oemof.tabular

Release 0.0.1dev

Jun 20, 2019

Contents

CHAPTER 1

Overview

Load oemof energy systems from tabular data sources.

- Free software: BSD 3-Clause License

1.1 Installation

We are currently using features which haven't made it to a proper *oemof* release yet. This means that you have to install *oemof.tabular* from source, since *pip* doesn't allow packages on *PyPI* to have dependencies which are not hosted on *PyPI*:

```
pip install 'git+https://git@github.com/oemof/oemof-tabular.git'
```

You also need at least *pip* version *18.1* for this to work.

1.2 Documentation

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

1.3 Development

To run the all tests run:

```
tox
```

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

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

CHAPTER 2

Installation

At the command line:

```
pip install oemof.tabular
```

CHAPTER 3

Usage

To use oemof.tabular in a project:

```
import oemof.tabular
```

3.1 Background

Energy systems modelling requires versatile tools to model systems with different levels of accuracy and detail. For example, for some countries the lack of data may force energy system analysts to apply simplistic approaches. In other cases comprehensive interlinked models may be applied to analyse energy systems and future pathways.

A major part of energy system modelling is the data handling including input collection, processing and result analysis. There is yet no standardized or custom broadly used model-agnostic data container in the scientific field of energy system modelling to hold energy system related data. To enable transparency and reproducibility as well as reusability of existing data the following data model description has been developed to store energy system related data in the datapackage format. The underlying concept is closely linked to the graph based description of energy as developed for the Open Energy Modelling Framework.

The Open Energy Modelling Framework is based on a graph structure at its core. In addition it provides an optimization model generator to construct individual and suitable models. The internal logic, used terminology and software architecture is abstract and rather designed for model developers and experienced modellers.

Oemof users / developers can model energy systems with different degrees of freedom:

1. Modelling based using existing classes
2. Add own classes
3. Add own constraints based on the underlying algebraic modelling library

However, in some cases complexity of this internal logic and full functionality is neither necessary nor suitable for model users. Therefore we provide so called facade classes that provide an energy specific and reduced access to the underlying oemof functionality.

Currently we provide the following facades:

- Dispatchable
- Volatile
- Link
- Conversion
- Storage
- Reservoir
- Extraction
- Backpressure
- Reservoir
- Load

Modelling energy systems based on these classes is straightforward. Parametrization of an energy system can either be done via python scripting or by using the datapackage structure described below.

3.2 Datapackage

To construct a model based on the datapackage the following 2 steps are required:

1. Add the topology of the energy system based on the components and their **exogenous model variables**.
2. Create a python script to construct the energy system and the model from that data.

We recommend a specific workflow to allow to publish your scenario (input data, assumptions, model and results) altogether in one consistent block based on the datapackage standard (see: Reproducible Workflows).

3.2.1 How to create a Datapackage

We adhere to the frictionless (tabular) datapackage standard. On top of that structure we add our own logic. We require at least two things:

1. A directory named *data* containing at least one sub-folder called *elements* (optionally it may contain a directory *sequences* and *geometries*. Of course you may add any other directory, data or other information.)
2. A valid meta-data *.json* file for the datapackage

NOTE: You **MUST** provide one file with the buses called *bus.csv*!

The resulting tree of the datapackage could for example look like this:

```
|-- datapackage
    |-- data
        |-- elements
            |-- demand.csv
            |-- generator.csv
            |-- storage.csv
            |-- bus.csv
        |-- sequences
    |-- scripts
    |-- datapackage.json
```

Inside the datapackage, data is stored in so called resources. For a tabular-datapackage, these resources are CSV files. Columns of such resources are referred to as *fields*. In this sense field names of the resources are equivalent to parameters of the energy system elements and sequences.

To distinguish elements and sequences these two are stored in sub-directories of the data directory. In addition geometrical information can be stored under *data/geometries* in a *.geojson* format. To simplifiy the process of creating and processing a datapackage you may also use the funtionalities of the *oemof.tabular.datapackage*

You can use functions to read and write resources (pandas.DataFrames in python). This can also be done for sequences and geometries.

```
from oemof.tabular.datapackage import building
...

building.read_elements('volatile.csv')

# manipulate data ...

building.write_elements('volatile.csv')
```

