oemof Documentation

Release

oemof-Team

Mar 28, 2017

Contents

Contents:

CHAPTER 1

Getting started

Oemof stands for "Open Energy System Modelling Framework" and provides a free, open source and clearly documented toolbox 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.

Everybody is welcome to use and/or develop oemof. Read our *Why should I contribute?* section.

Documentation

Full documentation can be found at readthedocs. Use the project site of readthedocs to choose the version of the documentation. To get the latest news visit and follow our website.

Installing oemof

If you have a working Python3 environment, use pypi to install the latest oemof version.

```
pip install oemof
```

For more details have a look at *Installation and setup*.

The packages **feedinlib**, **demandlib** and **oemof.db** have to be installed separately. See section *Using oemof* for more details about all oemof packages.

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

Structure of the oemof cosmos

Oemof packages are organised in different levels. The basic oemof interfaces are defined by the core libraries (network). The next level contains libraries that depend on the core libraries but do not provide interfaces to other oemof libraries (solph, outputlib). The third level are libraries that do not depend on any oemof interface and therefore can be used as stand-alone application (demandlib, feedinlib). Together with some other recommended projects (pvlib, windpowerlib) the oemof cosmos provides a wealth of tools to model energy systems. If you want to become part of it, feel free to join us.

Examples

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

You can execute examples of solph applications (*Solph Examples*) from command-line by

```
oemof_examples
```

Further reading in the '*Run examples to check the installation*' section.

Keep in touch

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

License

CHAPTER 2

About oemof

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

The idea of an open framework

The Open Energy System Modelling Framework has been developed for the modelling and analysis of energy supply systems considering power and heat as well as prospectively mobility.

Oemof has been implemented in Python and uses several Python packages for scientific applications (e.g. mathematical optimisation, network analysis, data analysis), optionally in combination with a PostgreSQL/PostGIS database. It offers a toolbox of various features needed to build energy system models in high temporal and spatial resolution. For instance, the wind energy feed-in in a model region can be modelled based on weather data, 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 different libraries. For the communication between these libraries different interfaces are provided. The oemof libraries and their modules are used to build what we call an 'application' (app) which depicts a concrete energy system model or a subprocess of this model. Generally, applications can be developed highly individually by the use of one or more libraries depending on the scope and purpose. The following image illustrates the typical application building process.

It gets clear that applications can be build flexibly using different libraries. Furthermore, single components of applications can be substituted easily if different functionalities are needed. This allows for individual application development and provides all degrees of freedom to the developer which is particularly relevant in environments such as scientific work groups that often work spatially distributed.

Among other applications, the apps 'renpassG!S' and 'reegis' are currently developed based on the framework. 'renpassG!S' 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. Another application is 'HESYSOPT' which has been desined to simulate combined heat and power systems with MILP on the component level. These three examples show that the modular approach of the framework allows applications with very different objectives.

Application Examples

Some applications are publicly available and continiously developed in these locations:

- renpassG!S
- HESYSOPT
- reegis_HP
- eos

More examples and a screenshot gallery can be found on oemof's official homepage.

Why are we developing oemof?

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 or cross-border interaction between national energy systems. Besides, energy system models are often developed for a certain application and cannot (or only with great effort) be adjusted to other requirements.

The Center for Sustainable Energy Systems (ZNES) Flensburg together with the Reiner Lemoine Institute (RLI) in Berlin and the Otto-von-Guericke-University of Magdeburg (OVGU) are developing the Open Energy System Modelling Framework (oemof) to address these problems by offering a free, open and clearly documented framework for energy system modelling. 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.

Why should I contribute?

- You do not want to start at the very beginning. - You are not the first one, who wants to set up a energy system model. So why not start with existing code?

- You want your code to be more stable. - If other people use your code, they may find bugs or will have ideas to improve it.
- Tired of 'write-only-code'. - Developing as part of a framework encourages you to document sufficiently, so that after years you may still understand your own code.
- You want to talk to other people when you are deadlocked. - People are even more willing to help, if they are interested in what you are doing because they can use it afterwards.
- You want your code to be seen and used. We try to make oemof more and more visible to the modelling community. Together it will be easier to increase the awareness of this framework and therefore for your contribution.

We know, sometimes it is difficult to start on an existing concept. It will take some time to understand it and you will need extra time to document your own stuff. But once you understand the libraries you will get lots of interesting features, always with the option to fit them to your own needs.

Just contact us, if you have any questions!

Join oemof with your own approach or project

Oemof is designed as a framework and there is a lot of space for own ideas or own libraries. No matter if you want a heuristic solver library or different linear solver libraries. You may want to add tools to analyse the results or something we never heard of. You want to add a GUI or your application to be linked to. We think, that working together in one framework will increase the probability that somebody will use and test your code (see *Why should I contribute?*).

Interested? Together we can talk about how to transfer your ideas into oemof or even integrate your code. Maybe we just link to your project and try to adept the API for a better fit in the future.

CHAPTER 3

Installation and setup

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

Linux

Having Python 3 installed

As oemof is designed as a Python package it is mandatory to have Python 3 installed. It is highly recommended to use a virtual environment. See this tutorial for more help or see the sections below. If you already have a Python 3 environment you can install oemof using pip:

```
pip install oemof
```

If you do not yet have pip installed in your python environment, see section *Additional Python packages* below for further help.

Using Linux repositories to install Python

Most Linux distributions will have Python 3 in their repository. Use the specific software management to install it. If you are using Ubuntu/Debian try executing the following code in your terminal:

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

You can also download different versions of Python via https://www.python.org/downloads/.

Using Virtualenv (community driven)

Skip the steps you have already done. Check your architecture first (32/64 bit).

1. Install virtualenv using the package management of your Linux distribution, pip install or install it from source (see virtualenv documentation)
2. Open terminal to create and activate a virtual environment by typing:

```
virtualenv -p /usr/bin/python3 your_env_name
source your_env_name/bin/activate
```

3. In terminal type: `pip install oemof`
4. Install a *Solver* if you want to use solph and execute the solph examples (See *Run examples to check the installation*) to check if the installation of the solver and oemof was successful

Warning: If you have an older version of virtualenv you should update pip `pip install --upgrade pip`.

Using Anaconda

Skip the steps you have already done. Check your architecture first (32/64 bit).

1. Download latest Anaconda for Python 3.x (64 or 32 bit)
2. Install Anaconda
3. Open terminal to create and activate a virtual environment by typing:

```
conda create -n yourenvname python=3.4
source activate yourenvname
```

4. In terminal type: `pip install oemof`
5. Install a *Solver* if you want to use solph and execute the solph examples (See *Run examples to check the installation*) to check if the installation of the solver and oemof was successful

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.

There are two common OpenSource solvers available (CBC, GLPK), while oemof recommends CBC (Coin-or branch and cut). But sometimes its worth comparing the results of different solvers.

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

Check the solver installation by executing the test_installation example (see *Run examples to check the installation*).

To learn how to install (other) solvers (Gurobi, Cplex...) have a look at the pyomo solver notes.

Additional 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 on Debian/Ubuntu try executing the following code in your terminal or use the software management of you Linux distribution:

```
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 (e.g. virtualenv):

```
pip3 install package_name
```

For further information on how to install Python modules check out https://docs.python.org/3/installing/index.html.

Windows

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 a working Python 3 environment you can install oemof by using pip. Run the following code in the command window of your python environment:

```
pip install oemof
```

If pip is not part of your python environment, see section *Additional Python packages* below for further help or use WinPython/Anaconda (see below).

Using WinPython (community driven)

Skip the steps you have already done. Check your architecture first (32/64 bit)

1. Download latest WinPython for Python 3.x (64 or 32 bit)
2. Install WinPython
3. Open the 'WinPython Command Prompt' and type: `pip install oemof`
4. Install a *Windows Solver* if you want to use solph and execute the solph examples (See *Run examples to check the installation*) to check if the installation of the solver and oemof was successful

Using Anaconda

Skip the steps you have already done. Check your architecture first (32/64 bit)

1. Download latest Anaconda for Python 3.x (64 or 32 bit)
2. Install Anaconda
3. Open 'Anaconda Prompt' to create and activate a virtual environment by typing:

```
conda create -n yourenvname python=3.4
activate yourenvname
```

It is recommended to use python 3.4. Some users reported that oemof does not work with Windows + Anaconda + Python 3.5

4. In 'Anaconda Prompt' type: `pip install oemof`
5. Install a *Windows Solver* if you want to use solph and execute the solph examples (See *Run examples to check the installation*) to check if the installation of the solver and oemof was successful

Windows 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.

You do not have to install both solvers. Oemof recommends the CBC (Coin-or branch and cut) solver. But sometimes its worth comparing the results of different solvers (e.g. GLPK).

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

Check the solver installation by executing the test_installation example (see *Run examples to check the installation*).

For commercial solvers (Gurobi, Cplex...) have a look at the pyomo solver notes.

Additional 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:

```
pip install package_name
```

For further information on how to install Python modules check out https://docs.python.org/3/installing/. Using pip all necessary packages are installed automatically. Have a look at the setup.py to see all requirements.

Mac OSX

Installation guidelines for Mac OS are still incomplete and not tested. As we do not have Mac users we could not test the following approaches, but they should work. If you are a Mac user please help us to improve this installation guide. Have look at the installation guide of Linux or Windows to get an idea what to do.

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.

Virtualenv: http://sourabhbajaj.com/mac-setup/Python/README.html

Anaconda: https://www.continuum.io/downloads#osx

You have to install a solver if you want to use solph and execute the solph examples (See *Run examples to check the installation*) to check if the installation of the solver and oemof was successful.

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

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

Run examples to check the installation

Run the examples to check the installation. From the command-line (or Anaconda Prompt / WinPython Command Prompt) execute:

```
oemof_examples <name-of-example> [-s <name-of-solver>]
```

You can choose from the list of examples

- test_installation
- storage_investment (solph)
- simple_dispatch (solph)
- csv_reader_investment (solph)
- flexible_modelling (solph)
- csv_reader_dispatch (solph)

Test the installation and the installed solver:

```
oemof_examples test_installation
```

Execute an example with different solver (default: 'cbc').

