp-problem
        LP-Problem-Variable, which contains the extended linear problem [2]_

    Other Parameters
    ----------------
    only_seldom_used_keywords : type
        Explanation
    common_parameters_listed_above : type
        Explanation
    Timesteps [t] : main_dt['timesteps']
        np-array with the timesteps according to the timeseries
    Regions [r] : main_dt['energy_system']['regions']
        See: solph.extenddc [4]_
    Electric demand : main_dt['timeseries']['demand'][r]['lele'][t]
        r = region, t = timesteps
    main_dt['energy_system'] : dict-branch with lists of components
        Definition of the 'energy_system' see: :py:mod:`solph.extenddc`
    main_dt['lp'] : dict-branch with all lp-variables
        Definition of lp-variables see: :py:mod:`solph.lp_definition`

    Raises
    ------
    BadException
        Because you shouldn't have done that.

    See Also
    --------
    otherfunc : relationship (optional)
    newfunc : Relationship (optional), which could be fairly long, in which
              case the line wraps here.
    thirdfunc, fourthfunc, fifthfunc
    solph.main_model.create_model_equations : Blubber
```

```
    Notes
    -----
    Notes about the implementation algorithm (if needed).

    This can have multiple paragraphs.

    You may include some math:

    .. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}

    And even use a greek symbol like :math:`omega` inline.

    References
    ----------
    Cite the relevant literature, e.g. [3]_.  You may also cite these
    references in the notes section above.

    .. [1] Link to the description of the main_dt for solph.
    .. [2] `PuLP <https://code.google.com/p/pulp-or/>`_, PuLP Documentation.
    .. [3] O. McNoleg, "The integration of GIS, remote sensing,
       expert systems and adaptive co-kriging for environmental habitat
       modelling of the Highland Haggis using object-oriented, fuzzy-logic
       and neural-network techniques," Computers & Geosciences, vol. 22,
       pp. 585-588, 1996.

    Examples
    --------
    These are written in doctest format, and should illustrate how to
    use the function.

    >>> a=[1,2,3]
    >>> print [x + 3 for x in a]
    [4, 5, 6]
    >>> print "a\n\nb"
    a
    b

    """
```

2.2.3 General documentation

The general implementation-independent documentation such as installation guide, flow charts, and mathematical models is done via ReStructuredText (rst). The files can be found in the folder */oemof/doc*. For further information on restructured text see: http://docutils.sourceforge.net/rst.html.

2.3 oemof *base classes*

Currently, oemof provides the following classes. The first three levels represent the basic components to model energy systems. Additional subclasses can be defined underneath.

- Entity
  - Bus
  - Component

    * Sink
        - Simple
    * Source
        - Commodity
        - DispatchSource
        - FixedSource
    * Transformer
        - Simple
        - CHP
        - SimpleExtractionCHP
        - Storage
    * Transport
        - Simple

More information on the functionality of the respective classes can be found in their ApiDocs [Link!].

2.4 The *feedinlib* package

The modelling library feedinlib is currently in a development stage. Using feedinlib energy production timeseries of several energy plants can be created. Focus is on fluctuating renewable energies like wind energy and photovoltaics. The output timeseries can be input for the components of the energy system and therefore incorporated in the optimization within the modelling library solph. However, a stand-alone usage of feedinlib is also intended.

Clone or fork the 'feedinlib' from github and use it within your project. Don't forget to play back your fixes and improvements. We are pleased to get your feedback.

2.5 The *demandlib* package

The demand timeseries modeling library is designed to generated synthetic demand timeseries. It founds on methodolody of *Standardlastprofile* defined by the *Bundesverband der Energie- und Wasserwirtschaft (BDEW)*. .. Load profiles for the electricity sector founds on data of .. EWE.

2.6 The *solph* package

The solph module of oemof allows to create and solve linear (and mixed-integer) optimization problems. The optimization problem is build based on a energy system defined via oemof-entities. These entities are instances of oemof base classes (e. g. buses or components). For the definition of variables, constraints and an objective function as well as for communication with solvers etc. the python packages Pyomo is used.

2.6.1 Structure of solph

At its core solph has a class called *OptimizationModel()* which is a child of the pyomo class *ConcreteModel()*. This class contains different methods. An important type of methods are so called *assembler* methods. These methods correspond exactly to one oemof-base-class. For example the *transfomer.Simple()* class of oemof will have a associated method called simple_transformer_assembler(). This method groups all necessary constraints to model a simple transformer. The constraints expressions are defined in extra module (*linear_constraints.py*, *linear_mixed_integer_constraints.py*). All necessary constraints related with variables are defined in *variables.py*.

Constructor

The whole pyomo model is build when instantiating the optimization model. This is why the constructor of the *OptimizationModel()* class plays an important role.

The general procedure is as basically follows:

1. Set some options
2. Create all necessary optimization variables
3. Loop trough all entities and group existing objects by class
4. Call the associated *assembler* method for every **group** of objects. This builds constraints to model components.
5. Build the bus constraints with bus *assembler*.
6. Build objective *assembler*.

Assembler methods

The *assembler* methods can be specified in two different ways. Firstly, functions from the solph-library called *linear_constraints.py* can be used to add constraints to the *assembler*. Secondly, *assembler* methods can use other *assembler* methods and then be extended by functions from the library. The same holds for the objective *assembler*. The objective function uses pre-defined objectives from the solph-library called *objectives.py*. These pre-defined objectives are build by the use of objective expressions defined in *objective_expressions*. Different objectives for optimization models can be selected by setting the option *objective_types* inside the *objective_assembler* method.

If necessary, the two libraries used be *assemlber* methods can be extended and used in methods of *OptimizationModel()* afterwards.

Solve and other

Moreover, the *OptimizationModel()* class contains a method for solving the optimization model.

2.6.2 Postprocessing of results

To extract values from the optimization problem variables their exist a postprossing module containing different functions. Results can be written back to the oemof-objects or to excel-spreadsheets.

CHAPTER 3

Installation and setup

3.1 Introduction

Following you find guidelines for the installation process for different operation systems.

3.2 Linux

3.2.1 If you have Python 3 installed

As oemof is designed as a Phyton-module it is mandatory to have Python 3 installed. If you already have Python 3 you can install oemof by using pip. Run the following code in your terminal:

```
sudo pip3 install oemof
```

If you do not yet have pip installed, see section “Required Python packages” below for further help.

3.2.2 If you do not have Python 3 installed

There are different ways to install Python on your system. One way is to install Python 3 through the Linux repositories. If you are using Ubuntu try executing the following code in your terminal:

```
sudo apt-get install python3
```

Otherwise you can download different versions of Python via https://www.python.org/downloads/.

3.2.3 Required Python packages

To be able to install additional Python packages an installer program is needed. The preferred installer is pip which is included by default in the installation of Python 3.4 and later versions. To install pip for earlier Python versions try executing the following code in your terminal:

```
sudo apt-get install python3-pip
```

For further information refer to https://packaging.python.org/en/latest/installing/#install-pip-setuptools-and-wheel.

In order to install a package using pip execute the following and substitute package_name by the desired package:

```
sudo pip3 install package_name
```

For further information on how to install Python modules check out https://docs.python.org/3/installing/index.html. The following table shows which packages are needed for which oemof library:

Packages	solph	core	demandlib	example storage_optimization
matplotlib	x	x		
pandas	x	x	x	x
numpy	x	x	x	x
pyomo	x			
descartes		x		
shapely		x		

3.2.4 Solver

In order to use solph you need to install a solver. There are various commercial and open-source solvers that can be used with oemof. The recommended open-source solver is Cbc (Coin-or branch and cut). See the CBC wiki for download and installation instructions: https://projects.coin-or.org/CoinBinary.

3.3 Windows

3.3.1 If you have Python 3 installed

As oemof is designed as a Phyton-module it is mandatory to have Python 3 installed. If you already have Python 3 you can install oemof by using pip. Run the following code in your command window:

```
pip3 install oemof
```

If you do not yet have pip installed, see section "Required Python packages" below for further help.

3.3.2 If you do not have Python 3 installed

To install python3 download the winpython version suitable for your system from http://winpython.sourceforge.net/ and follow the installation instructions.

Next, set the system's PATH variable to include directories that include python components and packages. To do this go to *My Computer -> Properties -> Advanced System Settings -> Environment Variables*. In the User Variables section, edit or create the PATH statement to include the following (make sure to replace the path to winpython by your own path):

```
C:\winpython;C:\winpython\python\Lib\site-packages\;C:\windpython\python\Scripts\;
```

3.3.3 Required Python packages

To be able to install additional Python packages an installer program is needed. The preferred installer is pip which is included in the winpython download. If you do not have pip installed see here: https://packaging.python.org/en/latest/installing/#install-pip-setuptools-and-wheel.

In order to install a package using pip execute the following and substitute package_name by the desired package:

```
pip3 install package_name
```

For further information on how to install Python modules check out https://docs.python.org/3/installing/ The following table shows which packages are needed for which oemof library:

Packages	solph	core	demandlib	example storage_optimization
matplotlib	x	x		
pandas	x	x	x	x
numpy	x	x	x	x
pyomo	x			
descartes		x		
shapely		x		

3.3.4 Solver

In order to use solph you need to install a solver. There are various commercial and open-source solvers that can be used with oemof. The recommended open-source solver is Cbc (Coin-or branch and cut). See the CBC wiki for download and installation instructions: https://projects.coin-or.org/CoinBinary.

3.4 Mac OS

Installation guidelines for Mac OS will follow.

You can download python here: https://www.python.org/downloads/mac-osx/. For information on the installation process and on how to install python packages see here: https://docs.python.org/3/using/mac.html.

CHAPTER 4

Developer Notes

Here we gather important notes for the developing of oemof and elements within the framework.

We highly encourage you to contribute to further development of oemof. If you want to collaborate see description below or contact us.

4.1 Install the developer version

To install the developer version two steps are necessary:

```
git clone git@github.com:oemof/oemof.git
sudo pip3 install -e /path/to/the/repository
```

Newly added required packages (via PyPi) are installed by performing a manual upgrade of oemof. Therefore, run

```
sudo pip3 install --upgrade -e /path/to/the/repository
```

4.2 Documentation

See the developer version of the documentation of the dev branch at readthedocs.org.

4.3 Collaboration with pull requests

To collaborate use the pull request functionality of github.

4.3.1 How to create a pull request

- ...
- ...
- Tests must run, see ...
- Name the related team within the title of the pull request, like “solph: pull request for new feature within solph”.
- Assign a team member.

4.3.2 Tests

- ...
- ...

4.4 Style guidelines

We mostly follow standard guidelines instead of developing own rules. So if anything is not defined in this section, search for a PEP rule and follow it.

4.4.1 Docstrings

We decided to use the style of the numpydoc docstrings. See the following link for an example.

4.4.2 PEP8 (Python Style Guide)

- We adhere to PEP8 for any code produced in the framework.
- We use pylint to check your code. Pylint is integrated in many IDEs and Editors. Check here or ask the maintainer of your IDE or Editor
- Some IDEs have pep8 checkers, which are very helpful, especially for python beginners.

4.4.3 Quoted strings

As there is no recommendation in the PEP rules we use double quotes for strings read by humans such as logging/error messages and single quotes for internal strings such as keys and column names. However one can deviate from this rules if the string contains a double or single quote to avoid escape characters. According to PEP 257 and numpydoc we use three double quotes for docstrings.

```
logging.info("We use double quotes for messages")

my_dictionary.get('key_string')

logging.warning('Use three " to quote docstrings!'  # exception to avoid escape characters
```

4.5 Naming Conventions

- We use plural in the code for modules if there is possibly more than one child class (e.g. import transformers AND NOT transformer). If there are arrays in the code that contain multiple elements they have to be named in plural (e.g. *transformers = [T1, T2,...]*).
- Please, follow the naming conventions of pylint
- Use talking names
  - Variables/Objects: Name it after the data they describe (power_line, wind_speed)
  - Functions/Method: Name it after what they do: **use verbs** (get_wind_speed, set_parameter)

4.6 Using git

4.6.1 Branching model

So far we adhere mostly to the git branching model by Vincent Driessen.

Differences are:

- instead of the name `origin/develop` we call the branch `origin/dev`.
- feature branches are named like `features/*`
- release branches are named like `releases/*`

4.6.2 Commit message

Use this nice little tutorial to learn how to write a nice commit message.

4.7 Testing

We use nosetests for testing. Make sure that all tests are successfull before merging back into the `dev`.