To create meta-data *json* file you can use the following code:

```
from datapackage_utilities import building

building.infer_metadata(
                        package_name="my-datapackage",
                        foreign_keys={
                                      "bus": [
                                                  "volatile",
                                                  "dispatchable",
                                                  "storage",
                                                  "heat_storage",
                                                  "load",
                                                  "ror",
                                                  "reservoir",
                                                  "phs",
                                                  "excess",
                                                  "boiler",
                                                  "commodity",
                                      ],
                                      "profile": ["load", "volatile", "heat_load",
↪"ror", "reservoir"],
                                      "from_to_bus": ["link", "conversion", "line"],
                                      "chp": ["backpressure", "extraction"],
                        },
                        path="/home/user/datpackages/my-datapackage"
        )
```

3.2.2 Elements

We recommend using one tabular data resource (i.e. one csv-file) for each type you want to model. The fields (i.e. column names) match the attribute names specified in the description of the facade classes.

Example for **Load**:

```
| name      | type    | tech  |amount | profile          | bus              |
|-----------|---------| ------|-------|------------------|------------------|
```

(continues on next page)

(continued from previous page)

```
| el-demand | load    | load  | 2000  | demand-profile1 | electricity-bus |
| ...       |  ...    | .... | ...   |     ...         |     ...         |
```

The corresponding meta data *schema* of the resource would look as follows:

```
"schema": {
    "fields": [
        {
            "name": "name",
            "type": "string",
        },
        {
            "name": "type",
            "type": "string",
        },
        {
            "name": "tech",
            "type": "string",
        },
        {
            "name": "amount",
            "type": "number",
        },
        {
            "name": "profile",
            "type": "string",
        },
        {
            "name": "bus",
            "type": "string",
        }
    ],
    "foreignKeys": [
          {
              "fields": "bus",
              "reference": {
                  "fields": "name",
                  "resource": "bus"
              }
          },
          {
              "fields": "profile",
              "reference": {
                  "resource": "load_profile"
              }
          }
    ],
}
```

Example for **Dispatchable**:

```
| name  | type         | capacity | capacity_cost   | bus             | marginal_cost␣
↪|
|-------|--------------|----------|-----------------|-----------------|--------------
↪|
| gen   | dispatchable | null     | 800             | electricity-bus | 75           ␣
↪|
```

(continues on next page)

(continued from previous page)

```
| ...   |     ...       |    ...   |     ...          |     ...          | ...          ␣
↪|
```

3.2.3 Sequences

A resource stored under */sequences* should at leat contain the field *timeindex* with the following standard format ISO 8601, i.e. *YYYY-MM-DDTHH:MM:SS*.

Example:

```
| timeindex         |  load-profile1    |  load-profile2    |
|-------------------|-------------------|-------------------|
| 2016-01-01T00:00  |      0.1          |       0.05        |
| 2016-01-01T01:00  |      0.2          |       0.1         |
```

The schema for resource *load_profile* stored under *sequences/load_profile.csv* would be described as follows:

```
"schema": {
    "fields": [
        {
            "name": "timeindex",
            "type": "datetime",
        },
        {
            "name": "load-profile1",
            "type": "number",
        },
        {
            "name": "load-profile2",
            "type": "number",
        }
    ]
}
```

3.3 Foreign Keys

Parameter types are specified in the (json) meta-data file corresponding to the data. In addition foreign keys can be specified to link elements entries to elements stored in other resources (for example buses or sequences).

To reference the *name* field of a resource with the bus elements (bus.csv, resource name: bus) the following FK should be set in the element resource:

```
"foreignKeys": [
  {
    "fields": "bus",
    "reference": {
      "fields": "name",
      "resource": "bus"
    }
  }
]
```

This structure can also be used to reference sequences, i.e. for the field *profile* of a resource, the reference can be set like this:

```
"foreignKeys": [
  {
    "fields": "profile",
    "reference": {
      "resource": "generator_profile"
    }
  }
]
```

In contrast to the above example, where the foreign keys points to a special field, in this case references are resolved by looking at the field names in the generators-profile resource.

NOTE: This usage breaks with the datapackage standard and creates non-valid resources.

3.4 Scripting

Currently the only way to construct a model and compute it is by using the *oemof.solph* library. As described above, you can simply use the command line tool on your created datapackage. However, you may also use the *facades.py* module and write your on application.