```
oemof_examples simple_least_costs
oemof_examples simple_least_costs -s glpk
```

If you want to run solph examples you need to have a solver installed (recommended: cbc), see the "*Solver*" or "*Windows Solver*" section. To get more information about the solph examples see the "*Solph Examples*" section.

CHAPTER 4

Using oemof

Oemof is a framework and even though it is in an early stage it already provides useful tools to model energy systems. To model an energy system you have to write your own application in which you combine the oemof libraries for you specific task. The example section shows how an oemof application may look like.

Current oemof libraries

- *oemof-network*
- *oemof-solph*
- *oemof-outputlib*
- *feedinlib*
- *demandlib*

oemof-network

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

The code of the example above:

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

# create the energy system
es = EnergySystem()

# create bus 1
```

```
bus_1 = Bus(label="bus_1")

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

# create sink 1
Sink(label='sink_1', inputs={bus_1: []})

# create sink 2
Sink(label='sink_2', inputs={bus_2: []})

# create source
Source(label='source', outputs={bus_1: []})

# create transformer
Transformer(label='transformer', inputs={bus_1: []}, outputs={bus_2: []})
```

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

oemof-solph

The *oemof-solph* library is part of the oemof installation. Solph is designed to create and solve linear or mixed-integer linear optimization problems. It is based on optimization modelling language pyomo.

To use solph at least one linear solver has to installed on you system. See the pyomo installation guide to learn which solvers are supported. Solph is tested with the open source solver *cbc* and the *gurobi* solver (free for academic use). The open *glpk* solver recently showed some odd behaviour.

The formulation of the energy system is based on the oemof-network library but contains additional components such as storages. Furthermore the network class are enhanced with additional parameters such as efficiencies, bounds, cost and more. See the API documentation for more details. Try the solph examples to learn how to build a linear energy system.

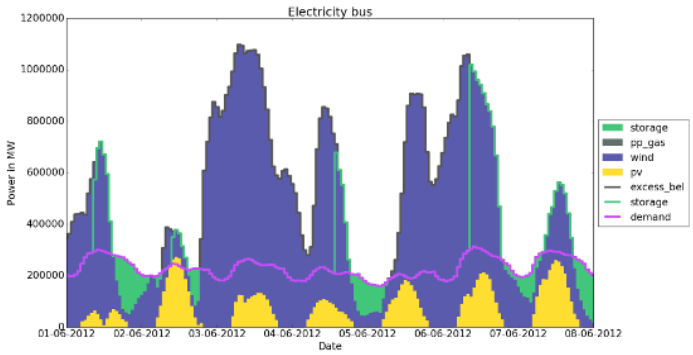
oemof-outputlib

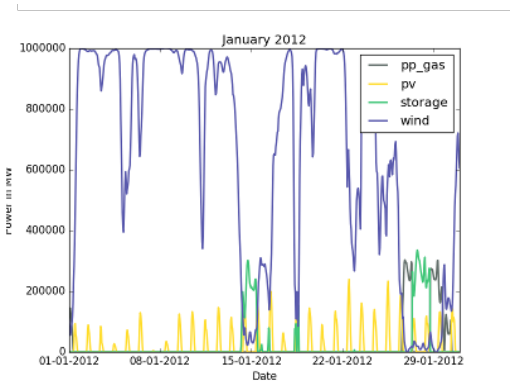
The *oemof-outputlib* library is part of the oemof installation. The outputlib presents the results of an optimisation as a pandas MultiIndex DataFrame. This makes it easy to process or plot the results using the capabilities of the pandas library.

The following code returns the output (flow from power plant to electricity bus) of a power plant. The result is written to a csv file using the pandas i/o methods. To get the input of the power plant the type has to be changed to *from_bus*.

```
pp_gas = myresults.slice_by(obj_label='pp_gas', type='to_bus',
                            date_from='2012-01-01 00:00:00',
                            date_to='2012-12-31 23:00:00')
pp_gas.to_csv('pp_gas.csv')
```

Beside this the outputlib provides some basic plot methods to create nice plots. The oemof plot methods can be used additionally and can easily be combined with the plot capabilities of pandas and matplotlib.





feedinlib

The feedinlib library is not part of the oemof installation and has to be installed separately using pypi. At the current state the feedinlib can calculate the output from a wind and a pv power plant passing parameters describing the power plant and a weather data set.

```
my_weather = weather.FeedinWeather()
my_weather.read_feedinlib_csv(filename='weather.csv')

E126_power_plant = plants.WindPowerPlant(**enerconE126)
E126_feedin = E126_power_plant.feedin(weather=my_weather,
                                      installed_capacity=15000000)   # 15 MW

yingli_module = plants.Photovoltaic(**yingli210)
pv_feedin = yingli_module.feedin(weather=my_weather, number=30000)   # 30000 modules
```

See the documentation of the feedinlib for a full description of the library and the example above.

demandlib

The demandlib library is not part of the oemof installation and has to be installed separately using pypi. At the current state the demandlib can be used to create load profiles for elctricity and heat knowing the annual demand. See the documentation of the demandlib for examples and a full description of the library.

CHAPTER 5

Developing oemof

Here you find important notes for developing 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.

Install the developer version

To avoid problems make sure you have fully uninstalled previous versions of oemof. It is highly recommended to use a virtual environment. See this virtualenv tutorial for more help. Afterwards two steps are necessary to install the developer version:

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

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

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

Documentation

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

Collaboration with pull requests

To collaborate use the pull request functionality of github.

How to create a pull request

- Fork the oemof repository
- Create a pull request and describe what you will do and why
- Change, add or remove code
- Read the *Style guidelines* and follow them
- Add new tests according to what you have done
- Add/change the documentation (new feature, API changes ...)
- Add a whatsnew entry and your name to Contributors
- Check if all tests still work including the example tests

Tests

Run the following test before pushing a successful merge.

```
nosetests -w "/path/to/oemof" --with-doctest
python3 path/to/oemof/examples/oemof_full_check.py
```

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.

Docstrings

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

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.

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.