```
cd /path/to/oemof/
nosetests3 --with-doctest           # or
nosetests3 --with-doctest --rednose # if you like it
```

4.8 Issue-Management

Section about workflow for issues is still missing (when to assign an issue with what kind of tracker to whom etc.).

CHAPTER 5

Mathematical notation for solph

5.1 Sets

Symbol	Description	Python-type of object in set
$\mathcal{E}$	Set of all Entities	Entity
$\vec{\mathcal{E}}$	Set of all weighted edges (e_i, e_j)	Tuple
$\mathcal{E}_\mathcal{B}$	Set of all edges	Bus
$\mathcal{E}_\mathcal{C}$	Set of all components	Component
$\mathcal{E}_I$	Set of components with 1 input	Sink
$\mathcal{E}_O$	Set of components with 1 output	Source
$\mathcal{E}_{IO}$	Components with 1 input, 1 output $i_e \neq o_{e,n}$	SimpleTransformer
$\mathcal{E}_{IOO}$	Components with 1 input, 2 outputs	SimpleCHP
$\mathcal{E}_S$	Components with 1 input, 1 output $i_e = o_{e,n}$	Storage
$\mathcal{I}_e$	All inputs of Entity e	Dict
$\mathcal{O}_e$	All outputs of Entity e	Dict
$\mathcal{T}$	Set of timesteps	List

5.2 Variables

Symbol	Description	Possible Set	Python variable
LP-models			
$w_{e_1,e_2}(t)$	Weight of Edge (e_1, e_2) at t	$(e_1, e_2) \in \vec{\mathcal{E}}$	`w[e1,e2,t]`
$l_e(t)$	Level of component e at t	$e \in \mathcal{E}_S$	`cap[e,t]`
$g^{pos}_{e_{o,1}}(t)$	Positive gradient between two sequential timesteps	$e \in \mathcal{E}_C$	`grad_pos_var[e,t]`
$g^{neg}_{e_{o,1}}(t)$	Negative gradient between two sequential timesteps	$e \in \mathcal{E}_C$	`grad_neg_var[e,t]`
Dispatch-source only			
$w^{cut}_{e,o_e}(t)$	Curtailment of value $w_{e,o_e}(t)$	$e \in \mathcal{E}_O$	`curtailment_var[e1,e2,t]`
Investment models			
$\overline{w}^{add}_{o_e}$	Optimized extension of bound $\overline{W}_{e,o_{e,1}}$	$e \in \mathcal{E}_C$	`add_out[e]`
$\overline{l}^{add}_{e}$	Optimized extension of bound $\overline{L}_e$	$e \in \mathcal{E}_S$	`add_cap[e]`
MILP-models			
$y_e(t)$	Binary status variable of component e at t	$e \in \mathcal{E}_C$	`y[e,t]`
$z^{start}_e(t)$	Binary startup variable of component e at t	$e \in \mathcal{E}_C$	`z_start[e,t]`
$z^{stop}_e(t)$	Binary shutdown variable of component e at t	$e \in \mathcal{E}_C$	`z_stop[e,t]`

5.3 Parameters

Parameters will be notate with uppercase.

Symbol	Description	Python variable	Pyth type
V_e	Value of Component $e \in \{\mathcal{E}_o, \mathcal{E}_I\}$	`val`	
V_e^{norm}	Normend value of component $e \in \{\mathcal{E}_o, \mathcal{E}_I\}$	`val`	
$\eta_{i_e,o_{e,n}}$	Efficiency at conversion of input i_e to $n-$th output $o_{e,n}$ of component e	`eta`	list
$\overline{W}_{e_1,e_2}$	Upper bound of variable w_{e_1,e_2}	`out_max` / `in_max`	list
$\underline{W}_{e_1,e_2}$	Lower bound of variable w_{e_1,e_2}	`out_min` / `in_min`	list
$\overline{L}_e$	Upper bound of variable l_e	`cap_max`	float
$\underline{L}_e$	Lower bound of variable l_e	`cap_min`	float
$C_{e_i}^{rate}$	C-rate input of component e	`c_rate_in`	float
$C_{e_a}^{rate}$	C-rate output of component e	`c_rate_out`	float
$\overline{O}_e^{global}$	Global limit for all outputs of entity e	`sum_out_limit`	float
$\overline{G}_{e_{o,1}}^{pos}$	Upper bound for positive gradient of 1st output	`grad_pos`	float
$\overline{G}_{e_{o,1}}^{neg}$	Upper bound for negative gradient of 1st output	`grad_neg`	float
C_e^{loss}	Loss of energy per timestep	`cap_loss`	float
$T_e^{min,off}$	Minimum down-time of component e	`t_min_off`	float
$T_e^{min,on}$	Minimum up-time of component e	`t_min_on`	float
Cost/Revenue parameters			
$C_{e,i}$	Costs for one unit inflow of Component e	`input_costs`	list
$C_{e,o}$	Costs for one unit outflow of Component e	`output_costs` `opex_var`	list
$R_{e,i}$	Revenues for one unit inflow of Component e	`input_revenues`	list
R_{e,o_n}	Revenues for one unit outflow of the n-th output of Component e	`output_revenues`	list
C_e^{cut}	Costs for curtailment of variable	`curtailment_costs`	float

CHAPTER 6

What's New

These are new features and improvements of note in each release

Releases

6.1 v0.0.8 ()

6.1.1 New features

6.1.2 Documentation

6.1.3 Testing

6.1.4 Bug fixes

6.1.5 Other changes

6.1.6 Contributors

6.2 v0.0.7 (May 4, 2016)

6.2.1 Bug fixes

- Exclude non working pyomo version

6.3 v0.0.6 (April 29, 2016)

6.3.1 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)

6.3.2 Documentation

6.3.3 Testing

- Fix constraint tests (Issue #137)

6.3.4 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.

6.3.5 Other changes

6.3.6 Contributors

- Uwe Krien
- Stephan Günther
- Simon Hilpert
- Cord Kaldemeyer
- Birgit Schachler

6.4 v0.0.5 (April 1, 2016)

6.4.1 New features

- There's now a `flexible transformer` 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)

6.4.2 Documentation

- The documentation of the `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.

6.4.3 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).

6.4.4 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)

6.4.5 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.

6.4.6 Contributors

- Uwe Krien
- Stephan Günther
- Guido Pleßmann

6.5 v0.0.4 (March 03, 2016)

6.5.1 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

6.5.2 Documentation

- Update developer notes

6.5.3 Testing

- Described testing procedures in developer notes
- New constraint tests for heating buses

6.5.4 Bug fixes

- Use of pyomo fast build
- Broken result-DataFrame in outputlib
- Dumping of EnergySystem

6.5.5 Other changes

- PEP8

6.5.6 Contributors

- Cord Kaldemeyer
- Uwe Krien
- Simon Hilpert
- Stephan Günther
- Clemens Wingenbach
- Elisa Papdis
- Martin Soethe
- Guido Plessmann

6.6 v0.0.3 (January 29, 2016)

6.6.1 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)

6.6.2 Documentation

- Added the Readme.rst as "Getting started" to the documentation.
- Fixed installation description (issue #38)
- Improved the developer notes.

6.6.3 Testing

- With this release we start implementing nosetests (issue #47
- Tests added to test constraints and the registration process (issue #73).

6.6.4 Bug fixes

- Fix contraints in solph
- Fix pep8

6.6.5 Other changes

6.6.6 Contributors

- Cord Kaldemeyer
- Uwe Krien
- Clemens Wingenbach
- Simon Hilpert
- Stephan Günther

6.7 v0.0.2 (December 22, 2015)

6.7.1 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` now keep track of `Entities` via the `entities` attribute. (issue #20)
- There's now a standard way of working with the results obtained via a call to `OptimizationModel#results`. See its documentation, the documentation of `EnergySystem#optimize` and finally the discussion at issue #33 for more information.
- New class `VariableEfficiencyCHP` to model combined heat and power units with variable electrical efficiency.
- New methods for `VariableEfficiencyCHP` inside the solph-module:
- `MILP-constraint`
- `Linear-constraint`

6.7.2 Documentation

- missing docstrings of the core subpackage added (issue #9)
- missing figures of the meta-documentation added
- missing content in developer notes (issue #34)

6.7.3 Testing

6.7.4 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

6.7.5 Other changes

6.7.6 Contributors

- Uwe Krien
- Simon Hilpert
- Cord Kaldemeyer
- Guido Pleßmann
- Stephan Günther

6.8 v0.0.1 (November 25, 2015)

First release by the oemof developing group.

CHAPTER 7

API

7.1 oemof

7.1.1 oemof package

Subpackages

oemof.core package

Subpackages

oemof.core.network package

Subpackages

oemof.core.network.entities package

Subpackages

oemof.core.network.entities.components package

Submodules

oemof.core.network.entities.components.sinks module

class oemof.core.network.entities.components.sinks.**Simple**(***kwargs*)

Bases: *oemof.core.network.entities.components.Sink*

A simple sink. Use this if you do not know which sink to use.

optimization_options = {}

oemof.core.network.entities.components.sources module

class oemof.core.network.entities.components.sources.**Commodity**(***kwargs*)
Bases: *oemof.core.network.entities.components.Source*

The commodity component can be used to model inputs to resource busses. At the moment no constraint etc. are implemented for this component.

optimization_options = {}

class oemof.core.network.entities.components.sources.**DispatchSource**(***kwargs*)
Bases: *oemof.core.network.entities.components.Source*

Dispatch sources only have one output (like FixedSource) but the output can be reduced inside the optimization problem.

optimization_options = {}

class oemof.core.network.entities.components.sources.**FixedSource**(***kwargs*)
Bases: *oemof.core.network.entities.components.Source*

A fixed source only has one output always. The value of the output is fixed for all timesteps in the timehorizon of the optimization problem.

optimization_options = {}

oemof.core.network.entities.components.transformers module

class oemof.core.network.entities.components.transformers.**CHP**(***kwargs*)
Bases: *oemof.core.network.entities.components.Transformer*

A CombinedHeatPower Transformer always has a simple input output relation with a constant efficiency

Parameters **eta** (*list*) – constant effciency for converting input into output. First element of list is used for conversion of input into first element of attribute *outputs*. Second element for second element of attribute *outputs*. E.g. eta = [0.3, 0.4]

optimization_options = {}

class oemof.core.network.entities.components.transformers.**Simple**(***kwargs*)
Bases: *oemof.core.network.entities.components.Transformer*

Simple Transformers always have a simple input output relation with a constant efficiency

Parameters **eta** (*list*) – constant efficiency for conversion of input into output (0 <= eta <= 1) e.g. eta = [0.4]

optimization_options = {}

class oemof.core.network.entities.components.transformers.**SimpleExtractionCHP**(***kwargs*)
Bases: *oemof.core.network.entities.components.transformers.CHP*

Class for combined heat and power unit with extraction turbine and constant power to heat coeffcient in back-pressure mode

Parameters

- **eta_el_cond** (*float*) – constant el. efficiency for transformer in condensing mode
- **beta** (*float*) – power loss index
- **sigma** (*float*) – power to heat ratio P/Q in backpressure mode

optimization_options = {}

class oemof.core.network.entities.components.transformers.**Storage**(***kwargs*)
Bases: *oemof.core.network.entities.components.Transformer*

Parameters

- **cap_max** (*float*) – absolut maximum state of charge if invest=FALSE, absolut maximum state of charge of built capacity if invest=TRUE
- **cap_min** (*float*) – absolut minimum state of charge
- **cap_initial** (*float, optional*) – The state of charge (soc) at timestep 0.
- **add_cap_limit** (*float*) – limit of additional installed capacity (only investment models)
- **eta** (*list (optional)*) – constant efficiency for charging, discharging input e.g. eta = [0.9, 0.9]
- **eta_in** (*float*) – efficiency at charging
- **eta_out** (*float*) – efficiency at discharging
- **cap_loss** (*float or list/pandas.Series with length of simulation timesteps*) – capacity loss per timestep in p/100
- **c_rate_in** (*float*) – c-rate for charging (unit is s^-1)
- **c_rate_out** (*float*) – c-rate for discharging (unit is s^-1)

optimization_options = {}

class oemof.core.network.entities.components.transformers.**TwoInputsOneOutput**(***kwargs*)
Bases: *oemof.core.network.entities.components.transformers.Simple*

A transformer with one output and two input flows

The two input flows are connect by the factor f. This transformer might represent components such as heat pumps and instant flow heater.

Parameters

- **eta** (*list*) – Constant effciency for converting the incoming flows to the internal input flows. The first element of eta is the efficiency of the first element of inputs and so on. If not set the effciency will be one. You have to define both elements or no element.
- **f** (*array-like*) – A relation-factor between the first and the second input flow.

class oemof.core.network.entities.components.transformers.**VariableEfficiencyCHP**(***kwargs*)
Bases: *oemof.core.network.entities.components.transformers.CHP*