Just read the *.json* file to create an *solph.EnergySystem* object from the datapackage. Based on this you can create the model, compute it and process the results.

```
from oemof.solph import EnergySystem, Model
from renpass.facades import Load, Dispatchable, Bus

es = EnergySystem.from_datapackage(
    'datapackage.json',
    attributemap={
        Demand: {"demand-profiles": "profile"}},
    typemap={
        'load': Load,
        'dispatchable': Dispatchable,
        'bus': Bus})

m = Model(es)
m.solve()
```

Note: You may use the *attributemap* to map your your field names to facade class attributes. In addition you may also use different names for types in your datapackage and map those to the facade classes (use *typemap* attribute for this)

3.4.1 Write results

For writing results you either use the *oemof.outputlib* functionalities or / and the oemof tabular specific postprocessing functionalities of this package.

3.5 Reproducible Workflows

To proudce reproducible results we recommend setting up a folder strucutre as follows:

```
|-- model
        |-- environment
                |--requirements.txt
        |-- raw-data
        |-- scenarios
                |--scenario1.toml
                |--scenatio2.toml
                |-- ...
        |-- scripts
                |--create_input_data.py
                |--compute.py
                |-- ...
        |-- results
                |--scenario1
                        |--input
                        |--output
                 |-- scenario2
                        |--input
                        |--ouput
```

The *raw-data* directory contains all input data files required to build the input datapckages for your modelling. The *scenatios* directory allows you to specify different scenarios and describe them in a basic way. The scripts inside the *scripts* directory will build input data for your scenarios from the *.toml* files and the raw-data. In addition the script to compute the models can be stored there.

Of course the structure may be adapted to your needs. However you should provide all this data when publishing results.

3.6 Debugging

Debugging can sometimes be tricky, here are some things you might want to consider:

3.6.1 Components do not end up in the model

- Does the data resource (i.e. csv-file) for your components exist in the *datapackage.json* file
- Did you set the *attributemap* and *typemap* arguments of the *EnergySystem.from_datapackge()* method correctly? Make sure all classes with their types are present.

3.6.2 Cast errors when reading a datapackage

- Does the column order match the order of fields in the (tabular) data resource?
- Does the type match the types in of the columns (i.e. for integer, obviously only integer values should be in the respective column)

3.6.3 OEMOF related errors

If you encounter errors from oemof, the objects are not instantiated correctly which may happen if something of the following is wrong in your metadata file.

- Errors regarding the non-int type like this one:

```
...
self.flows[o, i].nominal_value)
TypeError: can't multiply sequence by non-int of type 'float'
```

Check your type(s) in the *datapackage.json* file. If meta-data are inferred types might be string instead of number or integer which most likely causes such an error.

- Profiles for volatile and load components

```
...
ValueError: Cannot fix flow value to None.
Please set the actual_value attribute of the flow
```

This error is likely to occur if your foreign keys are set correctly but the name in the field *profile* of your *volatilel.csv* resource does not match any name inside the *volatile_profile.csv* file, i.e. the profile is not found where it is looked for.

Another possible source of error might be the missing values in your sequences files. Check these files for NaNs.

3.6.4 Pyomo related errors

If you encounter an error for writing a lp-file, you might want to check if your foreign-keys are set correctly. In particular for resources with fk's for sequences. If this is missing, you will get unsupported operation string and numeric. This will unfortunately only happen on the pyomo level currently.

CHAPTER 4

Reference

4.1 oemof.tabular package

4.1.1 Subpackages

oemof.tabular.datapackage package

Submodules

oemof.tabular.datapackage.aggregation module

Module used for aggregation sequences and elements.

oemof.tabular.datapackage.aggregation.**loop_temporal**(*f*, *d*, *narray*, **args*)

oemof.tabular.datapackage.aggregation.**temporal_clustering**(*datapackage*, *n*, *path='/tmp'*, *how='daily'*)

oemof.tabular.datapackage.aggregation.**temporal_skip**(*datapackage*, *n*, *path='/tmp'*, *name=None*, **args*)

Creates a new datapackage by aggregating sequences inside the *sequence* folder of the specified datapackage by skipping *n* timesteps

Parameters

- **datapackage** (*string*) – String of meta data file datapackage.json
- **n** (*integer*) – Number of timesteps to skip
- **path** (*string*) – Path to directory where the aggregated datapackage is stored
- **name** (*string*) – Name of the new, aggregated datapackage. If not specified a name will be given

oemof.tabular.datapackage.building module

oemof.tabular.datapackage.building.**download_data**(*url*, *directory='cache'*, *unzip_file=None*, ***kwargs*)