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
```

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)

Using git

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/*`

Commit message

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

Issue-Management

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

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.

CHAPTER 6

What's New

These are new features and improvements of note in each release

Releases

v0.1.3 (March 28, 2017)

Bug fixes

- fix examples (issue #298)

Documentation

- Adapt installation guide.

Contributors

- Uwe Krien
- Stephan Günther

v0.1.2 (March 27, 2017)

New features

- Revise examples - clearer naming, cleaner code, all examples work with cbc solver (issue #238, issue #247).
- Add option to choose solver when executing the examples (issue #247).
- Add new transformer class: VariableFractionTransformer (child class of LinearTransformer). This class represents transformers with a variable fraction between its output flows. In contrast to the LinearTransformer by now it is restricted to two output flows.(issue #248)
- Add new transformer class: N1Transformer (counterpart of LinearTransformer). This class allows to have multiple inputflows that are converted into one output flow e.g. heat pumps or mixing-components.
- Allow to set addtional flow attributes inside NodesFromCSV in solph inputlib
- Add economics module to calculate investment annuities (more to come in future versions)
- Add module to store input data in multiple csv files and merge by preprocessing
- Allow to slice all information around busses via a new method of the ResultsDataFrame
- Add the option to save formatted balances around busses as single csv files via a new method of the ResultsDataFrame

Documentation

- Improve the installation guide.

Bug fixes

- Allow conversion factors as a sequence in the CSV reader

Other changes

- Speed up constraint-building process by removing unnecessary method call
- Clean up the code according to pep8 and pylint

Contributors

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

v0.1.1 (November 2, 2016)

Hot fix release to make examples executable.

Bug fixes

- Fix copy of default logging.ini (issue #235)
- Add matplotlib to requirements to make examples executable after installation (issue #236)

Contributors

- Guido Plessmann
- Uwe Krien

v0.1.0 (November 1, 2016)

The framework provides the basis for a great range of different energy system model types, ranging from LP bottom-up (power and heat) economic dispatch models with optional investment to MILP operational unit commitment models.

With v0.1.0 we refactored oemof (not backward compatible!) to bring the implementation in line with the general concept. Hence, the API of components has changed significantly and we introduced the new 'Flow' component. Besides an extensive grouping functionality for automatic creation of constraints based on component input data the documentation has been revised.

We provide examples to show the broad range of possible applications and the frameworks flexibility.

API changes

- The demandlib is no longer part of the oemof package. It has its own package now: (demandlib)

New features

- Solph's *EnergySystem* now automatically uses solph's GROUPINGS in addition to any user supplied ones. See the API documentation for more information.
- The *groupings* introduced in version 0.0.5 now have more features, more documentation and should generally be pretty usable:

  - They moved to their own module: *oemof.groupings* and deprecated constructs ensuring compatibility with prior versions have been removed.
  - It's possible to assign a node to multiple groups from one *Grouping* by returning a list of group keys from *key*.
  - If you use a non callable object as the *key* parameter to *Groupings*, the constructor will not make an attempt to call them, but use the object directly as a key.
  - There's now a *filter* parameter, enabling a more concise way of filtering group contents than using *value*.

Documentation

- Complete revision of the documentation. We hope it is now more intuitive and easier to understand.

Testing

- Create a structure to use examples as system tests (issue #160)

Bug fixes

- Fix relative path of logger (issue #201)
- More path fixes regarding installation via pip

Other changes

- Travis CI will now check PR's automatically
- Examples executable from command-line (issue #227)

Contributors

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

v0.0.7 (May 4, 2016)

Bug fixes

- Exclude non working pyomo version

v0.0.6 (April 29, 2016)

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)

Documentation

Testing

- Fix constraint tests (Issue #137)

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.

Other changes

Contributors

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

v0.0.5 (April 1, 2016)

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)

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.

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).

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)

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.

Contributors

- Uwe Krien
- Stephan Günther
- Guido Pleßmann

v0.0.4 (March 03, 2016)

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

Documentation

- Update developer notes

Testing

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

Bug fixes

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

Other changes

- PEP8

Contributors

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

v0.0.3 (January 29, 2016)

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)

Documentation

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

Testing

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

Bug fixes

- Fix contraints in solph
- Fix pep8

Other changes

Contributors

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

v0.0.2 (December 22, 2015)

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`

Documentation

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

Testing

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

Other changes

Contributors

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

v0.0.1 (November 25, 2015)

First release by the oemof developing group.

CHAPTER 7

oemof-network

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 *node* is either a *bus* or a *component*. A bus is always connected with one or several components. Likewise components are always connected with one or several buses. Based on their characteristics 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 conversion factor. Components of type *transformer* can also be used to model transmission lines. A *sink* has only an input but no output. With *sink* consumers like households can be modeled. A *source* has exactly one output but no input. Thus for example, wind energy and photovoltaic plants can be modeled.

Components and buses can be combined to an energy system. 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.

Example

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

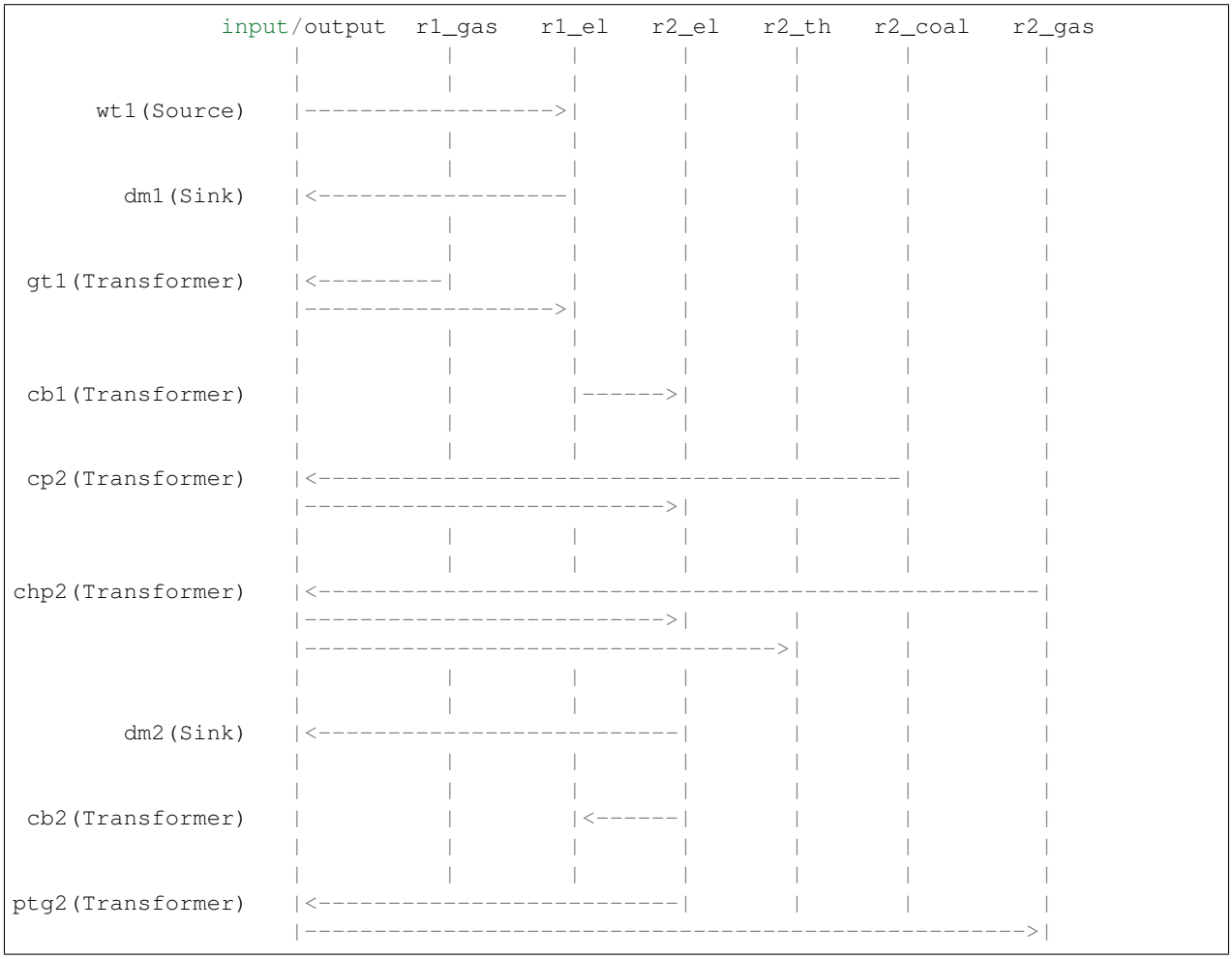
Region1:

components: wind turbine (wt1), electrical demand (dm1), gas turbine (gt1), cable to region2 (cb1) buses: 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)
buses: electrical grid (r2_el), local heat network (r2_th), coal reservoir (r2_coal), gas pipeline (r2_gas)

In oemof this would look as follows:



CHAPTER 8

oemof-solph

Solph is an oemof-package, designed to create and solve linear or mixed-integer linear optimization problems. The packages is based on pyomo. To create an energy system model the *oemof-network* ist used and extended by components such as storages. To get started with solph, checkout the solph-examples in the *Solph Examples* section.

How can I use solph?

To use solph you have to install oemof and at least one solver, which can be used together with pyomo. See pyomo installation guide. You can test it by executing one of the existing examples. Be aware that the examples require the CBC solver but you can change the solver name in the example files to your solver.

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

Set up an energy system

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

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

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

This index can be used to define the EnergySystem:

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

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

Add your components to the energy system

If you have defined an instance of the EnergySystem class all components you define will automatically be added to your EnergySystem.

Basically, there are four types of Nodes and every node has to be connected with one or more buses. The connection between a component and a bus is the flow.

- Sink (one input, no output)
- Source (one output, no input)
- LinearTransformer (one input, n outputs)
- Storage (one input, one output)

Using these types it is already possible to set up an simple energy model. But more types (e.g. flexible CHP transformer) are currently being developed. You can add your own types in your application (see below) but we would be pleased to integrate them into solph if they are of general interest.

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

Bus

All flows into and out of a bus are balanced. Therefore an instance of the Bus class represents a grid or network without losses. To define an instance of a Bus only a unique name is necessary. To make it easier to connect the bus to a component you can optionally assign a variable for later use.

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

The following code shows the difference between a bus that is assigned to a variable and one that is not.