A CombinedHeatPower Transformer with variable electrical efficiency Note: The model uses constraints which require binary variables, hence objects of this class will results in mixed-integer-linear-problems.

Parameters

- **eta_total** (*float*) – total constant efficiency for the transformer
- **eta_el** (*list*) – list containing the minimial (first element) and maximal (second element) electrical efficiency (0 <= eta_el <= 1)

optimization_options = {}

oemof.core.network.entities.components.transports module

class oemof.core.network.entities.components.transports.**Simple**(***kwargs*)
Bases: *oemof.core.network.entities.components.Transport*

Simple Transport connects two busses with a constant efficiency

Parameters

- **eta** (*float*) – Constant efficiency of the transport.
- **in_max** (*float*) – Maximum input the transport can handle, in MW.
- **out_max** (*float*) – Maximum output which can possibly be obtained when using the transport, in MW.

optimization_options = {}

Module contents

class oemof.core.network.entities.components.**Sink**(***kwargs*)
Bases: *oemof.core.network.entities.Component*

A Sink is special Component which only consumes some source commodity. Therefore its list of outputs has to be either None or empty (i.e. logically False).

class oemof.core.network.entities.components.**Source**(***kwargs*)
Bases: *oemof.core.network.entities.Component*

The opposite of a Sink, i.e. a Component which only produces and as a consequence has no input.

Parameters

- **in_max** (*list*) – maximum input of component (e.g. in MW)
- **out_max** (*list*) – maximum output of component (e.g. in MW)
- **add_out_limit** (*float*) – limit on additional output "capacity" (e.g. in MW)
- **capex** (*float*) – capital expenditure (e.g. in Euro / MW)
- **lifetime** (*float*) – lifetime of component (e.g. years)
- **wacc** (*float*) – weigted average cost of capital (dimensionless)
- **crf** (*float*) – capital recovery factor: (p*(1+p)^n)/(((1+p)^n)-1)
- **opex_fix** (*float*) – fixed operational expenditure (e.g. expenses for staff)
- **opex_var** (*float*) – variable operational expenditure (e.g. spare parts + fuelcosts)
- **co2_fix** (*float*) – fixed co2 emissions (e.g. t / MW)
- **co2_var** (*float*) – variable co2 emissions (e.g. t / MWh)
- **co2_cap** (*float*) – co2 emissions due to installed power (e.g. t/ MW)

optimization_options = {}

class oemof.core.network.entities.components.**Transformer**(***kwargs*)
Bases: *oemof.core.network.entities.Component*

A Transformer is a specific type of Component which transforms (possibly m) inputs into (possibly n) outputs. As such neither its list of inputs, nor its list of outputs are allowed to be empty.

Parameters

- **out_min** (*list*) – minimal output of transformer (e.g. min power output of powerplants)
- **in_min** (*list*) – minimal input of transformer (e.g. min fuel consumption of powerplants)
- **grad_pos** (*float*) – positive gradient (absolut value between two sequential timesteps)
- **grad_neg** (*float*) – negative gradient (absolut value between two sequential timesteps)
- **t_min_off** (*float*) – minimal off time in timesteps (e.g. 5 hours)

- **t_min_on** (*float*) – minimal on time in timesteps (e.g 5 hours)
- **outages** (*float or array*) – Outages of component. either: defined timesteps of timehorizon: e.g. [1,4,200] or: 0 <= scalar <= 1 as factor of the total timehorizon e.g. 0.05
- **input_costs** (*float*) – costs for usage of input (if not included in opex_var)
- **start_costs** (*float*) – cost per start up of transformer (only milp models)
- **stop_costs** (*float*) – cost per stop up of transformer (only milp models)
- **ramp_costs** (*float*) – costs for ramping
- **output_price** (*list*) – price for selling to output bus. prices ordered in the order of 'outputs'
- **eta_min** (*list*) – efficiency of transformer at minimum load for conversion of input to output (order of elements corresponding to order of elements out outputs,out_min etc.)

optimization_options = {}

class oemof.core.network.entities.components.**Transport**(***kwargs*)
Bases: *oemof.core.network.entities.Component*

A Transport is a simple connection transporting a commodity from one Bus to a different one. It is different from a Transformer in that it may not change the type of commodity being transported. But since the transfer can still change things about the commodity other than the type (loss, gain, time delay, etc.) this class exists to encapsulate such changes.

optimization_options = {}

Module contents

class oemof.core.network.entities.**Bus**(***kwargs*)
Bases: *oemof.core.network.Entity*

The other type of entity in an energy system graph (besides Components). A Bus acts as a kind of mediator between producers and consumers of a commodity of the same kind. As such it has a type, which signifies what kind of commodity goes through the bus.

Parameters

- **type** (*string*) – the type of the bus. Can be a meaningful value like e.g. "electricity" but may be anything that can be tested for equality and is distinct for incompatible Buses.
- **price** (*float*) – price per unit of type
- **balanced** (*boolean*) – if true a busbalance is created, otherwise the busbalance is ignored
- **sum_out_limit** (*float (default: +inf)*) – limit of sum of all outflows over the timehorizon
- **TODO** (*#*) –
- **excess** (*boolean*) – if true, an optimization variable is created that takes up the slack of outflows to keep the busbalance (sum inflows = sum outflows + excess)
- **shortage** (*boolean*) – if true, an optimization variable is created that takes the slack of inflows to keep the busbalance (sum inflows + shortage = sum outflows)
- **excess_costs** (*float*) – costs per unit of excess that is needed to balance the bus
- **shortage_costs** (*float*) – costs per unit of shortage that is needed to balance the bus

create_excess_slack_component()

create_shortage_slack_component()

optimization_options = {}

class oemof.core.network.entities.**Component**(***kwargs*)
Bases: *oemof.core.network.Entity*

Components are one specific type of entity comprising an energy system graph, the other being Buses. The important thing is, that connections in an energy system graph are only allowed between Buses and Components and not between Entities of equal subtypes. This class exists only to facilitate this distinction and is empty otherwise.

Parameters

- **in_max** (*list*) – maximum input of component (power)
- **out_max** (*list*) – maximum output of component (power)
- **ub_out** (*list of pandas.series (or array-like)*) – The time-depended maximum output of a component. If ub_out is not set out_max is used. Contrary to out_max ub_out has to be array-like. If ub_out is set out_max is used as the installed capacity and ub_out as the time-depended maximum output. It is in charge of the user that these values are not inconsistent. You may use max(ub_out) for out_max. (power)
- **add_out_limit** (*float*) – limit on additional output "capacity" (power)
- **capex** (*float*) – Capital expenditure. To get the capital cost it is multiplied with out_max (monetory value per power).
- **lifetime** (*float*) – lifetime of component (e.g. years)
- **wacc** (*float*) – weigted average cost of capital (dimensionless)
- **crf** (*float*) – capital recovery factor: (p*(1+p)^n)/(((1+p)^n)-1)
- **opex_fix** (*float*) – fixed operational expenditure (e.g. expenses for staff) (monetory value over the full time period).
- **opex_var** (*float*) – Variable operational expenditure (e.g. spare parts). You can use it to define the fuel costs. Fuel cost can be defined in different ways, so be aware not to definge them twice.
- **co2_var** (*float*) – variable co2 emissions (e.g. t / MWh)
- **co2_cap** (*float*) – co2 emissions due to installed power (e.g. t/ MW)

optimization_options = {}

class oemof.core.network.entities.**ExcessSlack**(***kwargs*)
Bases: *oemof.core.network.entities.Component*

A ExcessSlack is a special sink which takes the output slack i.e. excess of the bus to that it is connected.

class oemof.core.network.entities.**ShortageSlack**(***kwargs*)
Bases: *oemof.core.network.entities.Component*

A ShorageSlack is a special source which takes the input slack i.e. shortage of the bus to that it is connected.

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

class oemof.core.network.**Entity**(***kwargs*)
Bases: object

The most abstract type of vertex in an energy system graph. Since each entity in an energy system has to be uniquely identifiable and connected (either via input or via output) to at least one other entity, these properties are collected here so that they are shared with descendant classes.

Parameters

- **uid** (*string or tuple*) – Unique component identifier of the entity.
- **inputs** (*list*) – List of Entities acting as input to this Entity.
- **outputs** (*list*) – List of Entities acting as output from this Enity.
- **geo_data** (*shapely.geometry object*) – Geo-spatial data with informations for location/region-shape. The geometry can be a polygon/multi-polygon for regions, a line fore transport objects or a point for objects such as transformer sources.

registry
EnergySystem

The central registry keeping track of all *Entities* created. If this is *None*, *Entity* instances are not kept track of. When you instantiate an *EnergySystem* it automatically becomes the entity registry, i.e. all entities created are added to its *entities* attribute on construction.

add_regions(*regions*)
Add regions to self.regions

optimization_options = {}

registry = None

Submodules

oemof.core.energy_system module Created on Mon Jul 20 15:53:14 2015

@author: uwe

class oemof.core.energy_system.**EnergySystem**(***kwargs*)
Bases: object

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

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

Parameters

- **entities** (list of *Entity*, optional) – A list containing the already existing *Entities* that should be part of the energy system. Stored in the *entities* attribute. Defaults to *[]* if not supplied.
- **simulation** (*core.energy_system.Simulation object*) – Simulation object that contains all necessary attributes to start the solver library. Defined in the *Simulation* class.
- **regions** (*list of core.energy_system.Region objects*) – List of regions defined in the *Region* class.
- **time_idx** (*pandas.index, optional*) – Define the time range and increment for the energy system. This is an optional parameter but might be import for other functions/methods that use the EnergySystem class as an input parameter.

- **groupings** (*list*) – The elements of this list are used to construct *Groupings* or they are used directly if they are instances of *Grouping*. These groupings are then used to aggregate the entities added to this energy system into *groups*. By default, there'll always be one group for each `uid` containing exactly the entity with the given `uid`. See the *examples* for more information.

entities
list of *Entity*

A list containing the *Entities* that comprise the energy system. If this *EnergySystem* is set as the *registry* attribute, which is done automatically on *EnergySystem* construction, newly created *Entities* are automatically added to this list on construction.

groups
dict

simulation
core.energy_system.Simulation object

Simulation object that contains all necessary attributes to start the solver library. Defined in the *Simulation* class.

regions
list of core.energy_system.Region objects

List of regions defined in the *Region* class.

results
dictionary

A dictionary holding the results produced by the energy system. Is *None* while no results are produced. Currently only set after a call to *optimize()* after which it holds the return value of *om.results()*. See the documentation of that method for a detailed description of the structure of the results dictionary.

time_idx
pandas.index, optional

Define the time range and increment for the energy system. This is an optional atribute but might be import for other functions/methods that use the EnergySystem class as an input parameter.

Examples

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