Downloads data and stores it in specified directory

Parameters

- **url** (*str*) – Url of file to be downloaded.
- **directory** (*str*) – Name of directory where to store the downloaded data. Default is 'cache'-
- **unzip_file** (*str*) – Regular or directory file name to be extracted from zip source.
- **kwargs** – Additional keyword arguments.

oemof.tabular.datapackage.building.**get_config**(*file='config.json'*)

Read config json file

Parameters **file** (*string*) – String with name of config file

oemof.tabular.datapackage.building.**infer_metadata**(*package_name='default-name'*, *keep_resources=False*, *foreign_keys={'bus': ['volatile', 'dispatchable', 'storage', 'load', 'reservoir', 'shortage', 'excess'], 'chp': ['backpressure', 'extraction', 'chp'], 'from_to_bus': ['connection', 'line', 'conversion'], 'profile': ['load', 'volatile', 'ror']}*, *path=None*)

Add basic meta data for a datapackage

Parameters

- **package_name** (*string*) – Name of the data package
- **keep_resource** (*boolean*) – Flag indicating of the resources meta data json-files should be kept after main datapackage.json is created. The reource meta data will be stored in the *resources* directory.
- **foreign_keys** (*dict*) – Dictionary with foreign key specification. Keys for dictionary are: 'bus', 'profile', 'from_to_bus'. Values are list with strings with the name of the resources
- **path** (*string*) – Absoltue path to root-folder of the datapackage

oemof.tabular.datapackage.building.**infer_resources**(*directory='data/elements'*)

Method looks at all files in *directory* and creates datapackage.Resource object that will be stored

Parameters **directory** (*string*) – Path to directory from where resources are inferred

oemof.tabular.datapackage.building.**initialize_datapackage**(*config*)

Initialize datapackage by reading config file and creating required directories (data/elements, data/sequences etc.) if directories are not specified in the config file, the default directory setup up will be used.

oemof.tabular.datapackage.building.**input_filepath**(*file*, *directory='archive/'*)

oemof.tabular.datapackage.building.**package_from_resources**(*resource_path*, *output_path*, *clean=True*)

Collects resource descriptors and merges them in a datapackage.json

Parameters

- **resource_path** (*string*) – Path to directory with resources (in .json format)

- **output_path** (*string*) – Root path of datapackage where the newly created datapckage.json is stored
- **clean** (*boolean*) – If true, resources will be deleted

oemof.tabular.datapackage.building.**read_elements**(*filename*, *directory='data/elements'*)

Reads element resources from the datapackage

Parameters

- **filename** (*string*) – Name of the elements to be read, for example *load.csv*
- **directory** (*string*) – Directory where the file is located. Default: *data/elements*

Returns *pd.DataFrame*

oemof.tabular.datapackage.building.**read_geometries**(*filename*, *directory='data/geometries'*)

Reads geometry resources from the datapackage. Data may either be stored in geojson format or as WKT representation in CSV-files.

Parameters

- **filename** (*string*) – Name of the elements to be read, for example *buses.geojson*
- **directory** (*string*) – Directory where the file is located. Default: *data/geometries*

Returns *pd.Series*

oemof.tabular.datapackage.building.**read_sequences**(*filename*, *directory='data/sequences'*)

Reads sequence resources from the datapackage

Parameters

- **filename** (*string*) – Name of the sequences to be read, for example *load_profile.csv*
- **directory** (*string*) – Directory from where the file should be read. Default: *data/sequences*

oemof.tabular.datapackage.building.**timeindex**(*year=None*, *periods=None*, *freq=None*)

Create pandas datetimeindexself. If no parameters are not specified, config file will be used to set the values.

Parameters

- **year** (*string*) – Year of the index
- **periods** (*string*) – Number of periods
- **freq** (*string*) – Freq of the datetimeindex

oemof.tabular.datapackage.building.**update_package_descriptor**()

oemof.tabular.datapackage.building.**write_elements**(*filename*, *elements*, *directory='data/elements'*, *replace=False*)

Writes elements to filesystem.

Parameters

- **filename** (*string*) – Name of the elements to be read, for example *reservoir.csv*
- **elements** (*pd.DataFrame*) – Elements to be stored in data frame. Index: *name*
- **directory** (*string*) – Directory where the file is stored. Default: *data/elements*
- **replace** (*boolean*) – If set, existing data will be overwritten. Otherwise integrity of data (unique indices) will be checked

Returns **path** (*string*) – Returns the path where the file has been stored.

oemof.tabular.datapackage.building.**write_geometries**(*filename*, *geometries*, *directory='data/geometries'*)

Writes geometries to filesystem.