```
print(my_energsystem.groups['natural_gas']
print(electricity_bus)
```

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

Flow

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

```
solph.Flow()
```

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

Sink

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

The example shows the electricity demand of the electricity_bus defined above. The *'my_demand_series'* should be sequence of normalised values while the *'nominal_value'* is the maximum demand the normalised sequence is multiplied with. The parameter *'fixed=True'* means that the actual_value can not be changed by the solver.

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

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

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

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

Source

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

While a wind power plant will have an hourly feed-in depending on the weather conditions the natural_gas import might be restricted by maximum value (*nominal_value*) and an annual limit (*summed_max*). As we do have to pay for imported gas we should set variable costs. Comparable to the demand series an *actual_value* in combination with *'fixed=True'* is used to define the normalised output of a wind power plan. The *nominal_value* sets the installed capacity.

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

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

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

LinearTransformer (1xM)

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

```
solph.LinearTransformer(
    label="pp_gas",
    inputs={my_energsystem.groups['natural_gas']: solph.Flow()},
    outputs={electricity_bus: solph.Flow(nominal_value=10e10)},
    conversion_factors={electricity_bus: 0.58})
```

A CHP power plant would be defined in the same manner. New buses are defined to make the code cleaner:

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

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

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

VariableFractionTransformer

The VariableFractionTransformer inherits from the *LinearTransformer (1xM)* class. An instance of this class can represent a component with one input and two output flows and a flexible ratio between these flows. By now this class is restricted to one input and two output flows. One application example would be a flexible combined heat and power (chp) plant. The class allows to define a different efficiency for every time step but this series has to be predefined a

parameter for the optimisation. In contrast to the LinearTransformer, a main flow and a tapped flow is defined. For the main flow you can define a conversion factor if the second flow is zero (conversion_factor_single_flow).

```
solph.VariableFractionTransformer(
    label='variable_chp_gas',
    inputs={bgas: solph.Flow(nominal_value=10e10)},
    outputs={bel: solph.Flow(), bth: solph.Flow()},
    conversion_factors={bel: 0.3, bth: 0.5},
    conversion_factor_single_flow={bel: 0.5}
    )
```

The key of the parameter *'conversion_factor_single_flow'* will indicate the main flow. In the example above, the flow to the Bus *'bel'* is the main flow and the flow to the Bus *'bth'* is the tapped flow. The following plot shows how the variable chp (right) shedules it's electrical and thermal power production in contrast to a fixed chp (left). The plot is the output of the *Variable chp* below.

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

LinearTransformer (Nx1)

An instance of the LinearTransformer class can represent a node with m input flows an one output flows such as a heat pump, additional heat supply or any kind of a process where two input flows are reduced to one output flow. As the name indicates the efficiency has be to constant within one time step to get a linear transformation. You can define a different efficiency for every time step (e.g. the COP of an air heat pump according to the ambient temperature) but this series has to be predefined and cannot be changed within the optimisation.

```
solph.LinearN1Transformer(
    label="pp_gas",
    inputs={my_energsystem.groups['natural_gas']: solph.Flow()},
    outputs={electricity_bus: solph.Flow(nominal_value=10e10)},
    conversion_factors={electricity_bus: 0.58})
```

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

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

cop = 3  # coefficient of performance of the heat pump

solph.LinearN1Transformer(
    label='heat_pump',
    inputs={bus_elec: Flow(), bus_low_temp_heat: Flow()},
    outputs={bus_th_high: Flow()},
    conversion_factors={bus_elec: cop,
                        b_th_low: cop/(cop-1)})
```

If the low temperature reservoir is nearly infinite (ambient air heat pump) the low temperature bus is not needed and therefore 1x1-Transformer is sufficient.

Note: See the *LinearN1Transformer* class for all parameters and the mathematical background.

Storage

In contrast to the three classes above the storage class is a pure solph class and is not inherited from the oemof-network module. The *nominal_value* of the storage signifies the nominal capacity. To limit the input and output flows you can define the ratio between these flows and the capacity using *nominal_input_capacity_ratio* and *nominal_output_capacity_ratio*. Furthermore an efficiency for loading, unloading and a capacity loss per time increment can be defined. For more information see the definition of the *Storage* class.

```
solph.Storage(
    label='storage',
    inputs={b_el: solph.Flow(variable_costs=10)},
    outputs={b_el: solph.Flow(variable_costs=10)},
    capacity_loss=0.001, nominal_value=50,
    nominal_input_capacity_ratio=1/6,
    nominal_output_capacity_ratio=1/6,
    inflow_conversion_factor=0.98, outflow_conversion_factor=0.8)
```

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

Optimise your energy system

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

Cost do have to be monitory cost but could be emissions or other variable units.

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

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

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

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

Analysing your results

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

```
my_energysystem.dump('my_path', 'my_dump.oemof')
```

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

```
import pandas as pd
import oemof.solph as solph
```

```
my_index = pd.date_range('1/1/2011', periods=8760, freq='H')
new_energysystem = solph.EnergySystem(timeindex=my_index)
new_energysystem.restore('my_path', 'my_dump.oemof')
```

If you call dump/restore with any parameters, the dump will be stored as *'es_dump.oemof'* into the *'.oemof/dumps/'* folder created in your HOME directory.

In the outputlib the results will be converted to a pandas MultiIndex DataFrame. This makes it easy to plot, save or process the results. See *oemof-outputlib* for more information.

Using the investment mode

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

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

Basically an instance of the investment class can be added to a Flow or a Storage. Adding an investment object the *nominal_value* or *nominal_capacity* should not be set. All parameters that usually refer to the *nominal_value/capacity* will now refer to the investment variable. It is also possible to set a maximum limit for the capacity that can be build.

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

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

The periodical cost are typically calculated as follows:

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

The following code shows a storage with an investment object.

```
solph.Storage(
    label='storage', capacity_loss=0.01,
    inputs={electricity: solph.Flow()}, outputs={electricity: solph.Flow()},
    nominal_input_capacity_ratio=1/6, nominal_output_capacity_ratio=1/6,
    inflow_conversion_factor=0.99, outflow_conversion_factor=0.8,
    investment=solph.Investment(ep_costs=epc))
```

Note: At the moment the investment class is not compatible with the MIP classes *BinaryFlow* and *DiscreteFlow*.

Mixed Integer (Linear) Problems

Solph also allows you to model components with respect to more technical details. For example you can model a mimimal power production (Pmin-Constraint) within oemof. Therefore, the following two classes exist in the oemof.solph.options module: *BinaryFlow* and *DiscreteFlow*. Note that the usage of these classes is not compatible with the *Investment* class at the moment.

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

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

solph.LinearTransformer(
    label='pp_chp',
    inputs={bgas: Flow(discrete=DiscreteFlow())},
    outputs={b_el: Flow(nominal_value=30, binary=BinaryFlow()),
             b_th: Flow(nominal_value=40)},
    conversion_factors={b_el: 0.3, b_th: 0.4})
```

The created LinearTransformer will now force the flow variable of its input (gas) to be of the domain discrete, i.e. {min, ... 10, 11, 12, ..., max}. The BinaryFlow() object of the 'electrical' flow will create a 'status' variable for the flow. This will be used to model for example Pmin/Pmax constraints if the attribute *min* of the flow is set. It will also be used to include start up constraints and costs if correponding attributes of the class are provided. For more information see API of BinaryFlow() class and its corresponding block class: *BinaryFlow*.

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

Adding additional constraints

You can add additional constraints to your *OperationalModel*. For now, you have to check out the examples in the *Flexible modelling* example.

The Grouping module (Sets)

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

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

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

```
def constraint_grouping(node):
    if isinstance(node, Bus) and node.balanced:
        return blocks.Bus
    if isinstance(node, LinearTransformer):
```

```
        return blocks.LinearTransformer
GROUPINGS = [constraint_grouping]
```

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

Using the CSV reader

Alternatively to a manual creation of energy system component objects as describe above, these can also be created from a pre-defined csv-structure via a csv-reader. Technically speaking, the csv-reader is a simple parser that creates oemof nodes and their respective flows by interating line by line through texts files of a specific format. The original idea behind this approach was to lower the entry barrier for new users, to have some sort of GUI in form of platform independent spreadsheet software and to make data and models exchangeable in one archive.

Both, investment and dispatch (operational) models can be modelled. Two examples and more information about the functionality can be found in the *Csv_reader* section.

Solph Examples

The following examples are available for solph. See section "*Run examples to check the installation*" to learn how to execute the examples directly. Be aware that the CBC solver has to be installed to run the examples (solver_label). If you want to use a different solver you can download the examples below and change the solver name manually.

Csv_reader

The csv-reader provides an easy to use interface to the solph library. The objects are defined using csv-files and are automatically created. There are two examples available.

- Dispatch example (`source file`, `data file 1`, `data file 2`)
- Investment example (`source file`, `data file 1`, `data file 2`).

Flexible modelling

It is also possible to pass constraints to the model that are not provided by solph but defined in your application. Inside this example two different kind of constraints are added: (1) emission constraints, (2) shared constraints between flows. To understand the example it might be useful to know a little bit about the pyomo-package and how constraints are defined, moreover you should have understood the basic underlying oemof structure. This example shows how to do it (`source file`).

Dispatch modelling

Dispatch modelling is a typical thing to do with solph. However cost does not have to be monetary but can be emissions etc. In this example a least cost dispatch of different generators that meet an inelastic demand is undertaken. Some of the generators are renewable energies with marginal costs if zero. Additionally, it shows how combined heat and power units may be easily modelled as well. (`source file`, `data file`).

Storage investment

The investment object can be used to optimise the capacity of a component. In this example all components are given but the electrical storage. The optimal size of the storage will be determined (`source file`, `data file`).

Variable chp

This example is not a real use case of an energy system but an example to show how a variable combined heat and power plant (chp) works in contrast to a fixed chp (eg. block device).

Both chp plants distribute power and heat to a separate heat and power Bus with a heat and power demand. The plot shows that the fixed chp plant produces heat and power excess and therefore needs more natural gas. (`source file`, `data file`)

CHAPTER 9

oemof-outputlib

The outputlib converts the results to a pandas MultiIndex DataFrame. In this way we make the full power of the pandas package available to process the results. See the pandas documentation to learn how to visualise, read or write or how to access parts of the DataFrame to process them.

However, oemof provides some functionality that makes your life easier, especially at the beginning.

Slicing the DataFrame

(-> *slice_by()*)

You first need to create an instance of your MultiIndex DataFrame. On this DataFrame you can apply all functions provided by pandas. The slice_by method is an oemof method to access easily a specific component or bus to process or save the results. The *type* parameter signifies the direction of the flow, into or out of a bus. The following examples writes the output of a gas power plant to a csv-file. As we do slice the whole year, the *date_from/date_to* parameter is not necessary. We can use these parameters to slice shorter periods.