```
>>> from oemof.core.network import Entity
>>> from oemof.core.network.entities import Bus, Component
>>> es = EnergySystem()
>>> bus = Bus(uid='electricity')
>>> bus is es.groups['electricity']
True
```

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

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

```
>>> components == es.groups[Component]
True
```

add(*entity*)
Add an *entity* to this energy system.

connect(*bus1*, *bus2*, *in_max*, *out_max*, *eta*, *transport_class*)
Create two transport objects to connect two buses of the same type in both directions.

Parameters

- **bus2** (*bus1,*) – Two buses to be connected.
- **eta** (*float*) – Constant efficiency of the transport.
- **in_max** (*float*) – Maximum input the transport can handle, in MW.
- **out_max** (*float*) – Maximum output which can possibly be obtained when using the transport, in MW.
- **class** (*transport_class*) – Transport class to use for the connection

dump(*dpath=None*, *filename=None*, *keep_weather=True*)
Dump an EnergySystem instance.

optimize(*om=None*)
Start optimizing the energy system using solph.

Parameters **om** (*OptimizationModel*, optional) – The optimization model used to optimize the *EnergySystem*. If not given, an *OptimizationModel* instance local to this method is created using the current *EnergySystem* instance as an argument. You only need to supply this if you want to observe any side effects that solving has on the *om*.

Returns self

Return type *EnergySystem*

restore(*dpath=None*, *filename=None*)
Restore an EnergySystem instance.

class oemof.core.energy_system.**Grouping**(*key*, *value=<function Grouping.<lambda>>*, *collide=<function Grouping.<lambda>>*, *insert=None*)
Bases: object

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

UID = <oemof.core.energy_system.Grouping object>
The default grouping, which is always present in addition to user defined ones. Stores every *entity* in a group of its own under its uid and raises an error if another *entity* with the same uid get's added to the energy system.

static create(*argument*)

class oemof.core.energy_system.**Region**(***kwargs*)
Bases: object

Defining a region within an energy supply system.

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

Parameters

- **entities** (*list of core.network objects*) – List of all objects of the energy system. All class descriptions can be found in the *oemof.core.network* package.
- **name** (*string*) – A unique name to identify the region. If possible use typical names for regions and english names for countries.
- **code** (*string*) – A short unique name to identify the region.
- **geom** (*shapely.geometry object*) – The geometry representing the region must be a polygon or a multi polygon.

entities
list of core.network objects

List of all objects of the energy system. All class descriptions can be found in the *oemof.core.network* package.

name
string

A unique name to identify the region. If possible use typical names for regions and english names for countries.

geom
shapely.geometry object

The geometry representing the region must be a polygon or a multi polygon.

add_entities(*entities*)
Add a list of entities to the existing list of entities.

For every entity added to a region the region attribute of the entity is set

Parameters **entities** (*list of core.network objects*) – List of all objects of the energy system that belongs to area covered by the polygon of the region. All class descriptions can be found in the *oemof.core.network* package.

code
Creating a short code based on the region name if no code is set.

class oemof.core.energy_system.**Simulation**(***kwargs*)
Bases: object

Defining the simulation related parameters according to the solver lib.

Parameters

- **solver** (*string*) – Name of the solver supported by the used solver library. (e.g. 'glpk', 'gurobi')
- **debug** (*boolean*) – Set the chosen solver to debug (verbose) mode to get more information.
- **verbose** (*boolean*) – If True, solver output etc. is streamed in python console
- **duals** (*boolean*) – If True, results of dual variables and reduced costs will be saved
- **objective_options** (*dictionary*) –

  'function': function to use from *oemof.solph.predefined_objectives*

  'cost_objects': list of str(*class*) elements. Objects of type *class* are include in cost terms of objective function.

'revenue_objects': list of str(*class*) elements. . Objects of type *class* are include in revenue terms of objective function.

- **timesteps** (*list or sequence object*) – Timesteps to be simulated or optimized in the used library
- **relaxed** (*boolean*) – If True, integer variables will be relaxed (only relevant for milp-problems)
- **fast_build** (*boolean*) – If True, the standard way of pyomo constraint building is skipped and a different function is used. (Warning: No guarantee that all expected 'standard' pyomo model functionalities work for the constructed model!)

Module contents

oemof.demandlib package

Submodules

oemof.demandlib.bdew_heatprofile module Implementation of the bdew heat load profiles

oemof.demandlib.bdew_heatprofile.**create_bdew_profile**(*datapath*, *year*, *temperature*, *annual_heat_demand*, *shlp_type*, *wind_class*, ***kwargs*)

Calculation of the hourly heat demand using the bdew-equations

Parameters

- **year** (*int*) – year or which the profile is created
- **datapath** (*string*) – path where csv-files with bdew data are located
- **annual_heat_demand** (*float*) – annual heat demand of building in kWh
- **building_class** (*int*) – class of building according to bdew classification possible numbers are: 1 - 11
- **shlp_type** (*string*) – type of standardized heat load profile according to bdew possible types are: GMF, GPD, GHD, GWA, GGB, EFH, GKO, MFH, GBD, GBA, GMK, GBH, GGA, GHA
- **wind_class** (*int*) – wind classification for building location (1 if windy, else 0)

oemof.demandlib.bdew_heatprofile.**get_SF_values**(*df*, *datapath*, *filename='shlp_hour_factors.csv'*, *building_class=None*, *shlp_type=None*)

Determine the h-values

Parameters

- **datapath** (*string*) – path where csv files are located
- **filename** (*string*) – name of file where sigmoid factors are stored
- **building_class** (*int*) – class of building according to bdew classification

oemof.demandlib.bdew_heatprofile.**get_sigmoid_parameters**(*datapath*, *building_class=None*, *shlp_type=None*, *wind_class=None*, *ww_incl=True*, *filename='shlp_sigmoid_factors.csv'*)

Retrieve the sigmoid parameters from csv-files

Parameters

- **datapath** (*string*) – path where csv files are located
- **filename** (*string*) – name of file where sigmoid factors are stored
- **building_class** (*int*) – class of building according to bdew classification
- **shlp_type** (*string*) – type of standard heat load profile according to bdew
- **wind_class** (*int*) – wind classification for building location (0=not windy or 1=windy)
- **ww_incl** (*boolean*) – decider whether warm water load is included in the heat load profile

oemof.demandlib.bdew_heatprofile.**get_temperature_interval**(*df*)

Appoints the corresponding temperature interval to each temperature in the temperature vector.

Parameters **df** (*pandas.DataFrame*) – dataframe containing informations on temperature in column "temperature"

oemof.demandlib.bdew_heatprofile.**get_weekday_parameters**(*df*, *datapath*, *filename='shlp_weekday_factors.csv'*, *shlp_type=None*)

Retrieve the weekday parameter from csv-file

Parameters

- **df** (*pandas.DataFrame*) – dataframe containing informations on temperature and date/time
- **datapath** (*string*) – path where csv files are located
- **filename** (*string*) – name of file where sigmoid factors are stored
- **shlp_type** (*string*) – type of standard heat load profile according to bdew

oemof.demandlib.bdew_heatprofile.**weighted_temperature**(*df*, *how='geometric_series'*)

A new temperature vector is generated containing a multi-day average temperature as needed in the load profile function.

Parameters

- **df** (*pandas.DataFrame*) – dataframe containing hourly temperature data in column with label "temperature"
- **how** (*string*) – string which type to return ("geometric_series" or "mean")

Notes

Equation for the mathematical series of the average tempaerature [1]:

$$T = \frac{T_D + 0.5 \cdot T_{D-1} + 0.25 \cdot T_{D-2} + 0.125 \cdot T_{D-3}}{1 + 0.5 + 0.25 + 0.125}$$

[1] BDEW, BDEW Documentation for heat profiles.

with T_D = Average temperature on the present day T_{D-i} = Average temperature on the day - i

References

oemof.demandlib.demand module Created on Fri Jul 24 19:11:38 2015

@author: caro

class oemof.demandlib.demand.**electrical_demand**(*method*, ***kwargs*)
Bases: object

Calculate the electrical demand for a region with different methods.

This class calculates the electrical demand for a region. Therefore several different methods can be applied.

Parameters **method** (*{'scale_profile_csv', 'scale_profile_db', 'scale_entsoe', 'calculate_profile'}*) – Method to calculate the demand for your region. Explanation:

'scale_profile_csv': read only profile from csv and scale it with given or calculated demand

'scale_profile_db': read only profile from database and scale it with given or calculated demand

'scale_entsoe': read entsoe profile from database and scale it with given or calculated demand

'calculate_profile: Calculate profile from the standard load profiles of the three demand sectors (households, service, industry) and the corresponding annual electric demand.

Depending on which of these values is chosen, additional parameters might be required, as detailed below.

Other Parameters

- **path** (*str*) – Required for a *method* value of *'scale_profile_csv'*.

  Should be the '/path/to/the/csv/file'.

- **filename** (*str*) – Required for a *method* value of *'scale_profile_csv'*.

  Should be the name of the file found under *path*.

- **conn** – Required for *method* values of *'scale_profile_db'* or *'scale_profile_entsoe'*.

- **annual_elec_demand** (*int*) – Required for *method* values of *'scale_profile_csv'*, *'scale_profile_db'* or *'scale_profile_entsoe'*.

  Annual demand of your region. Works so far only with a given value. Calculating the demand from statistic data for the whole region can be an option for further development.

- **ann_el_demand_per_sector** (*dictionary*) – Required for a *method* value of *'calculate_profile'*.

  Specification of annual electric demand and the corresponding standard load profile type (selp_type) for every sector. Dictionary is structured as follows. Key defining the sector is followed by value that can be int, float, None or can be omitted, e.g.:

```
ann_el_demand_per_sector = {
    'h0': int,
    'g0': float,
    'g1': None,
    ...
```

```
                'g6': int,
                'i0': int}
```

If ann_el_demand is None, more parameters to calculate the demand are necessary: (works so far only if ann_el_demand for every or no sector is specified)

- **population** (*int*) – Population of your region.
- **ann_el_demand_per_person** (*list of dictionaries*) – Specification of the annual electric demand for one household according to the household type (from single to four-person households), e.g.:

```
ann_el_demand_per_person = [
    {'ann_el_demand': int,
     'household_type': {'one', 'two', 'three', 'four'}},
     ...]
```

- **household_structure** (*list of dictionaries*) – Number of people living in every household type. Specification for your region, e.g.:

```
household_structure = [
    {household_members': int,
     'household_type': {'one', 'two', 'three', 'four'}},
     ...]
```

- **comm_ann_el_demand_state** (*int*) – Annual electric demand of the service sector of the next bigger region, if not given for your region.
- **comm_number_of_employees_state** (*int*) – Number of employees in the service sector of the next bigger region.
- **comm_number_of_employees_region** (*int*) – Number of employees in the service sector of your region.

annual_demand
int

Included in ***kwargs*. Given with initialization or calculated or to be calculated within this class according to selected method.

dataframe
pandas dataframe

elec_demand
array_like

profile

e_slp

Notes

References

statistics ...

B. Schachler: Bewertung des Einsatzes von Kraft-Wärme-Kopplungsanlagen hinsichtlich der CO2-Emissionen bei wachsendem Anteil Erneuerbarer Energien, Masterarbeit, Technische Universität Berlin, 2014

Examples

These are written in doctest format, and should illustrate how to use the function.

26. (a)Summe bilden von elec_demand und abgleichen mit ann_el_demand

calculate_annual_demand_commerce(***kwargs*)

calculate_annual_demand_households(***kwargs*)

calculate_annual_demand_industry(***kwargs*)

calculate_annual_demand_region()
calculate annual demand from statistic data

decider(*method*, ***kwargs*)

read_entsoe()

read_from_csv(***kwargs*)
read entire demand timeseries or only profile for further processing from csv

read_from_db()
read entire demand timeseries or only profile for further processing from database

read_selp()

scale_profile()
scale a given profile to a given annual demand, which is the sum of the single profile values

class oemof.demandlib.demand.**heat_demand**
Bases: object

oemof.demandlib.energy_buildings module

class oemof.demandlib.energy_buildings.**Building**(***kwargs*)
Bases: object

hourly_heat_demand(*fun=None*, ***kwargs*)
Calculate hourly heat demand

Parameters

- **self** (*building object*) –
- **fun** (*python function*) – function to use for heat demand calculation
- **kwargs** (*additional arguments*) – arguments to use in fun

class oemof.demandlib.energy_buildings.**IndustrialLoadProfile**(*method*, ***kwargs*)
Bases: object

Generate an industrial heat or electric load profile.

decider(*method*, ***kwargs*)

simple_industrial_profile(***kwargs*)
Create industrial load profile

Parameters

- **am** (*datetime.time*) – beginning of workday
- **pm** (*datetime.time*) – end of workday
- **week** (*list*) – list of weekdays

- **weekend** (*list*) – list of weekend days
- **profile_factors** (*dictionary*) – dictionary with scaling factors for night and day of weekdays and weekend days

slp

class oemof.demandlib.energy_buildings.**bdew_elec_slp**(*time_df*, *periods=None*)
Bases: object

Generate electrical standardized load profiles based on the BDEW method.

all_load_profiles(*time_df*)

create_bdew_load_profiles(*time_df*, *slp_types*)
Calculates the hourly electricity load profile in MWh/h of a region.

slp

year

Module contents

oemof.outputlib package

Submodules

oemof.outputlib.to_pandas module

class oemof.outputlib.to_pandas.**DataFramePlot**(***kwargs*)
Bases: *oemof.outputlib.to_pandas.ResultsDataFrame*

Creates plots based on the subset of a multi-indexed pandas dataframe of the *ResultsDataFrame class*.

Parameters

- **subset** (*pandas.DataFrame*) – A subset of the results DataFrame.
- **ax** (*matplotlib axis object*) – Axis object of the last plot.

subset
pandas.DataFrame

A subset of the results DataFrame.

ax
matplotlib axis object

Axis object of the last plot.

color_from_dict(*colordict*)
Method to convert a dictionary containing the components and its colors to a color list that can be directly useed with the color parameter of the pandas plotting method.

Parameters **colordict** (*dictionary*) – A dictionary that has all possible components as keys and its colors as items.

Returns Containing the colors of all components of the subset attribute

Return type list

io_plot(*bus_uid*, *cdict*, *line_kwa={}*, *lineorder=None*, *bar_kwa={}*, *barorder=None*, ***kwargs*)
Plotting a combined bar and line plot to see the fitting of in- and outcomming flows of a bus balance.

Parameters

- **bus_uid** (*string*) – Uid of the bus to plot the balance.
- **cdict** (*dictionary*) – A dictionary that has all possible components as keys and its colors as items.
- **line_kwa** (*dictionary*) – Keyword arguments to be passed to the pandas line plot.
- **bar_kwa** (*dictionary*) – Keyword arguments to be passed to the pandas bar plot.
- **lineorder** (*list*) – Order of columns to plot the line plot
- **barorder** (*list*) – Order of columns to plot the bar plot

Note: Further keyword arguments will be passed to the *slice_unstacked method*.

Returns Manipulated labels to correct the unsual construction of the stack line plot. You can use them for further maipulations.

Return type handles, labels

outside_legend(*reverse=False*, *plotshare=0.9*, ***kwargs*)

Move the legend outside the plot. Bases on the ideas of Joe Kington. See http://stackoverflow.com/questions/4700614/how-to-put-the-legend-out-of-the-plot for more information.

Parameters

- **reverse** (*boolean (default: False)*) – Print out the legend in reverse order. This is interesting for stack-plots to have the legend in the same order as the stacks.
- **plotshare** (*real (default: 0.9)*) – Share of the plot area to create space for the legend (0 to 1).
- **loc** (*string (default: 'center left')*) – Location of the plot.
- **bbox_to_anchor** (*tuple (default: (1, 0.5))*) – Set the anchor for the legend.
- **ncol** (*integer (default: 1)*) – Number of columns of the legend.
- **handles** (*list of handles*) – A list of handels if they are already modified by another function or method. Normally these handles will be automatically taken from the artis object.
- **lables** (*list of labels*) – A list of labels if they are already modified by another function or method. Normally these handles will be automatically taken from the artis object.

Note: All keyword arguments (kwargs) will be directly passed to the matplotlib legend class. See http://matplotlib.org/api/legend_api.html#matplotlib.legend.Legend for more parameters.

plot(***kwargs*)

Passing the subset attribute to the pandas plotting method. All parameters will be directly passed to pandas.DataFrame.plot(). See http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html for more information.

Returns

Return type self

rearrange_subset(*order*)
Change the order of the subset DataFrame

Parameters **order** (*list*) – New order of columns

Returns

Return type self

set_datetime_ticks(*tick_distance=None*, *number_autoticks=3*, *date_format='%d-%m-%Y %H:%M'*)
Set configurable ticks for the time axis. One can choose the number of ticks or the distance between ticks and the format.

Parameters

- **tick_distance** (*real*) – The disctance between to ticks in hours. If not set autoticks are set (see number_autoticks).
- **number_autoticks** (*int (default: 3)*) – The number of ticks on the time axis, independent of the time range. The higher the number of ticks is, the shorter should be the date_format string.
- **date_format** (*string (default: '%d-%m-%Y %H:%M')*) – The string to define the format of the date and time. See https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior for more information.

slice_unstacked(*unstacklevel='obj_uid'*, ***kwargs*)
Method for slicing the ResultsDataFrame. The subset attribute will set to an unstacked subset. The self-attribute is returned to allow chaining. This method is an extension of the *slice_unstacked* method of the *ResultsDataFrame* class (parent class).

Parameters **unstacklevel** (*string (default: 'obj_uid')*) – Level to unstack the subset of the DataFrame.

class oemof.outputlib.to_pandas.**ResultsDataFrame**(***kwargs*)
Bases: pandas.core.frame.DataFrame

Creates a multi-indexed pandas dataframe from a solph result object and holds methods to create subsets of the data.

Note: This is so far only a rough sketch and serves as a base for discussion.

Parameters

- **result_object** (*dictionary*) – solph result objects
- **bus_uids** (*list if strings*) – List of strings with busses that should be contained in dataframe. If not set, all busses are contained.
- **bus_types** (*list if strings*) – List of strings with bus types that should be contained in dataframe. If not set, all bus types are contained.

result_object
dictionary

solph result objects

bus_uids
list if strings

List of strings with busses that should be contained in dataframe. If not set, all busses are contained.

bus_types
list if strings

List of strings with bus types that should be contained in dataframe. If not set, all bus types are contained.

data_frame
pandas dataframe

Multi-indexed pandas dataframe holding the data from the result object. For more information on advanced dataframe indexing see: http://pandas.pydata.org/pandas-docs/stable/advanced.html

bus_uids
list if strings

List of strings with busses that should be contained in dataframe

bus_types
list if strings

List of strings with bus types that should be contained in dataframe.

slice_by(***kwargs*)
Method for slicing the ResultsDataFrame. A subset is returned.

Parameters

- **bus_uid** (*string*) –
- **bus_type** (*string (e.g. "el" or "gas")*) –
- **type** (*string (input/output/other)*) –
- **obj_uid** (*string*) –
- **date_from** (*string*) – Start date selection e.g. "2016-01-01 00:00:00". If not set, the whole time range will be plotted.
- **date_to** (*string*) – End date selection e.g. "2016-03-01 00:00:00". If not set, the whole time range will be plotted.

slice_unstacked(*unstacklevel='obj_uid'*, ***kwargs*)
Method for slicing the ResultsDataFrame. A unstacked subset is returned.

Parameters **unstacklevel** (*string (default: 'obj_uid')*) – Level to unstack the subset of the DataFrame.

Module contents

oemof.solph package

Submodules

oemof.solph.linear_constraints module The linear_contraints module contains the pyomo constraints wrapped in functions. These functions are used by the '_assembler- methods of the OptimizationModel()-class.

The module frequently uses the dictionaries I and O for the construction of constraints. I and O contain all components' uids as dictionary keys and the relevant input input/output uids as dictionary items.

Illustrative Example:

Consider the following example of a chp-powerplant modeled with 4 entities (3 busses, 1 component) and their unique ids being stored in a list called *uids*:

```
>>> uids = ['bus_el', 'bus_th', 'bus_coal','pp_coal']
>>> I = {'pp_coal': 'bus_coal'}
>>> O = {'pp_coal': ['bus_el', 'bus_th']}
>>> print(I['pp_coal'])
bus_coal
```

In mathematical notation I, O can be seen as indexed index sets The elements of the sets are the uids of all components (index: *e*). The the inputs/outputs uids are the elements of the accessed set by the component index *e*. Generally the index *e* is the index for the uids-sets containing the uids of objects for which the constraints are build. For all mathematical constraints the following definitions hold:

Inputs: $\mathcal{I}_e =$ Input-uids of entity $e \in \mathcal{E}$

Outputs: $\mathcal{O}_e =$ All output-uids of entity $e \in \mathcal{E}$

Simon Hilpert (simon.hilpert@fh-flensburg.de)

oemof.solph.linear_constraints.**add_bus_balance**(*model*, *block=None*)

Adds constraint for the input-ouput balance of bus objects.

The mathematical formulation for the balance is as follows:

$$\sum_{i \in \mathcal{I}_e} w_{i,e}(t) = \sum_{o \in O_e} w_{e,o}(t), \qquad \forall e, \forall t$$

With $e \in \mathcal{E}_B, t \in \mathcal{T}$

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_dispatch_source**(*model*, *block*)

Creates dispatchable source bounds/constraints.

First the maximum value for the output of the source will be set. Then a constraint is defined that determines the dispatch of the source. This dispatch can be used in the objective function to add cost for dispatch of sources.

The mathemathical formulation of the constraint is as follows:

$$w^{cut}_{e,o_e}(t) = V^{norm}_e(t) \cdot \overline{W}_{e,o_e} - w_{e,o_e}(t), \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped in the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_O$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_eta_total_chp_relation**(*model*, *block*)

Adds constraints for input-(output1,output2) relation as simple function for all objects in *block.objs*.

The mathematical formulation of the input-output relation of a simple transformer is as follows:

$$w_{i_e,e}(t) \cdot \eta^{total}_e = w_{e,o_{e,1}}(t) + w_{e,o_{e,2}}(t), \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_{IOO}$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_fixed_source**(*model*, *block*)
Sets fixed source bounds and constraints for all objects in *block.objs*

The mathematical formulation is as follows:

$$w_{e,o_e}(t) = V_e^{norm}(t) \cdot \overline{W}_{e,o_e}, \qquad \forall e, \forall t$$

For *investment* for component:

$$w_{e,o}(t) \leq (\overline{W}_{e,o_e} + \overline{w}_{e_o}^{add}) \cdot V_e^{norm}(t), \qquad \forall e, \forall t$$

$$\overline{w}_{e_o}^{add} \leq \overline{W}_{e_o}^{add}, \qquad \forall e$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_O$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_global_output_limit**(*model*, *block=None*)
Adds constraints to set limit for variables as sum over the total timehorizon for all objects in *block.objs*

The mathematical formulation is as follows:

$$\sum_{t \in \mathcal{T}} \sum_{o \in \mathcal{O}_e} w_{e,o}(t) \leq \overline{O}_e^{global}, \qquad \forall e \in E$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \{\mathcal{E}_B, \mathcal{E}_O\}$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_output_gradient_calc**(*model*, *block*, *grad_direc='both'*, *idx=0*)
Add constraint to calculate the gradient between two timesteps (positive and negative)

The mathematical formulation for constraints are as follows:

Positive gradient:

$$w_{e,o_{e,n}}(t) - w_{e,o_{e,n}}(t-1) \leq g_{e_o}^{pos}(t) \qquad \forall e, \forall t/t = 1$$

$$g_{e_o}^{pos}(t) \leq \overline{G}_{e_o}^{pos}, \qquad \forall e, \forall t$$

Negative gradient:

$$w_{e,o_{e,n}}(t-1) - w_{e,o_{e,n}}(t) \leq g_{e_o}^{neg}(t) \qquad \forall e, \forall t/t = 1$$

$$g_{e_o}^{neg}(t) \leq \overline{G}_{e_o}^{neg}, \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped in the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_C$.

n indicates the n-th output of component e (arg: idx)

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class
- **grad_direc** (*string*) – string defining the direction of the gradient constraint. ('positive', negative', 'both')

oemof.solph.linear_constraints.**add_simple_chp_relation**(*model*, *block*)

Adds constraint for output-output relation for all simple combined heat an power units in *block.objs*.

The mathematical formulation for the constraint is as follows:

$$\frac{w_{e,o_{e,1}}(t)}{\eta_{e,o_{e,1}}(t)} = \frac{w_{e,o_{e,2}}(t)}{\eta_{e,o_{e,2}}(t)}, \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_{IOO}$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_simple_extraction_chp_relation**(*model*, *block*)

Adds constraints for power to heat relation and equivalent output for a simple extraction combined heat an power units. The constraints represent the PQ-region of the extraction unit and are set for all objects in *block.objs*

The mathematical formulation is as follows:

For Power/Heat ratio:

$$w_{e,o_{e,1}}(t) = w_{e,o_{e,2}}(t) \cdot \sigma_e, \qquad \forall e, \forall t$$

$$\sigma_e = \text{Power to heat ratio of entity } e$$

For euivalent power:

$$w_{i_e,e}(t) = \frac{w_{e,o_{e,1}}(t) + \beta_e \cdot w_{e,o_{e,2}}(t)}{\eta_{i_e,o_{e,1}}}$$

$$\beta_e = \text{Power loss index of entity } e$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_{IOO}$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_simple_io_relation**(*model*, *block*, *idx=0*)

Adds constraints for input-output relation as simple function for all objects in *block.objs*.

The mathematical formulation of the input-output relation of a simple transformer is as follows:

$$w_{i_e,e}(t) \cdot \eta_{i,o_{e,n}} = w_{e,o_{e,n}}(t), \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally $\mathcal{E} \subset \{\mathcal{E}_{IO}, \mathcal{E}_{IOO}\}$.

n indicates the n-th output of component e (arg: idx)

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class
- **idx** (*integer*) – Index to choose which output to select (from list of Outputs: O[e][idx])

oemof.solph.linear_constraints.**add_storage_balance**(*model*, *block*)

Constraint to build the storage balance in every timestep

The mathematical formulation of the constraint is as follows:

$$l_e(t) = l_e(t-1) \cdot (1 - C^{loss}(e)) - \frac{w_{e,o_e}(t)}{\eta_e^{out}} + w_{i_e,e}(t) \cdot \eta_e^{in} \qquad \forall e, \forall t \in [2, t_{max}]$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped in the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_S$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_constraints.**add_storage_charge_discharge_limits**(*model*, *block*)

Constraints that limit the discharge and charge power by the c-rate

Constraints are for investment models only.

The mathematical formulation for the constraints is as follows:

Discharge:

$$w_{e,o_e}(t) \leq (\overline{L}_e + \overline{l}_e^{add}) \cdot C_{o_e}^{rate} \qquad \forall e, \forall t$$

Charge:

$$w_{i_e,e}(t) \leq (\overline{L}_e + \overline{l}_e^{add}) \cdot C_{i_e}^{rate} \qquad \forall e, \forall t$$

oemof.solph.linear_constraints.**add_two_inputs_one_output_relation**(*model*, *block*)

Adds constraint for the input-output relation of a post heating transformer with two input flows.

Then the amount of all input flows multiplied with their efficiency will be the output flow. The efficiency of the each will be calculated so that the sum of the efficiency of both flows might be greater than one.

The mathematical formulation for the constraint is as follows:

$$w_{e,i_{e,1}}(t) \cdot \eta_{e,i_{e,1}}(t) + w_{e,i_{e,2}}(t) \cdot \eta_{e,i_{e,2}}(t) = w_{e,o_e}, \qquad \forall e, \forall t$$

$$w_{e,i_{e,1}}(t) = w_{e,i_{e,2}}(t) \cdot f(t), \quad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally: $\mathcal{E} \subset \mathcal{E}_{IIO}$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_mixed_integer_constraints module The module contains linear mixed integer constraints.

@author: Simon Hilpert (simon.hilpert@fh-flensburg.de)

oemof.solph.linear_mixed_integer_constraints.**add_minimum_downtime**(*model*, *block*)

Adds minimum downtime constraints for for components grouped inside *block.objs*.

The mathematical formulation for constraints is as follows:

$$(y_e(t-1) - y_e(t)) \cdot T_e^{min,off} \leq T_e^{min,off} - \sum_{\gamma=0}^{T_e^{min,off}-1} y_e(t+\gamma) \qquad \forall e, \forall t \in [2, t_{max} - t_{min,off}]$$

Extra constraints for last timesteps:

$$(y_e(t-1) - y_e(t)) \cdot T_e^{min,off} \leq T_e^{min,off} - \sum_{\gamma=0}^{t_{max}-t} y_e(t+\gamma) \qquad \forall e, \forall t \in [t_{max} - t_{min,off}, t_{max}]$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_mixed_integer_constraints.**add_minimum_uptime**(*model*, *block*)

Adds minimum uptime constraints for for components in *block*

The mathematical formulation for constraints is as follows:

$$(y_e(t) - y_e(t-1)) \cdot T_e^{min,on} \leq \sum_{\gamma=0}^{T_e^{min,on}-1} y_e(t+\gamma) \qquad \forall e, \forall t \in [2, t_{max} - t_{min,on}]$$

Extra constraint for the last timesteps:

$$(y_e(t) - y_e(t-1)) \cdot T_e^{min,on} \leq \sum_{\gamma=0}^{t_{max}-t} y_e(t+\gamma) \qquad \forall e, \forall t \in [t_{max} - t_{min,on}, t_{max}]$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_mixed_integer_constraints.**add_output_gradient_constraints**(*model*, *block*, *grad_direc='both'*)

Creates constraints to model the output gradient for milp-models.

If gradient direction is *positive*:

$$w_{e,o_{e,1}}(t) - w_{e,o_{e,1}}(t-1) \leq \overline{G}_{e_{o,1}}^{pos} + \underline{W}_{e,o_{e,1}} \cdot (1 - y_e(t))$$

If gradient direction is *negative*:

$$w_{e,o_{e,1}}(t-1) - w_{e,o_{e,1}}(t) \leq \overline{G}_{e_{o,1}}^{neg} + \underline{W}_{e,o_{e,1}} \cdot (1 - y_e(t-1))$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class
- **grad_direc** (*string*) – direction of gradient (“both”, “positive”, “negative”)

References

oemof.solph.linear_mixed_integer_constraints.**add_shutdown_constraints**(*model*, *block*)

Creates constraints to model the shut down of a component.

The mathematical formulation for the constraint is as follows:

$$y_e(t-1) - y_e(t) \leq z_e^{stop}(t), \qquad \forall e, \forall t$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_mixed_integer_constraints.**add_startup_constraints**(*model*, *block*)

Creates constraints to model the start up of a components.

The mathematical formulation of constraint is as follows:

$$y_e(t) - y_e(t-1) \leq z_e^{start}(t), \qquad \forall e, \forall t$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_mixed_integer_constraints.**add_variable_linear_eta_relation**(*model*, *block*)

Adds constraint for input-output relation for all units with variable efficiency grouped in *block.objs*.

The mathematical formulation for the constraint is as follows:

$$w_{i_e,e}(t) = y_e(t) \cdot c_1 + c_2 \cdot w_{e,o_{e,1}}(t), \qquad \forall e, \forall t$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class

oemof.solph.linear_mixed_integer_constraints.**maximum_starts_per_period**(*model*, *block*, *period=24*)

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data. Bounds are altered at model attributes (variables) of *model*
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class
- **period** (*array like*) – length of period

oemof.solph.linear_mixed_integer_constraints.**set_bounds**(*model*, *block*, *side='output'*)

Set upper and lower bounds via constraints.

The bounds are set with constraints using the binary status variable of the components. The mathematical formulation is as follows:

If side is *output*:

$$w_{e,o_{e,1}}(t) \leq \overline{W}_{e,o_{e,1}} \cdot y_e(t), \qquad \forall e, \forall t$$

$$w_{e,o_{e,1}}(t) \geq \underline{W}_{e,o_{e,1}} \cdot y_e(t), \qquad \forall e, \forall t$$

If side is *input*:

$$w_{i_e,e}(t) \leq \overline{W}_{i_e,e} \cdot y_e(t), \qquad \forall e, \forall t$$

$$w_{i_e,e}(t) \geq \underline{W}_{i_e,e} \cdot y_e(t), \qquad \forall e, \forall t$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data. Bounds are altered at model attributes (variables) of *model*
- **block** (*SimpleBlock()*) – block to group all constraints and variables etc., block corresponds to one oemof base class
- **side** (*string*) – string to select on which side the bounds should be set (*ìnput*, *output*)

oemof.solph.objective_expressions module The module contains different objective expression terms.

@author: Simon Hilpert (simon.hilpert@fh-flensburg.de)

oemof.solph.objective_expressions.**add_capex**(*model*, *block*, *ref='output'*)

Add capital expenditure to linear objective.

If reference is *output* (e.g. powerplants):

If reference is *capacity* (e.g. storages):

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class
- **ref** (*string*) – string to check if capex is referred to capacity (storage) or output (e.g. powerplant)

Returns

Return type Expression

oemof.solph.objective_expressions.**add_curtailment_costs**(*model*, *block=None*)

Cost term for dispatchable sources in linear objective.

$$\sum_e \sum_t w_e^{cut}(t) \cdot C_e^{cut}$$

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class

Returns

Return type Expression

oemof.solph.objective_expressions.**add_excess_slack_costs**(*model*, *block=None*)
Artificial cost term for excess slack variables.

$$\sum_e \sum_t EXCESS_e(t) \cdot C_e^{excess}$$

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock() block : SimpleBlock()*) – block to group all objects corresponding to one oemof base class

Returns

Return type Expression

oemof.solph.objective_expressions.**add_input_costs**(*model*, *block*)
Adds costs for usage of input (fuel, elec, etc.) if not included in opex

$$\sum_e \sum_t w_{i_e,e}(t) \cdot C_{i_e,e}(t)$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class

Returns

Return type Expression

oemof.solph.objective_expressions.**add_opex_fix**(*model*, *block*, *ref=None*)
Fixed operation expenditure term for linear objective function.

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class
- **ref** (*string*) – string to check if capex is referred to capacity (storage) or output (e.g. powerplant)

Returns

Return type Expression

oemof.solph.objective_expressions.**add_opex_var**(*model*, *block*, *ref='output'*, *idx=0*)
Variable operation expenditure term for linear objective function.

If reference of opex is *output*:

$$\sum_e \sum_t w_{e,o_{e,1}}(t) \cdot C_{e,o_{e,1}}(t)$$

If reference of opex is *input*:

$$\sum_e \sum_t w_{i_e,e}(t) \cdot C_{i_e,e}(t)$$

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class
- **ref** (*string*) – Reference side on which opex are based on (e.g. powerplant MWhth -> input or MWhel -> output)
- **idx** (*int*) – Index to select output/input of list of inputs/ouputs if multiple i/o exist

Returns

Return type Expression

oemof.solph.objective_expressions.**add_ramping_costs**(*model*, *block*, *grad_direc='positive'*)

Add gradient costs for components to linear objective expression.

$$\sum_e \sum_t g_e^{pos}(t) \cdot C_e^{g,neg}$$

$$\sum_e \sum_t g_e^{neg}(t) \cdot C_e^{g,pos}$$

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class
- **grad_direc** (*string*) – direction of gradient for which the costs are added to the objective expression

Returns

Return type Expression

oemof.solph.objective_expressions.**add_revenues**(*model*, *block*, *ref='output'*, *idx=0*)

Revenue term for linear objective function.

$$\sum_e \sum_t w_{e,o_{e,1}}(t) \cdot R_{e,o_{e,1}}(t)$$

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class
- **ref** (*string*) – Reference side for revnues ('output' or 'input' Note: 'input' not defined)
- **idx** (*integer*) – Integer indicating which output from list to select if entity has multiple outputs

Returns

Return type Expression

oemof.solph.objective_expressions.**add_shortage_slack_costs**(*model*, *block=None*)

Artificial cost term for shortage slack variables.

$$\sum_e \sum_t SHORTAGE_e(t) \cdot C_e^{shortage}$$

With $e \in E$ and E beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class

Returns

Return type Expression

oemof.solph.objective_expressions.**add_shutdown_costs**(*model*, *block*)

Adds shutdown costs for components to objective expression

$$\sum_e \sum_t z_e^{stop}(t) \cdot C_e^{stop}$$

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class

Returns

Return type Expression

oemof.solph.objective_expressions.**add_startup_costs**(*model*, *block*)

Adds startup costs for components to objective expression

$$\sum_e \sum_t z_e^{start}(t) \cdot C_e^{start}$$

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class

Returns

Return type Expression

oemof.solph.objective_expressions.**linearized_invest_costs**(*model*, *block*, *ref*)

This functionality add linearized costs with sos2-constraint.

The capex attribute of objects in 'block.objs'- needs to be a list of tuples with interpolation points. Every first element is the absolut value of investment for the corresponding size of investment in the second element of the tuple.

e.g. capex = [(10, 20), (20, 15),..]

Parameters

- **model** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class
- **ref** (*string*) – string to check if capex is referred to capacity (storage) or output (e.g. powerplant)

oemof.solph.optimization_model module @contact Simon Hilpert (simon.hilpert@fh-flensburg.de)

class oemof.solph.optimization_model.**OptimizationModel**(*energysystem*, *loglevel=20*)

Bases: pyomo.core.base.PyomoModel.ConcreteModel

Create Pyomo model of the energy system.

Parameters

- **entities** (*list with all entity objects*) –
- **timesteps** (*list with all timesteps as integer values*) –
- **options** (*nested dictionary with options to set.*) –

build_bus_constraints(*bbt*)

build_component_constraints(*cbt*)

default_assembler(*block*)

Method for setting optimization model objects for blocks

Parameters

- **self** (*OptimizationModel() instance*) –
- **block** (*SimpleBlock()*) –

edges(*components*)

Method that creates a list with all edges for the objects in components.

Parameters

- **self** –
- **components** (*list of component objects*) –

Returns edges

Return type list with tupels that represent the edges

objective_assembler(*objective_options*)

calls functions to add predefined objective functions

Parameters

- **self** (*OptimizationModel() instance*) –
- **objective_options** (*dict*) – dictionary with infos about objects. key "function" hold function to build objective

results()

Returns a nested dictionary of the results of this optimization model.

The dictionary is keyed by the *Entities* of the optimization model, that is *om.results()[s][t]* holds the time series representing values attached to the edge (i.e. the flow) from *s* to *t*, where *s* and *t* are instances of *Entity*.

Time series belonging only to one object, like e.g. shadow prices of commodities on a certain *Bus*, dispatch values of a *DispatchSource* or storage values of a *Storage* are treated as belonging to an edge looping from the object to itself. This means they can be accessed via *om.results()[object][object]*.

Object attributes holding optimization results, like e.g. *add_cap* for storage objects, can be accessed like this:

```
om.results()[s].add_cap()
```

where *s* is a storage object.

The value of the objective function is stored under the `om.results().objective` attribute.

Note that the optimization model has to be solved prior to invoking this method.

solve(***kwargs*)

Method that takes care of the communication with the solver to solve the optimization model.

Parameters

- **self** (*pyomo.ConcreteModel() object*) –
- ****kwargs** (*keywords*) – Possible keys can be set see below
- **string** (*solver*) – solver to be used e.g. "glpk","gurobi","cplex"
- **debug** (*boolean*) – If True model is solved in debug mode. lp-file is written.
- **duals** (*boolean*) – If True, duals and reduced costs are imported from the solver results
- **verbose** (*boolean*) – If True informations are printed
- **solver_io** (*string*) – pyomo solver interface file format: "lp","python","nl", etc.
- **solve_kwargs** (*dict*) – Other arguments for the pyomo.opt.SolverFactory.solve() method Example : {"solver_io":"lp"}
- **solver_cmdline_options** (*dict*) – Dictionary with command line options for solver Examples: {"mipgap":"0.01"} results in "–mipgap 0.01" {"interior":" "} results in "–interior"

Returns self

Return type solved pyomo.ConcreteModel() instance

update_objective(*objective_options=None*)

Updates the objective function with new parameters from entities

Parameters

- **self** (*OptimizationModel() instance*) –
- **objective_options** (*dict*) – dictionary with infos about objects. key "function" hold function to build objective

write_lp_file(*path=None*, *filename='problem.lp'*)

oemof.solph.optimization_model.**assembler**(*e*, *om*, *block*)

Assemblers are the functions adding constraints to an optimization model for a set of objects.

This is the most general form of assembler function, called only if no other, more specific assemblers have been found. Since we don't know what to do in this case, we can only throw a `TypeError`.

Parameters

- **entity** (*An object. Only used to figure out which assembler function to call*) – by dispatching on its *type*. Not used otherwise. It's a good idea to set this to *None* if the function is called directly via `assembler.registry`.
- **om** (*The optimization model. Should be an instance of*) – `pyomo.ConcreteModel`.
- **block** (*A pyomo block.*) –

Returns **om** – bus balances.

Return type The optimization model passed in as an argument, with additional

oemof.solph.predefined_objectives module This module contains predefined objectives that can be used to model energy systems.

@author: Simon Hilpert (simon.hilpert@fh-flensburg.de)

oemof.solph.predefined_objectives.**minimize_cost**(*self*, *cost_objects=None*, *revenue_objects=None*)

Builds objective function that minimises the total costs.

Costs included are: opex_var, opex_fix, curtailment_costs (dispatch sources), annualised capex (investment components)

Parameters

- **self** (*pyomo model instance*) –
- **cost_blocks** (*array like*) – list containing classes of objects that are included in cost terms of objective function
- **revenue_blocks** (*array like*) – list containing classes of objects that are included in revenue terms of objective function

oemof.solph.pyomo_fastbuild module @author: Simon Hilpert

oemof.solph.pyomo_fastbuild.**l_constraint**(*model*, *name*, *constraints*, **args*)

A replacement for pyomo's Constraint that quickly builds linear constraints.

Instead of: model.constraint_name = Constraint(index1,index2,...,rule=f)

Call: l_constraint(model, constraint_name, constraints, index1, index2,...)

This is a copy of the code from the python module pypsa. Thanks to Tom Brown / Jonas Hoersch for implementing it!

Parameters

- **model** (*pyomo.ConcreteModel() / SimpleBlock() instance*) – pyomo model or block with constructed components
- **name** (*string*) – Name of constraint to be constructed
- **constraints** (*dict*) – Constraints is a dictionary of constraints of the form:

  constraints[i] = [[(coeff1, var1), (coeff2, var2),...], sense, constant_term]

  sense is one of "==","<=",">=".

  constant_term is constant rhs of equation

  I.e. variable coefficients are stored as a list of tuples.
- ***args** – arguments passed to the pyomo.Constraint() class.

oemof.solph.pyomo_fastbuild.**mutate_variable**(*var=None*, *value=None*, *bounds=(0*, *None)*, *fix=False*, *index=None*)

Mutates existing pyomo variable, could be used to extend an emtpy Var() object or an existing indexed pyomo variable. It's possible to overwrite information for existing indices.

Parameters

- **var** (*pyomo.core.base.var.Var() object*) – pyomo variable to be extended

- **value** (*list*) – values of variable in list
- **bounds** (*list*) – lower and upper bounds of variable as tuples in list
- **index** (*type of var._index elements (most likely tuple)*) – list of indices for extension (Warning: if existing indices are used, data for for these indices will be overwritten.)
- **fix** (*list*) – list with boolean elements, if true, variable value is fixed, i.e. converted to parameter in opt-problem

Example

extend_variable(om.w, value=[10], bounds=[(0,10)], fix=[True], index=[('bel','sink',155)])

oemof.solph.variables module Module contains variable definitions and constraint to bound variables (e.g. for investement).

@author: Simon Hilpert (simon.hilpert@fh-flensburg.de)

oemof.solph.variables.**add_binary**(*model*, *block*, *relaxed=False*)

Creates all status variables (binary) for *block.objs*

Status variable indicates if a unit is turned on or off. E.g. if a transformer is switched on/off -> y=1/0

Parameters

- **model** (*pyomo.ConcreteModel()*) – A pyomo-object to be solved containing all Variables, Constraints, Data Variables are added as attributes to the *model*
- **objs** (*array_like (list)*) – all components for which the status variable is created
- **relaxed** (*boolean*) – If True "binary" variables will be created as continuous variables with bounds of 0 and 1.

oemof.solph.variables.**add_continuous**(*model*, *edges*)

Adds all variables corresponding to the edges of the bi-partite graph for all timesteps.

The function uses the pyomo class *Var()* to create the optimization variables. As index-sets the provided edges of the graph (in general all edges) and the defined timesteps are used. The following variables are created: Variables for all the edges (If specific components such as disptach sources and storages exist.)

Parameters

- **model** (*OptimizationModel() instance*) – A object to be solved containing all Variables, Constraints, Data Variables are added as attributes to the *model*
- **edges** (*array_like (list)*) – *edges* will be a list containing tuples representing the directed edges of the graph by using unique ids of components and buses. e.g. [('coal', 'pp_coal'), ('pp_coal', 'b_el'),...]

oemof.solph.variables.**set_bounds**(*model*, *block*, *side='output'*)

Sets bounds for variables that represent the weight of the edges of the graph if investment models are calculated.

For investment models upper and lower bounds will be modeled via additional constraints. The mathematical description for the constraint is as follows:

If side is *output*

$$w_{e,o_{e,1}}(t) \leq \overline{W}_{e,o_{e,1}} \qquad \forall e, \forall t$$

With investment:

$$w_{e,o_{e,1}}(t) \leq \overline{W}_{e,o_{e,1}} + \overline{w}_{o_e}^{add}, \qquad \forall e, \forall t$$

If side is *input*:

$$w_{i_e,e}(t) \leq \overline{W}_{i_e,e} \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data Constraints are added as attributes to the *model*
- **block** (*SimpleBlock()*) –
- **side** (*string*) – Side of component for which the bounds are set ('input' or 'output')

oemof.solph.variables.**set_fixed_sink_value**(*model*, *block*)

Setting a value und fixes the variable of input.

The mathematical formulation is as follows:

$$w_{i_e,e}(t) = V_e(t), \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally $\mathcal{E} \subset \mathcal{E}_I$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data Attributes are altered of the *model*
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class

oemof.solph.variables.**set_outages**(*model*, *block*, *outagetype='period'*, *side='output'*)

Fixes component input/output to zeros for modeling outages.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data.
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class
- **outagetype** (*string*) – String indicates how to model outages of component. If outages is scalar 'period' yield one timeblock where component is off, while 'random_days' will sample random days over the timehorizon where component will forced to be offline.
- **side** (*string*) – Side of component to fix to zero: 'output', 'input'.

oemof.solph.variables.**set_storage_cap_bounds**(*model*, *block*)

Alters/sets upper and lower bounds for variables that represent the absolut state of charge e.g. filling level of a storage component.

For investment models upper and lower bounds will be modeled via additional constraints. The mathematical description for the constraint is as follows:

$$\underline{L}_e \leq l_e(t) \leq \overline{L}_e, \qquad \forall e, \forall t$$

If investment:

$$l_e(t) \leq \overline{L}_e + \overline{l}_e^{add}, \qquad \forall e, \forall t$$

With $e \in \mathcal{E}$ and $\mathcal{E}$ beeing the set of unique ids for all entities grouped inside the attribute *block.objs*.

Additionally $\mathcal{E} \subset \mathcal{E}_S$.

Parameters

- **model** (*OptimizationModel() instance*) – An object to be solved containing all Variables, Constraints, Data Bounds are altered at model attributes (variables) of *model*
- **block** (*SimpleBlock()*) – block to group all objects corresponding to one oemof base class

Module contents The solph-package contains funtionalities for creating and solving an optimizaton problem. The problem is created from oemof base classes. Solph depend on pyomo.

oemof.tools package

Submodules

oemof.tools.config module Created on Fri Sep 5 12:26:40 2014

module-author steffen

filename config.py

This module provides a highlevel layer for reading and writing config files. There must be a file called "config.ini" in the root-folder of the project. The file has to be of the following structure to be imported correctly.

this is a comment

the filestructure is like:

[netCDF]

RootFolder = c://netCDF

FilePrefix = cd2-

[mySQL]

host = localhost

user = guest

password = root

database = znes

[SectionName]

OptionName = value

Option2 = value2

`oemof.tools.config.get`(*section*, *key*)

returns the value of a given key of a given section of the main config file.

Parameters

- **section** (*str.*) – the section.

- **key** (*str.*) – the key.

Returns the value which will be casted to float, int or boolean. if no cast is successfull, the raw string will be returned.

oemof.tools.config.**init**()

oemof.tools.config.**main**()

oemof.tools.config.**set**(*section*, *key*, *value*)
sets a value to a [section] key - pair. if the section doesn't exist yet, it will be created.

Parameters

- **section** (*str.*) – the section.
- **key** (*str.*) – the key.
- **value** (*float, int, str.*) – the value.

oemof.tools.create_components module
oemof.tools.create_components.**instant_flow_heater**(*bus_low*, *bus_high*)

oemof.tools.entities_from_csv module @author: Simon Hilpert simon.hilpert@fh-flensburg.de

oemof.tools.entities_from_csv.**add_bus**(*row*, ***kwargs*)
Adds bus object ot list of busses. The function is used by apply() method of pandas data frames.

Parameters

- **row** (*row of pandas.DataFrame()*) –
- **busses** (*list*) – list of existing busses
- **busprices** (*pandas.DataFrame()*) – data frame containting the values for the busprices. Column index is the uid of bus to create.

oemof.tools.entities_from_csv.**add_chp**(*row*, ***kwargs*)
Adds chps objects ot list of transformers. The function is used by apply() method of pandas data frames.

Parameters

- **row** (*row of pandas.DataFrame()*) –
- **busses** (*list*) – list of existing busses
- **transformers** (*list*) – list of transformers where object are appended

oemof.tools.entities_from_csv.**add_sink**(*row*, ***kwargs*)
Adds sink object ot list of transformers. The function is used by apply() method of pandas data frames.

Parameters

- **row** (*row of pandas.DataFrame()*) –
- **busses** (*list*) – list of existing busses
- **sink** (*list*) – list of sinks where object are appended
- **sinkvalues** (*pandas.DataFrame()*) – data frame containting the values for the sink column index is uid of sink

oemof.tools.entities_from_csv.**add_source**(*row*, ***kwargs*)
Adds source object ot list of source. The function is used by apply() method of pandas data frames.

Parameters

- **row** (*row of pandas.DataFrame()*) –
- **busses** (*list*) – list of existing busses
- **sink** (*list*) – list of sources where object are appended
- **sourcevalues** (*pandas.DataFrame()*) – data frame containting the values for the source column index is uid of source to create.

oemof.tools.entities_from_csv.**add_storage**(*row*, ***kwargs*)
Adds storage objects ot list of storages. The function is used by apply() method of pandas data frames.

Parameters

- **row** (*row of pandas.DataFrame()*) –
- **busses** (*list*) – list of existing busses
- **transformers** (*list*) – list of transformers where object are appended

oemof.tools.entities_from_csv.**add_transformer**(*row*, ***kwargs*)
Adds transformer objects ot list of transformers. The function is used by apply() method of pandas data frames.

Parameters

- **row** (*row of pandas.DataFrame()*) –
- **busses** (*list*) – list of existing busses
- **transformers** (*list*) – list of transformers where object are appended

oemof.tools.entities_from_csv.**add_transport**(*row*, ***kwargs*)
Adds transpport objects ot list of transports. The function is used by apply() method of pandas data frames.

oemof.tools.entities_from_csv.**entities_from_csv**(*files*, *entities_dict=None*)
Creates 'oemof-objects' from csv files by the use of pandas dataframes

Parameters

- **files** (*dict*) – dictionary containing the paths to files for object creation. Keys are: 'transformers','busses','storages','chps', 'sources', 'sourcevalues', 'sinks', 'sinkvalues', 'busprices'
- **entities_dict** (*dict*) – dictionary containing lists of oemof base class objects

oemof.tools.helpers module Created on Mon Aug 17 11:08:15 2015

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

All special import should be in try/except loops to avoid import errors.

oemof.tools.helpers.**abbreviation_of_state**(*statename*)
Get the abbreviation of a german state name or the other way round.

Parameters **statename** (*string*) – Fullname of the state or its abbreviation

Returns Fullname of the state or its abbreviation, depending on the input.

Return type string

oemof.tools.helpers.**create_basic_dataframe**(*year*, ***kwargs*)
Giving back a DataFrame containing weekdays and optionally holidays for the given year.

Parameters

- **year** (*the year for which dataframe should be created*) –

- **Parameters** (*Optional*) –
- **----------** –
- **holidays** (*array with information for every hour of the year, if holiday or*) – not (0: holiday, 1: no holiday)

Returns pandas.DataFrame

Return type DataFrame with a time index

Notes

Using Pandas > 0.16

oemof.tools.helpers.**cut_lists**(*list_a*, *list_b*)
Returns a list with the values of list_a AND list_b.

oemof.tools.helpers.**dict2pickle**(*dic*, *filename=None*, *path=None*)
Dumping a python dictionary into a pickle file.

oemof.tools.helpers.**dict2textfile**(*dic*, *filename=None*, *path=None*)
Writing a dictionary to textfile in a readable and clearly formatted way.

oemof.tools.helpers.**download_file**(*filename*, *url*)
Copy a file from the given url to the given filename.

oemof.tools.helpers.**extend_basic_path**(*subfolder*)

oemof.tools.helpers.**fetch_admin_from_coord_google**(*coord*)
Reverse geocoding using google.

Parameters **coord** (*array_like*) – Geo-coordinates in the order (longitude, latitude)

Returns Containing the name of the country and the name of the state if available

Return type list of strings

See also:

fetch_admin_from_coord_google()

Examples