Parameters

- **filename** (*string*) – Name of the geometries stored, for example *buses.geojson*
- **geometries** (*pd.Series*) – Index entries become name fields in GeoJSON properties.
- **directory** (*string*) – Directory where the file is stored. Default: *data/geometries*

Returns **path** (*string*) – Returns the path where the file has been stored.

oemof.tabular.datapackage.building.**write_sequences**(*filename*, *sequences*, *directory='data/sequences'*, *replace=False*)

Writes sequences to filesystem.

Parameters

- **filename** (*string*) – Name of the sequences to be read, for example *load_profile.csv*
- **sequences** (*pd.DataFrame*) – Sequences to be stored in data frame. Index: *datetimeindex* with format %Y-%m-%dT%H:%M:%SZ
- **directory** (*string*) – Directory where the file is stored. Default: *data/elements*
- **replace** (*boolean*) – If set, existing data will be overwritten. Otherwise integrity of data (unique indices) will be checked

Returns **path** (*string*) – Returns the path where the file has been stored.

oemof.tabular.datapackage.processing module

oemof.tabular.datapackage.processing.**clean_datapackage**(*path=None, directories=['data', 'cache', 'resources']*)

Parameters

- **path** (*str*) – Path to root directory of the datapackage, if no path is passed the current directory is to be assumed the root.
- **directories** (*list (optional)*) – List of directory names inside the root directory to clean (remove).

oemof.tabular.datapackage.processing.**copy_datapackage**(*source*, *destination*, *subset=None*)

Parameters

- **source** (*str*) – datapackage.json
- **destination** (*str*) – Destination of copied datapackage
- **only_data** (*str*) – Name of directory to only copy subset of datapackage (for example only the 'data' directory)

oemof.tabular.datapackage.processing.**merge_packages**(*p1, p2, destpath, target_bus=None, inserted_bus=None, name=None, how=None, refm={'demand': 'amount', 'dispatchable-generator': 'capacity', 'pumped-storage': ['capacity', 'power'], 'transshipment': None, 'volatile-generator': 'capacity'}*)

Parameters

- **p1** (*string*) – Path to datapackage.json of package to insert in *p2*
- **p2** (*string*) – Path to datapackage.json of package where *p1* data should be inserted
- **destpath** (*string*) – Path to destination directory.
- **target_bus** (*string*) – The name of the bus where the *insereted_bus* should be connceted to.
- **inserted_bus** (*string*) – The name of the bus that should be inserted / connected to target bus.
- **name** (*string*) – Name of new (merged) datapackage
- **how** (*string*) – How the merge should take place. Default is None which will simply merge the resources. Options are *techwise* and *elementwise* (1:1)
- **refm** (*dict*) – Resource-element-field mapper. Used for substraction etc. The dict specifies which field should be merged for a specific resource.
- **Examples**
- **———**
- **merge_packages(p1='/home/user/Bordelum/datapackage.json',** – p2='/home/user/e-highway/datapackage.json', target_bus='DE-electricity', inserted_bus='Bordelum-electricity', name='merged-package', how='techwise')

oemof.tabular.datapackage.processing.**to_dict**(*value*)

Convert value from e.g. csv-reader to valid json / dict

oemof.tabular.datapackage.reading module

Tools to deserialize energy systems from datapackages.

WARNING

This is work in progress and still pretty volatile, so use it at your own risk. The datapackage format and conventions we use are still a bit in flux. This is also why we don't have documentation or tests yet. Once things are stabilized a bit more, the way in which we extend the datapackage spec will be documented along with how to use the functions in this module.

oemof.tabular.datapackage.reading.**deserialize_energy_system**(*cls, path, typemap={}, attributemap={}*)

oemof.tabular.datapackage.reading.**read_facade**(*facade, facades, create, typemap, data, objects, sequence_names, fks, resources*)

Parse the resource *r* as a facade.

oemof.tabular.datapackage.reading.**sequences**(*r*, *timeindices=None*)
Parses the resource *r* as a sequence.

oemof.tabular.tools package

oemof.tabular's kitchen sink module.

Contains all the general tools needed by other tools dealing with specific tabular data sources.

class oemof.tabular.tools.**HSN**
Bases: types.SimpleNamespace

A hashable variant of *types.Simplenamespace*.