```
myresults = outputlib.DataFramePlot(energy_system=my_energysystem)
pp_gas = myresults.slice_by(obj_label='pp_gas', type='to_bus',
                            date_from='2012-01-01 00:00:00',
                            date_to='2012-12-31 23:00:00')
pp_gas.to_csv('pp_gas.csv')solph.Flow()
```

You can use this approach also to plot the results, but plotting a slice of a DataFrame is not as easy as plotting a normal DataFrame so you can use the plotting class described in the next section.

Plotting parts of the DataFrame

(-> *DataFramePlot*)

This class will only add some methods to make things easier. It is still possible to access the full functionality of the matplotlib package.

Some feature provided by the outputlib:

- Configure the x-axis of you plot with date/time ticks depending on your requirements (*set_datetime_ticks*)
- Placing the legend outside the plot (*outside_legend*)
- Plotting a balance plot around a bus with all inputs/outputs (*io_plot*)
- Use one specific color for every component of your energy model (*color_from_dict*)
- Change the order of the flows in your subset for bar plots or stacked plots (*rearrange_subset*)

The following examples shows how to use the pandas plot method with the oemof's DataFramePlot class. The parameter *linewidth* and *title* belong to pandas.

```
myplot = outputlib.DataFramePlot(energy_system=energysystem)
myplot.slice_unstacked(bus_label="electricity", type="to_bus",
                       date_from="2012-01-01 00:00:00",
                       date_to="2012-01-31 00:00:00")
myplot.plot(linewidth=2, title="January 2012")
```

The following examples shows how to combine code of the matplotlib with the DataFramePlot class. Matplotlib code is used to define the figure and the font size. With the *ax* parameter this configuration is passed to the pandas plot function. Further matplolib and oemof functions are used to design the plot.

```
fig = plt.figure(figsize=(24, 14))  # matplotlib
plt.rc('legend', **{'fontsize': 19})  # matplotlib
plt.rcParams.update({'font.size': 19})  # matplotlib
plt.style.use('grayscale')  # matplotlib  # oemof
myplot.slice_unstacked(bus_label="electricity", type="from_bus")
myplot.plot(title="Year 2016", colormap='Spectral', linewidth=2, ax=fig.add_subplot(1,
↪ 1, 1))  # pandas
myplot.ax.set_ylabel('Power in MW')  # matplotlib
myplot.ax.set_xlabel('Date')  # matplotlib
myplot.ax.set_title("Electricity bus")  # matplotlib
myplot.set_datetime_ticks(number_autoticks=5, date_format='%d-%m-%Y')  # oemof
myplot.outside_legend()  # oemof
```

Creating a colour dictionary

A colour dictionary is useful to have the same colour for a specific component in every plot. Therefore you can define a colour for every component of your model using a dictionary. The key has to be the labell of the component while the value is the colour.

```
cdict = {'wind': '#5b5bae',
         'pv': 'blue',
         'storage': 'r',
         'demand': (0.34, 0.2, 0.89)}
```

As shown in the example there are different ways to defining a colour using matplolib. Have a look at the general description of matplotlib colours or use the named colours.

Creating an input/output plot for buses

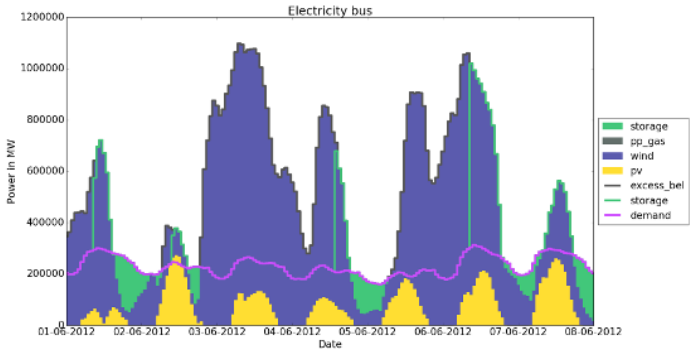
An input/output plot (i/o-plot) can be used to see the balance around a bus. All input flows are plotted as a stacked line plot, all output flows as a stacked bar plot. See *rearrange_subset* for all possible parameters. The following example shows the code of left plot below:

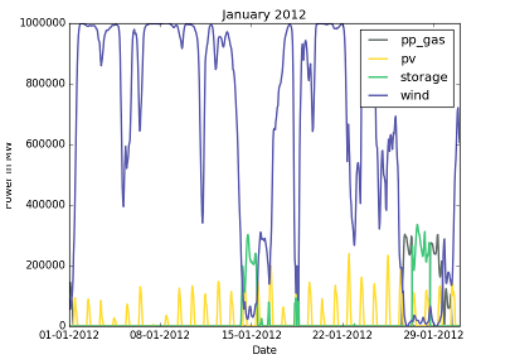
```
myplot = outputlib.DataFramePlot(energy_system=energysystem)
handles, labels = myplot.io_plot(
    bus_label='electricity', cdict=cdict,
    barorder=['pv', 'wind', 'pp_gas', 'storage'],
    lineorder=['demand', 'storage', 'excess_bel'],
    line_kwa={'linewidth': 4},
    date_from="2012-06-01 00:00:00",
    date_to="2012-06-8 00:00:00",
    )
myplot.ax.set_ylabel('Power in MW')
myplot.ax.set_xlabel('Date')
myplot.ax.set_title("Electricity bus")
myplot.set_datetime_ticks(tick_distance=24, date_format='%d-%m-%Y')
myplot.outside_legend(handles=handles, labels=labels)