```
>>> fetch_admin_from_coord_google((12.7, 51.8))
['DE', 'SA']
```

oemof.tools.helpers.**fetch_admin_from_coord_osm**(*coord*)
Reverse geocoding using osm.

Parameters **coord** (*array_like*) – Geo-coordinates in the order (longitude, latitude)

Returns Containing the name of the country and the name of the state if available

Return type list of strings

See also:

fetch_admin_from_coord_google()

Examples

```
>>> fetch_admin_from_coord_osm((12.7, 51.8))
['Deutschland', 'ST']
```

oemof.tools.helpers.**get_basic_path**()

oemof.tools.helpers.**get_fullpath**(*path*, *filename*)

oemof.tools.helpers.**get_german_holidays**(*year, place=[None, None, None], scope='legal'*)

Returns German holiday dates. Use the abbreviations for the german federal states like 'BY' for Bayern.

Parameters

- **year** (*int*) – Year of which the holidays are wanted
- **place** (*list of strings*) – List of names: [country, state, city]
- **scope** (*string*) – Type of holidays. So far only legal is possible.

Returns The keys are in the datetime format the values represent the names.

Return type dictionary

Notes

So far only the data of german holidays is available. Feel free to add other countries or to implement workalendar. https://github.com/novapost/workalendar

Examples

get_holidays(2015, place=['Germany', 'BE', 'Berlin'])

oemof.tools.helpers.**get_polygon_from_shp_file**(*file*)

A one-line summary that does not use variable names or the function name.

Several sentences providing an extended description. Refer to variables using back-ticks, e.g. *var*.

Parameters

- **var1** (*array_like*) – Array_like means all those objects – lists, nested lists, etc. – that can be converted to an array. We can also refer to variables like *var1*.
- **var2** (*int*) – The type above can either refer to an actual Python type (e.g. `int`), or describe the type of the variable in more detail, e.g. `(N,) ndarray` or `array_like`.
- **Long_variable_name** (*{'hi', 'ho'}, optional*) – Choices in brackets, default first when optional.

Returns

- *type* – Explanation of anonymous return value of type `type`.
- **describe** (*type*) – Explanation of return value named *describe*.
- **out** (*type*) – Explanation of *out*.

Other Parameters

- **only_seldom_used_keywords** (*type*) – Explanation
- **common_parameters_listed_above** (*type*) – Explanation

Raises `BadException` – Because you shouldn't have done that.

See also:

`otherfunc()` relationship (optional)

`newfunc()` Relationship (optional), which could be fairly long, in which case the line wraps here.

`thirdfunc()`, `fourthfunc()`, `fifthfunc()`

Notes

Notes about the implementation algorithm (if needed).

This can have multiple paragraphs.

You may include some math:

$$X(e^{j\omega}) = x(n)e^{-j\omega n}$$

And even use a greek symbol like $omega$ inline.

References

Cite the relevant literature, e.g. [1]. You may also cite these references in the notes section above.

Examples

These are written in doctest format, and should illustrate how to use the function.

```
>>> a=[1,2,3]
>>> print([x + 3 for x in a])
[4, 5, 6]
>>> print("a\nb")
a
b
```

`oemof.tools.helpers.`**`mkdir_if_not_exist`**(*path*)
Creates directory if not exist

`oemof.tools.helpers.`**`pickle2dict`**(*filename=None*, *path=None*)
Reading a python dictionary from a pickle file.

`oemof.tools.helpers.`**`remove_from_list`**(*orig_list*, *remove_list*)
Removes the values inside the remove_list from the orig_list.

`oemof.tools.helpers.`**`time_logging`**(*start*, *text*, *logging_level='debug'*)
Logs the time between the given start time and the actual time. A text and the debug level is variable.

Uwe Krien (uwe.krien@rl-institut.de)

Parameters start : start time : float text : text to describe the log : string

Keyword arguments logging_level : logging_level [default='debug'] : string

[1] O. McNoleg, "The integration of GIS, remote sensing, expert systems and adaptive co-kriging for environmental habitat modelling of the Highland Haggis using object-oriented, fuzzy-logic and neural-network techniques," Computers & Geosciences, vol. 22, pp. 585-588, 1996.

oemof.tools.helpers.**unique_list**(*seq*)
Returns a unique list without preserving the order

oemof.tools.helpers.**unlist**(*val*)
Returns single value if a single value in a list is given

oemof.tools.holiday module Holiday information for Germany.

oemof.tools.holiday.**fetch_admin_from_latlon**(*lat=None*, *lon=None*, *coord=None*)
Receive reverse geocoded information from lat/lon point :rtype : dict :param lat: latitude :param lon: longitude :param zoom: detail level of information :return: list: [country, state]

oemof.tools.holiday.**get_abbreviation_of_state**(*statename*)

oemof.tools.holiday.**get_german_holidays**(*year, place=[None, None, None], scope='legal', _=<class 'str'>*)
Returns German holiday dates. Use the s for the german federal states like 'Germany/BY ' for Bayern. : Example: get_holidays(2015, place=['Germany', 'BE', 'Berlin'])

oemof.tools.logger module Author: Uwe Krien (uwe.krien@rl-institut.de) Changes by: Responsibility: Uwe Krien (uwe.krien@rl-institut.de)

oemof.tools.logger.**check_git_branch**()
Passes the used brance and commit to the logger

oemof.tools.logger.**check_version**()
Returns the actual version number of the used oemof version.

oemof.tools.logger.**define_logging**(*inifile='logging.ini'*, *basicpath=None*, *subdir='log_files'*)
Initialise the logger using the logging.conf file in the local path.

Several sentences providing an extended description. Refer to variables using back-ticks, e.g. *var*.

Parameters

- **inifile** (*string, optional (default: logging.ini)*) – Name of the configuration file to define the logger. If no ini-file exist a default ini-file will be downloaded from 'http://vernetzen.uni-flensburg.de/~git/logging_default.ini' and used.
- **basicpath** (*string, optional (default: '.oemof' in HOME)*) – The basicpath for different oemof related informations. By default a ".oemof" folder is created in your home directory.
- **subdir** (*string, optional (default: 'log_files')*) – The name of the subfolder of the basicpath where the log-files are stored.

Notes

By default the INFO level is printed on the screen and the debug level in a file, but you can easily configure the ini-file. Every module that wants to create logging messages has to import the logging module. The oemof logger module has to be imported once to initialise it.

Examples

To define the default logge you have to import the python logging library and this function. The first logging message should be the path where the log file is saved to.

```
>>> import logging
>>> from oemof.tools import logger
>>> logger.define_logging()
17:56:51-INFO-Path for logging: /HOME/.oemof/log_files
...
>>> logging.debug("Hallo")
```

Module contents

Module contents

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

O

A

P

R

S

T

U

V

W

Y