By making it hashable, we can use the instances as dictionary keys, which is necessary, as this is the default type for flows.

oemof.tabular.tools.**raisestatement**(*exception*, *message=''*)
A version of *raise* that can be used as a statement.

oemof.tabular.tools.**remap**(*mapping*, *renamings*, *selection*)
Change *mapping*'s keys according to the *selection* in *renamings*.

The *renaming* found under *selection* in *renamings* is used to rename the keys found in *mapping*. I.e., return a copy of *mapping* with every *key* of *mapping* that is also found in *renaming* replaced with *renaming[key]*.

If key doesn't have a renaming, it's returned as is. If *selection* doesn't appear as a key in *renamings*, *mapping* is returned unchanged.

Example

```
>>> renamings = {'R1': {'zero': 'nada'}, 'R2': {'foo': 'bar'}}
>>> mapping = {'zero': 0, 'foo': 'foobar'}
>>> remap(mapping, renamings, 'R1') == {'nada': 0, 'foo': 'foobar'}
True
>>> remap(mapping, renamings, 'R2') == {'zero': 0, 'bar': 'foobar'}
True
```

As a special case, if *selection* is a *class*, not only *selection* is considered to select a renaming, but the classes in *selection*'s *mro* are considered too. The first class in *selection*'s *mro* which is also found to be a key in *renamings* is used to determine which renaming to use. The search starts at *selection*.

Parameters

- **mapping** (*Mapping*) – The *Mapping* whose keys should be renamed.
- **renamings** (*Mapping* of *Mappings* <*collections.abc.Mapping*>)
- **selection** (*Hashable*) – Key specifying which entry in *renamings* is used to determine the new keys in the copy of *mapping*. If *selection* is a *class*, the first entry of *selection*'s *mro* which is found in *renamings* is used to determine the new keys.

Submodules

oemof.tabular.tools.geometry module

4.1.2 Submodules

4.1.3 oemof.tabular.facades module

'Facade's are classes providing a simplified view on more complex classes.

More specifically, the *Facade's in this module act as simplified, energy specific wrappers around 'oemof*'s and *oemof.solph*'s more abstract and complex classes. The idea is to be able to instantiate a *Facade* using keyword arguments, whose value are derived from simple, tabular data sources. Under the hood the *Facade* then uses these arguments to construct an *oemof* or *oemof.solph* component and sets it up to be easily used in an *EnergySystem*.

SPDX-License-Identifier: BSD-3-Clause

class oemof.tabular.facades.**BackpressureTurbine**(**args*, ***kwargs*)

Bases: oemof.solph.network.Transformer, *oemof.tabular.facades.Facade*

Combined Heat and Power (backpressure) unit with one input and two outputs.

Parameters

- **electricity_bus** (*oemof.solph.Bus*) – An oemof bus instance where the chp unit is connected to with its electrical output
- **heat_bus** (*oemof.solph.Bus*) – An oemof bus instance where the chp unit is connected to with its thermal output
- **fuel_bus** (*oemof.solph.Bus*) – An oemof bus instance where the chp unit is connected to with its input
- **carrier_cost** (*numeric*) – Input carrier cost of the backpressure unit, Default: 0
- **capacity** (*numeric*) – The electrical capacity of the chp unit (e.g. in MW).
- **electric_efficiency** – Electrical efficiency of the chp unit
- **thermal_efficiency** – Thermal efficiency of the chp unit
- **marginal_cost** (*numeric*) – Marginal cost for one unit of produced electrical output E.g. for a powerplant: marginal cost =fuel cost + operational cost + co2 cost (in Euro / MWh) if timestep length is one hour. Default: 0
- **capacity_cost** (*numeric*) – Investment costs per unit of electrical capacity (e.g. Euro / MW) . If capacity is not set, this value will be used for optimizing the chp capacity.

build_solph_components()

class oemof.tabular.facades.**Conversion**(**args*, ***kwargs*)

Bases: oemof.solph.network.Transformer, *oemof.tabular.facades.Facade*

Conversion unit with one input and one output.