plt.show()
```

Due to the rearrangement of the order with in the i/o-plot (*barorder*, *lineorder*) the handles and labels are manipulated. Therefore you have to pass them in the new order to the *outside_legend* method to get the legend in the right order. Otherwise the legend will be outside but will be in the first order.

Typical outputs of the outputlib





CHAPTER 10

oemof-tools

The oemof tools package contains little helpers to create your own application. You can use a configuration file in the ini-format to define computer specific parameters such as paths, addresses etc.. Furthermore a loggin modules helps you creating log files for your application.

List of oemof tools

Logging

No content so far, please help.

Configuration file

No content so far, please help.

CHAPTER 11

API

oemof

oemof package

Subpackages

oemof.outputlib package

Module contents

class oemof.outputlib.**DataFramePlot**(***kwargs*)

Bases: *oemof.outputlib.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 used 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_label*, *cdict*, *line_kwa=None*, *lineorder=None*, *bar_kwa=None*, *barorder=None*, ***kwargs*)

Plotting a combined bar and line plot to see the fitting of in- and out-coming flows of a bus balance.

Parameters

- **bus_label** (*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 unusual construction of the stack line plot. You can use them for further manipulations.

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).

Other Parameters

- **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 artist 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 artist 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 distance 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_label'*, ***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_label')*) – Level to unstack the subset of the DataFrame.

class oemof.outputlib.**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_labels** (*list if strings*) – List of strings with buses that should be contained in dataframe. If not set, all buses are contained.

result_object
dictionary – solph result objects

bus_labels
list if strings – List of strings with buses that should be contained in dataframe. If not set, all buses 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_balance_to_csv(*bus_labels=None*, *output_path=''*)
Method for saving bus balances of the ResultsDataFrame as single csv files. A balance around each bus with inputs, outputs and other values is saved for a passed list of bus labels. If no labels are passed, all busses are saved. Additionally, an output path for the files can be specified.

Parameters

- **bus_labels** (*list of strings*) –
- **output_path** (*string*) –

slice_bus_balance(*bus_label*)
Method for slicing the ResultsDataFrame. An balance around a bus with inputs, outputs and other values is returned.

Parameters **bus_label** (*string*) –

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

Other Parameters

- **bus_label** (*string*)
- **type** (*string (to_bus/from_bus/other)*)
- **obj_label** (*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_label'*, *formatted=False*, ***kwargs*)
Method for slicing the ResultsDataFrame. An unstacked subset is returned.

Parameters

- **unstacklevel** (*string (default: 'obj_label')*) – Level to unstack the subset of the DataFrame.
- **formatted** (*boolean*) –

oemof.solph package

Submodules

oemof.solph.blocks module

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

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

Bases: pyomo.core.base.block.SimpleBlock

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

BINARY_FLOWS A set of flows with the attribute `binary` of type `options.Binary`.

MIN_FLOWS A subset of set BINARY_FLOWS with the attribute `min` greater than zero for at least one timestep in the simulation horizon.

STARTUP_FLOWS A subset of set BINARY_FLOWS with the attribute `startup_costs` being not None.

SHUTDOWN_FLOWS A subset of set BINARY_FLOWS with the attribute `shutdown_costs` being not None.

The following variable are created:

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

Startup variable (binary) om.BinaryFlow.startup: Variable indicating startup of flow (component) indexed by STARTUP_FLOWS

Shutdown variable (binary) om.BinaryFlow.shutdown: Variable indicating shutdown of flow (component) indexed by SHUTDOWN_FLOWS

The following constraints are created:

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

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

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

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

Startup constraint om.BinaryFlow.startup_constr[i,o,t]

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

Shutdown constraint om.BinaryFlow.shutdown_constr[i,o,t]

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

The following parts of the objective function are created:

If binary.startup_costs is set by the user:

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

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

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

class `oemof.solph.blocks.`**`Bus`**(**args*, ***kwargs*)
Bases: `pyomo.core.base.block.SimpleBlock`

Block for all balanced buses.

The following constraints are build:

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

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

class `oemof.solph.blocks.`**`DiscreteFlow`**(**args*, ***kwargs*)
Bases: `pyomo.core.base.block.SimpleBlock`

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

DISCRETE_FLOWS A set of flows with the attribute `discrete` of type `options.Discrete`.

class `oemof.solph.blocks.`**`Flow`**(**args*, ***kwargs*)
Bases: `pyomo.core.base.block.SimpleBlock`

Flow block with definitions for standard flows.

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

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

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

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

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

The following constraints are build:

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

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

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

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

Negative gradient constraint `om.Flow.negative_gradient_constr[i, o]`:

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

Positive gradient constraint om.Flow.positive_gradient_constr[i, o]:

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

The following parts of the objective function are created:

If variable_costs are set by the user:

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

Additionally, if fixed_costs are set by the user:

$$\sum_{(i,o)} nominal\_value(i,o) \cdot fixed\_costs(i,o)$$

The expression can be accessed by om.Flow.fixed_costs and their value after optimization by om.Flow.fixed_costs() . This works similar for variable costs with *.variable_costs.

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

Block for all flows with investment being not None.

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

FLOWS A set of flows with the attribute invest of type *options.Investment*.

FIXED_FLOWS A set of flow with the attribute fixed set to *True*

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

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

MIN_FLOWS A subset of FLOWS with flows having set a least minimum value for at least one timestep in the optimization model.

The following variables are created:

invest om.InvestmentFlow.invest[i, o] Value of the investment variable (i.e. equivalent to the nominal value of the flows after optimization (indexed by FLOWS)

The following constraints are build:

Actual value constraint for fixed invest flows om.InvestmentFlow.fixed[i, o, t]

$$flow(i,o,t) = actual\_value(i,o,t) \cdot invest(i,o),$$
$$\forall (i,o) \in \text{FIXED\_FLOWS},$$
$$\forall t \in \text{TIMESTEPS}.$$

Lower bound (min) constraint for invest flows om.InvestmentFlow.min[i, o, t]

$$flow(i,o,t) \geq min(i,o,t) \cdot invest(i,o),$$
$$\forall (i,o) \in \text{MIN\_FLOWS},$$
$$\forall t \in \text{TIMESTEPS}.$$

Upper bound (max) constraint for invest flows `om.InvestmentFlow.max[i, o, t]`

$$flow(i, o, t) \leq max(i, o, t) \cdot invest(i, o), \quad \forall (i, o) \in \text{FLOWS}, \quad \forall t \in \text{TIMESTEPS}.$$

Flow max sum for invest flow `om.InvestmentFlow.summed_max[i, o]`

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

Flow min sum for invest flow `om.InvestmentFlow.summed_min[i, o]`

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

The following parts of the objective function are created:

Equivalent periodical costs (epc) expression `om.InvestmentFlow.investment_costs`:

$$\sum_{i,o} invest(i, o) \cdot ep\_costs(i, o)$$

Additionally, if `fixed_costs` are set by the user:

$$\sum_{i,o} invest(i, o) \cdot fixed\_costs(i, o)$$

The expression can be accessed by `om.InvestmentFlow.fixed_costs` and their value after optimization by `om.InvestmentFlow.fixed_costs()` . This works similar for variable costs with `*.variable_costs` etc.

class `oemof.solph.blocks.`**`InvestmentStorage`**(**args*, ***kwargs*)
Bases: `pyomo.core.base.block.SimpleBlock`

Storage with an *`Investment`* object.

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

INVESTSTORAGES A set with all storages containing an Investment object.

INITIAL_CAPACITY A subset of the set INVESTSTORAGES where elements of the set have an initial_capacity attribute.

MIN_INVESTSTORAGES A subset of INVESTSTORAGES where elements of the set have an capacity_min attribute greater than zero for at least one time step.

The following variables are created:

capacity `om.InvestmentStorage.capacity[n, t]` Level of the storage (indexed by STORAGES and TIMESTEPS)

invest `om.InvestmentStorage.invest[n, t]` Nominal capacity of the storage (indexed by STORAGES)

The following constraints are build:

Storage balance

$$\begin{aligned} capacity(n,t) =& capacity(n, t\_previous(t)) \cdot (1 - capacity\_loss(n)) \\ & - (flow(n, target(n), t))/(outflow\_conversion\_factor(n) \cdot \tau) \\ & + flow(source(n), n, t) \cdot inflow\_conversion\_factor(n) \cdot \tau, \\ & \forall n \in \text{INVESTSTORAGES}, \\ & \forall t \in \text{TIMESTEPS}. \end{aligned}$$

Initial capacity of *network.Storage*

$$\begin{aligned} capacity(n, t_{last}) = invest(n) \cdot initial\_capacity(n), \\ \forall n \in \text{INITIAL\_CAPACITY}, \\ \forall t \in \text{TIMESTEPS}. \end{aligned}$$

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

$$\begin{aligned} InvestmentFlow.invest(source(n), n) = invest(n) * nominal\_input\_capacity\_ratio(n) \\ \forall n \in \text{INVESTSTORAGES} \end{aligned}$$

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

$$\begin{aligned} InvestmentFlow.invest(n, target(n)) == invest(n) * nominal_output_capacity_ratio(n) \\ \forall n \in \text{INVESTSTORAGES} \end{aligned}$$

Maximal capacity `om.InvestmentStorage.max_capacity[n, t]`

$$\begin{aligned} capacity(n,t) \leq invest(n) \cdot capacity\_min(n,t), \\ \forall n \in \text{MAX\_INVESTSTORAGES}, \\ \forall t \in \text{TIMESTEPS}. \end{aligned}$$

Minimal capacity `om.InvestmentStorage.min_capacity[n, t]`

$$\begin{aligned} capacity(n,t) \geq invest(n) \cdot capacity\_min(n,t), \\ \forall n \in \text{MIN\_INVESTSTORAGES}, \\ \forall t \in \text{TIMESTEPS}. \end{aligned}$$

The following parts of the objective function are created:

Equivalent periodical costs (investment costs):

$$\sum_n invest(n) \cdot ep\_costs(n)$$

Additionally, if fixed costs are set by the user:

$$\sum_n invest(n) \cdot fixed\_costs(n)$$

The expression can be accessed by `om.InvestStorages.fixed_costs` and their value after optimization by `om.InvestStorages.fixed_costs()` . This works similar for investment costs with `*.investment_costs`.

class oemof.solph.blocks.**LinearN1Transformer**(**args*, ***kwargs*)
Bases: pyomo.core.base.block.SimpleBlock

Block for the linear relation of nodes with type class:*.LinearN1Transformer*

The following constraints are created:

Linear relation om.LinearN1Transformer.relation[i,o,t]

$$flow(i, n, t) \cdot conversion_factor(i, n, t) = flow(n, o, t),$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall n \in \text{LINEAR\_N1\_TRANSFORMERS},$$
$$\forall i \in \text{INPUTS(n)}.$$

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

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

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

LINEAR_TRANSFORMERS A set with all *LinearTransformer* objects.

The following constraints are created:

Linear relation om.LinearTransformer.relation[i,o,t]

$$flow(i, n, t) \cdot conversion\_factor(n, o, t) = flow(n, o, t),$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall n \in \text{LINEAR\_TRANSFORMERS},$$
$$\forall o \in \text{OUTPUTS(n)}.$$

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

Storages (no investment)

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

STORAGES A set with all *Storage* objects (and no attr:*investement* of type *Investment*)

The following variables are created:

capacity Capacity (level) for every storage and timestep. The value for the capacity at the beginning is set by the parameter *initial_capacity* or not set if *initial_capacity* is None. The variable of storage s and timestep t can be accessed by: *om.Storage.capacity[s, t]*

The following constraints are created:

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

$$capacity(n, t) = capacity(n, previous(t)) \cdot (1 - capacity\_loss_n(t))) - \frac{flow(n, o, t)}{\eta(n, o, t)} \cdot \tau + flow(i, n, t) \cdot \eta(i, n, t) \cdot \tau$$

The following parts of the objective function are created:

If fixed_costs is set by the user:

$$\sum_n nominal\_capacity(n, t) \cdot fixed\_costs(n)$$

The fixed costs expression can be accessed by *om.Storage.fixed_costs* and their value after optimization by: *om.Storage.fixed_costs()*.

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

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

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

VARIABLE_FRACTION_TRANSFORMERS A set with all *VariableFractionTransformer* objects.

The following constraints are created:

Variable i/o relation om.VariableFractionTransformer.relation[i,o,t]

$$flow(input, n, t) = (flow(n, main\_output, t) + flow(n, tapped\_output, t) \cdot main\_flow\_loss\_index(n, t))/ efficiency\_condensing(n, t)$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall n \in \text{VARIABLE\_FRACTION\_TRANSFORMERS}.$$

Out flow relation om.VariableFractionTransformer.relation[i,o,t]

$$flow(n, main\_output, t) = flow(n, tapped\_output, t) \cdot conversion\_factor(n, main\_output, t)/conversion\_factor(n, tapped\_output, t$$
$$\forall t \in \text{TIMESTEPS},$$
$$\forall n \in \text{VARIABLE\_FRACTION\_TRANSFORMERS}.$$

clear()

set_value(*value*)

oemof.solph.groupings module

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

TODO: Maybe move this to the module docstring? It should be somewhere prominent so solph user's immediately see that they need to use GROUPINGS when they want to create an energy system for use with solph.

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

from oemof.network import EnergySystem import solph

energy_system = EnergySystem(groupings=solph.GROUPINGS)

oemof.solph.groupings.**constraint_grouping**(*node*)
Grouping function for constraints.

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

oemof.solph.models module

class oemof.solph.models.**ExpansionModel**
Bases: pyomo.core.base.PyomoModel.ConcreteModel

An energy system model for optimized capacity expansion.

class oemof.solph.models.**OperationalModel**(*es*, ***kwargs*)

Bases: pyomo.core.base.PyomoModel.ConcreteModel

An energy system model for operational simulation with optimized dispatch.

Parameters

- **es** (*EnergySystem object*) – Object that holds the nodes of an oemof energy system graph
- **constraint_groups** (*list*) – Solph looks for these groups in the given energy system and uses them to create the constraints of the optimization problem. Defaults to OperationalModel.CONSTRAINTS
- **timeindex** (*pandas DatetimeIndex*) – The time index will be used to calculate the timesteps and the time increment for the optimization model.
- **timesteps** (sequence *(optional)*) – Timesteps used in the optimization model. If provided as list or pandas.DatetimeIndex the sequence will be used to index the time dependent variables, constraints etc. If not provided we will try to compute this sequence from attr:*timeindex*.
- **timeincrement** (*float or list of floats (optional)*) – Time increment used in constraints and objective expressions. If type is 'float', will be converted internally to solph.plumbing.Sequence() object for time dependent time increment. If a list is provided this list will be taken. Default is calculated from timeindex if provided.
- **following sets are created** (***The*) –
- **NODES** – A set with all nodes of the given energy system.
- **TIMESTEPS** – A set with all timesteps of the given time horizon.
- **FLOWS** – A 2 dimensional set with all flows. Index: *(source, target)*
- **NEGATIVE_GRADIENT_FLOWS** – A subset of set FLOWS with all flows where attribute *negative_gradient* is set.
- **POSITIVE_GRADIENT_FLOWS** – A subset of set FLOWS with all flows where attribute *positive_gradient* is set.
- **following variables are created** (***The*) –
- **flow** – Flow from source to target indexed by FLOWS, TIMESTEPS. Note: Bounds of this variable are set depending on attributes of the corresponding flow object.
- **negative_flow_gradient** – Difference of a flow in consecutive timesteps if flow is reduced indexed by NEGATIVE_GRADIENT_FLOWS, TIMESTEPS.
- **positive_flow_gradient** – Difference of a flow in consecutive timesteps if flow is increased indexed by NEGATIVE_GRADIENT_FLOWS, TIMESTEPS.

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

objective_function(*sense=1*, *update=False*)

receive_duals()

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

relax_problem()

Relaxes integer variables to reals of optimization model self

results()
Returns a nested dictionary of the results of this optimization 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]`.

Other result from the optimization model can be accessed like attributes of the flow, e.g. the invest variable for capacity of the storage 'stor' can be accessed like:

`om.results()[stor][stor].invest` attribute

For the investment flow of a 'transformer' trsf to the bus 'bel' this can be accessed with:

`om.results()[trsf][bel].invest` attribute

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

Parameters

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

Other Parameters

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

oemof.solph.network module

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

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

Notes

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

- *Bus*

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

A variant of `EnergySystem` specially tailored to solph.

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

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

class `oemof.solph.network.`**`Flow`**(***kwargs*)
Bases: `object`

Define a flow between two nodes. Note: Some attributes can only take numeric scalar as some may either take scalar or sequences (array-like). If for latter a scalar is passed, this will be internally converted to a sequence.

Parameters

- **`nominal_value`** (*`numeric`*) – The nominal value of the flow. If this value is set the corresponding optimization variable of the flow object will be bounded by this value multiplied with min(lower bound)/max(upper bound).
- **`min`** (*`numeric (sequence or scalar)`*) – Normed minimum value of the flow. The flow absolute maximum will be calculated by multiplying `nominal_value` with `min`
- **`max`** (*`numeric (sequence or scalar)`*) – Nominal maximum value of the flow. (see. `min`)
- **`actual_value`** (*`numeric (sequence or scalar)`*) – Specific value for the flow variable. Will be multiplied with the nominal_value to get the absolute value. If fixed attr is set to True the flow variable will be fixed to actual_value * `nominal_value` , I.e. this value is set exogenous.
- **`positive_gradient`** (*`numeric (sequence or scalar)`*) – The normed maximal positive difference (flow[t-1] < flow[t]) of two consecutive flow values.
- **`negative_gradient`** (*`numeric (sequence or scalar)`*) – The normed maximum negative difference (from[t-1] > flow[t]) of two consecutive timesteps.
- **`summed_max`** (*`numeric`*) – Specific maximum value summed over all timesteps. Will be multiplied with the nominal_value to get the absolute limit.
- **`summed_min`** (*`numeric`*) – see above
- **`variable_costs`** (*`numeric (sequence or scalar)`*) – The costs associated with one unit of the flow. If this is set the costs will be added to the objective expression of the optimization problem.
- **`fixed_costs`** (*`numeric`*) – The costs of the whole period associated with the absolute nominal_value of the flow.
- **`fixed`** (*`boolean`*) – Boolean value indicating if a flow is fixed during the optimization problem to its ex-ante set value. Used in combination with the `actual_value`.
- **`investment`** (*`oemof.solph.options.Investment`* object) – Object indicating if a nominal_value of the flow is determined by the optimization problem. Note: This will refer all attributes to an investment variable instead of to the nominal_value. The nominal_value should not be set (or set to None) if an investment object is used.
- **`binary`** (*`oemof.solph.options.BinaryFlow`* object) – If an binary flow object is added here, the flow constraints will be altered significantly as the mathematical model for the flow will be different, i.e. constraint etc from *`oemof.solph.blocks.BinaryFlow`* will be used instead of *`oemof.solph.blocks.Flow`*. Note: this does not work in combination with the investment attribute set at the moment.

Notes

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

- *Flow*
- *InvestmentFlow* (additionally if Investment object is present)
- ***BinaryFlow*** **(If** binary object is present, CAUTION: replaces *Flow* class)

Examples

Creating a fixed flow object:

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

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

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

class oemof.solph.network.**LinearN1Transformer**(**args*, ***kwargs*)
Bases: *oemof.network.Transformer*

A Linear N:1 Transformer object.

Parameters **conversion_factors** (*dict*) – Dictionary containing conversion factors for conversion of inflow(s) to specified outflow. Keys are output bus objects. The dictionary values can either be a scalar or a sequence with length of time horizon for simulation.

Examples

Defining an linear transformer:

```
>>> gas = Bus()
>>> biomass = Bus()
>>> trsf = LinearN1Transformer(conversion_factors={gas: 0.4,
...                                                biomass: [1, 2, 3]})
>>> trsf.conversion_factors[gas][3]
0.4
```

Notes

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

- *LinearN1Transformer*

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

A Linear Transformer object.

Parameters **conversion_factors** (*dict*) – Dictionary containing conversion factors for conversion of inflow to specified outflow. Keys are output bus objects. The dictionary values can either be a scalar or a sequence with length of time horizon for simulation.

Examples

Defining an linear transformer:

```
>>> bel = Bus()
>>> bth = Bus()
>>> bng = Bus()
>>> trsf = LinearTransformer(conversion_factors={bel: 0.4,
...                                              bth: [1, 2, 3]},
...                          inputs={bng: Flow()})
>>> trsf.conversion_factors[bel][3]
0.4
```

Notes

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

- *LinearTransformer*

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

An object with one input flow.

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

An object with one output flow.

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

Parameters

- **nominal_capacity** (*numeric*) – Absolute nominal capacity of the storage
- **nominal_input_capacity_ratio** (*numeric*) – Ratio between the nominal inflow of the storage and its capacity.
- **nominal_output_capacity_ratio** (*numeric*) – Ratio between the nominal outflow of the storage and its capacity. Note: This ratio is used to create the Flow object for the outflow and set its nominal value of the storage in the constructor.
- **nominal_input_capacity_ratio** – see: nominal_output_capacity_ratio
- **initial_capacity** (*numeric*) – The capacity of the storage in the first (and last) time step of optimization.
- **capacity_loss** (*numeric (sequence or scalar)*) – The relative loss of the storage capacity from between two consecutive timesteps.
- **inflow_conversion_factor** (*numeric (sequence or scalar)*) – The relative conversion factor, i.e. efficiency associated with the inflow of the storage.

- **outflow_conversion_factor** (*numeric (sequence or scalar)*) – see: inflow_conversion_factor
- **capacity_min** (*numeric (sequence or scalar)*) – The nominal minimum capacity of the storage as fraction of the nominal capacity (between 0 and 1, default: 0). To set different values in every time step use a sequence.
- **capacity_max** (*numeric (sequence or scalar)*) – see: capacity_min
- **investment** (*oemof.solph.options.Investment* object) – Object indicating if a nominal_value of the flow is determined by the optimization problem. Note: This will refer all attributes to an investment variable instead of to the nominal_capacity. The nominal_capacity should not be set (or set to None) if an investment object is used.

Notes

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

- *Storage* (if no Investment object present)
- *InvestmentStorage* (if Investment object present)

class oemof.solph.network.**VariableFractionTransformer**(*conversion_factor_single_flow*, **args*, ***kwargs*)

Bases: *oemof.solph.network.LinearTransformer*

A linear transformer with more than one output, where the fraction of the output flows is variable. By now it is restricted to two output flows.

One main output flow has to be defined and is tapped by the remaining flow. Thus, the main output will be reduced if the tapped output increases. Therefore a loss index has to be defined. Furthermore a maximum efficiency has to be specified if the whole flow is led to the main output (tapped_output = 0). The state with the maximum tapped_output is described through conversion factors equivalent to the LinearTransformer.

Parameters

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

Examples

```
>>> bel = Bus(label='electricityBus')
>>> bth = Bus(label='heatBus')
>>> bgas = Bus(label='commodityBus')
>>> vft = VariableFractionTransformer(
...     label='variable_chp_gas',
...     inputs={bgas: Flow(nominal_value=10e10)},
...     outputs={bel: Flow(), bth: Flow()},
...     conversion_factors={bel: 0.3, bth: 0.5},
...     conversion_factor_single_flow={bel: 0.5})
```

Notes

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

- *VariableFractionTransformer*

oemof.solph.network.**storage_nominal_value_warning**(*flow*)

oemof.solph.options module

Optional classes to be added to a network class.

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

Parameters

- **startup_costs** (*numeric*) – Costs associated with a start of the flow (representing a unit).
- **shutdown_costs** (*numeric*) – Costs associated with the shutdown of the flow (representing a until).
- **minimum_uptime** (*numeric*) – Minimum time that a flow must be greater then its minimum flow after startup.
- **minimum_downtime** (*numeric*) – Minimum time a flow is forced to zero after shutting down.
- **initial_status** (*numeric (0 or 1)*) – Integer value indicating the status of the flow in the first time step (0 = off, 1 = on).

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

Parameters **integers** (*boolean*) – Specify domain of flow variable: If True, flow is forced to integer values.

class oemof.solph.options.**Investment**(*maximum=inf*, *minimum=0*, *ep_costs=0*)
Bases: object

Parameters

- **maximum** (*float*) – Maximum of the additional invested capacity
- **minimum** (*float*) – Minimum of the addtional invested capacity
- **ep_costs** (*float*) – Equivalent periodical costs for the investment, if period is one year these costs are equal to the equivalent annual costs.

oemof.solph.plumbing module

oemof.solph.plumbing.**sequence**(*sequence_or_scalar*)
Tests if an object is sequence (except string) or scalar and returns a the original sequence if object is a sequence and a 'emulated' sequence object of class _Sequence if object is a scalar or string.

Parameters **sequence_or_scalar** (*array-like, None, int, float*) –

Examples

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

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

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

Module contents

oemof.tools package

Submodules

oemof.tools.config module

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*)

Parameters

- **section** (*str*) – Section of the config file
- **key** (*str*) – Key of the config file

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

Return type float, int, str, boolean

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

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

oemof.tools.config.**set**(*section*, *key*, *value*)

Parameters

- **section** (*str*) – Section of the config file
- **key** (*str*) – Key of the config file
- **value** (*float, int, str, boolean*) – Value for the given key.

oemof.tools.helpers module

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

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

oemof.tools.helpers.**get_basic_path**()
Returns the basic oemof path and creates it if necessary.

oemof.tools.helpers.**get_fullpath**(*path*, *filename*)
Combines path and filename to a full path.

oemof.tools.logger module

oemof.tools.logger.**check_git_branch**()
Passes the used branch 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'*, *log_version=True*)
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 copied from 'default_files' and used.

- **basicpath** (*string, optional (default: '.oemof' in HOME)*) – The basicpath for different oemof related information. 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.

- **log_version** (*boolean*) – If True the actual version or commit is logged while initialising the logger.

Returns str

Return type Place where the log file is 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 logger 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")
```

oemof.tools.logger.**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.

Parameters

- **start** (*float*) – start time
- **text** (*string*) – text to describe the log
- **logging_level** (*string*) – logging_level [default='debug']

Module contents

Submodules

oemof.energy_system module

Created on Mon Jul 20 15:53:14 2015

@author: uwe

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

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

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

Parameters

- **entities** (list of Entity, optional) – A list containing the already existing Entities that should be part of the energy system. Stored in the *entities* attribute. Defaults to *[]* if not supplied.
- **timeindex** (*pandas.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

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

timeindex
pandas.index, optional – Define the time range and increment for the energy system. This is an optional 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.network import Entity
>>> from oemof.network import Bus, Sink
>>> es = EnergySystem()
>>> bus = Bus(label='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 sets of entities stored under the returned keys, like in this example, where entities are grouped by their *type*:

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

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

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

flows()

groups

nodes

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

oemof.groupings module

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

oemof.groupings.**DEFAULT = <oemof.groupings.Grouping object>**
The default *Grouping*.

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

class oemof.groupings.**Flows**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)
Bases: *oemof.groupings.Nodes*

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

value(*flows*)
Returns a set containing only flows, so groups are sets of flows.

class oemof.groupings.**FlowsWithNodes**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)
Bases: *oemof.groupings.Nodes*

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

value(*tuples*)
Returns a set containing only tuples, so groups are sets of tuples.

class oemof.groupings.**Grouping**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)
Bases: object

Used to aggregate entities in an energy system into groups.

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

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

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

Notes

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

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

Parameters

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

filter(*group*)

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

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

key(*e*)

Obtain a key under which to store the group.

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

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

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

merge(*new*, *old*)

Merge a known old group with a new one.

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

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

value(*e*)
Generate the group obtained from `e`.

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

The default returns the `entity` itself.

class oemof.groupings.**Nodes**(*key=None*, *constant_key=None*, *filter=None*, ***kwargs*)
Bases: *oemof.groupings.Grouping*

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

merge(*new*, *old*)
`Updates old` to be the union of `old` and `new`.

value(*e*)
Returns a `set` containing only `e`, so groups are `sets` of `entities`.

oemof.network module

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

class oemof.network.**Component**(**args*, ***kwargs*)
Bases: *oemof.network.Node*

class oemof.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 Entity.
- **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

class oemof.network.**Node**(**args*, ***kwargs*)
Bases: object

Represents a Node in an energy system graph.

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

Parameters

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

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

inputs
dict – Dictionary mapping input Node`s `n to flows from *n* into *self*.

outputs
dict – Dictionary mapping output Node`s `n to flows from *self* into *n*.

inputs

label

outputs

registry = None

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

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

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

Module contents

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

K

L

M

N

O

P

R

S

T

V