Parameters

- **from_bus** (*oemof.solph.Bus*) – An oemof bus instance where the conversion unit is connected to with its input.
- **to_bus** (*oemof.solph.Bus*) – An oemof bus instance where the conversion unit is connected to with its output.
- **capacity** (*numeric*) – The conversion capacity (output side) of the unit.
- **efficiency** (*numeric*) – Efficiency of the conversion unit (0 <= efficiency <= 1). Default: 1

- **marginal_cost** (*numeric*) – Marginal cost for one unit of produced output. Default: 0
- **capacity_cost** (*numeric*) – Investment costs per unit of output capacity. If capacity is not set, this value will be used for optimizing the conversion output capacity.
- **input_parameters** (*dict (optional)*) – Set parameters on the input edge of the storage (see oemof.solph for more information on possible parameters)
- **ouput_parameters** (*dict (optional)*) – Set parameters on the output edge of the storage (see oemof.solph for more information on possible parameters)

build_solph_components()

class oemof.tabular.facades.**Dispatchable**(**args*, ***kwargs*)
Bases: oemof.solph.network.Source, *oemof.tabular.facades.Facade*

Dispatchable element with one output for example a gas-turbine

Parameters

- **bus** (*oemof.solph.Bus*) – An oemof bus instance where the unit is connected to with its output
- **capacity** (*numeric*) – The installed power of the generator (e.g. in MW). If not set the capacity will be optimized (s. also *capacity_cost* argument)
- **profile** (*array-like (optional)*) – Profile of the output such that profile[t] * installed yields output for timestep t
- **marginal_cost** (*numeric*) – Marginal cost for one unit of produced output, i.e. for a powerplant: mc = fuel_cost + co2_cost + ... (in Euro / MWh) if timestep length is one hour. Default: 0
- **capacity_cost** (*numeric (optional)*) – Investment costs per unit of capacity (e.g. Euro / MW) . If capacity is not set, this value will be used for optimizing the generators capacity.
- **output_paramerters** (*dict (optional)*) – Parameters to set on the output edge of the component (see. oemof.solph Edge/Flow class for possible arguments)
- **capacity_potential** (*numeric*) – Max install capacity if capacity is to be optimized

build_solph_components()

class oemof.tabular.facades.**Excess**(**args*, ***kwargs*)
Bases: oemof.solph.network.Sink, *oemof.tabular.facades.Facade*

class oemof.tabular.facades.**ExtractionTurbine**(**args*, ***kwargs*)
Bases: oemof.solph.components.ExtractionTurbineCHP, *oemof.tabular.facades.Facade*

Combined Heat and Power (extraction) unit with one input and two outputs.

Parameters

- **electricity_bus** (*oemof.solph.Bus*) – An oemof bus instance where the chp unit is connected to with its electrical output
- **heat_bus** (*oemof.solph.Bus*) – An oemof bus instance where the chp unit is connected to with its thermal output
- **fuel_bus** (*oemof.solph.Bus*) – An oemof bus instance where the chp unit is connected to with its input
- **carrier_cost** (*numeric*) – Cost per unit of used input carrier

- **capacity** (*numeric*) – The electrical capacity of the chp unit (e.g. in MW) in full extraction mode.
- **electric_efficiency** – Electrical efficiency of the chp unit in full backpressure mode
- **thermal_efficiency** – Thermal efficiency of the chp unit in full backpressure mode
- **condensing_efficiency** – Electrical efficiency if turbine operates in full extraction mode
- **marginal_cost** (*numeric*) – Marginal cost for one unit of produced electrical output E.g. for a powerplant: marginal cost =fuel cost + operational cost + co2 cost (in Euro / MWh) if timestep length is one hour.
- **capacity_cost** (*numeric*) – Investment costs per unit of electrical capacity (e.g. Euro / MW) . If capacity is not set, this value will be used for optimizing the chp capacity.

build_solph_components()

class oemof.tabular.facades.**Facade**(**args*, ***kwargs*)
Bases: oemof.network.Node

Parameters **_facade_requires_** (*list of str*) – A list of required attributes. The constructor checks whether these are present as keywort arguments or whether they are already present on self (which means they have been set by constructors of subclasses) and raises an error if he doesn't find them.

update()

class oemof.tabular.facades.**Generator**(**args*, ***kwargs*)
Bases: *oemof.tabular.facades.Dispatchable*

class oemof.tabular.facades.**Link**(**args*, ***kwargs*)
Bases: oemof.solph.custom.Link, *oemof.tabular.facades.Facade*

Bi-direction link for two buses (e.g. to model transshipment)

Parameters

- **from_bus** (*oemof.solph.Bus*) – An oemof bus instance where the link unit is connected to with its input.
- **to_bus** (*oemof.solph.Bus*) – An oemof bus instance where the link unit is connected to with its output.
- **capacity** (*numeric*) – The maximal capacity (output side each) of the unit. If not set, attr *capacity_cost* needs to be set.
- **loss** – Relative loss through the link (default: 0)
- **capacity_cost** (*numeric*) – Investment costs per unit of output capacity. If capacity is not set, this value will be used for optimizing the chp capacity.

build_solph_components()

class oemof.tabular.facades.**Load**(**args*, ***kwargs*)
Bases: oemof.solph.network.Sink, *oemof.tabular.facades.Facade*

Load object with one input

Parameters

- **bus** (*oemof.solph.Bus*) – An oemof bus instance where the demand is connected to.
- **amount** (*numeric*) – The total amount for the timehorzion (e.g. in MWh)

- **profile** (*array-like*) – Load profile with normed values such that *profile[t] * amount* yields the load in timestep t (e.g. in MWh)
- **input_parameters** (*dirct (optional)*)

build_solph_components()

class oemof.tabular.facades.**Reservoir**(**args*, ***kwargs*)

Bases: oemof.solph.components.GenericStorage, *oemof.tabular.facades.Facade*

A Reservoir storage unit, that is initially half full.

Note that the investment option is not available for this facade at the current development state.

Parameters

- **bus** (*oemof.solph.Bus*) – An oemof bus instance where the storage unit is connected to.
- **storage_capacity** (*numeric*) – The total storage capacity of the storage (e.g. in MWh)
- **capacity** (*numeric*) – Installed production capacity of the turbine installed at the reservoir
- **efficiency** (*numeric*) – Efficiency of the turbine converting inflow to electricity production, default: 1
- **profile** (*array-like*) – Absolute profile of inflow into the storage
- **input_parameters** (*dict*) – Dictionary to specifiy parameters on the input edge. You can use all keys that are available for the oemof.solph.network.Flow class.
- **output_parameters** (*dict*) – see: input_parameters

build_solph_components()

class oemof.tabular.facades.**Shortage**(**args*, ***kwargs*)

Bases: *oemof.tabular.facades.Dispatchable*

class oemof.tabular.facades.**Storage**(**args*, ***kwargs*)

Bases: oemof.solph.components.GenericStorage, *oemof.tabular.facades.Facade*

Storage unit

Parameters

- **bus** (*oemof.solph.Bus*) – An oemof bus instance where the storage unit is connected to.
- **storage_capacity** (*numeric*) – The total capacity of the storage (e.g. in MWh)
- **capacity** (*numeric*) – Maximum production capacity (e.g. in MW)
- **capacity_ratio** (*numeric*) – Ratio between *storage_capacity* and *capacity*
- **efficiency** (*numeric*) – Efficiency of charging and discharging process: Default: 1
- **capacity_cost** (*numeric*) – Investment costs for the storage unit e.g in €/MW-capacity
- **loss** (*numeric*) – Standing loss per timestep in % of capacity. Default: 0
- **storage_capacity_initial** (*numeric*) – The state of the storage capacity in the first (and last) time step of optimization. Default: 0.5
- **input_parameters** (*dict (optional)*) – Set parameters on the input edge of the storage (see oemof.solph for more information on possible parameters)
- **ouput_parameters** (*dict (optional)*) – Set parameters on the output edge of the storage (see oemof.solph for more information on possible parameters)

build_solph_components()

class oemof.tabular.facades.**Volatile**(**args*, ***kwargs*)

Bases: oemof.solph.network.Source, *oemof.tabular.facades.Facade*

Volatile element with one output, for example a wind turbine

Parameters

- **bus** (*oemof.solph.Bus*) – An oemof bus instance where the generator is connected to
- **capacity** (*numeric*) – The installed power of the unit (e.g. in MW).
- **profile** (*array-like*) – Profile of the output such that profile[t] * capacity yields output for timestep t
- **marginal_cost** (*numeric*) – Marginal cost for one unit of produced output, i.e. for a power-plant: mc = fuel_cost + co2_cost + . . . (in Euro / MWh) if timestep length is one hour.
- **capacity_cost** (*numeric (optional)*) – Investment costs per unit of capacity (e.g. Euro / MW) . If capacity is not set, this value will be used for optimizing the generators capacity.
- **output_paramerters** (*dict (optional)*) – Parameters to set on the output edge of the component (see. oemof.solph Edge/Flow class for possible arguments)
- **capacity_potential** (*numeric*) – Max install capacity if investment

build_solph_components()

CHAPTER 5

Contributing

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

5.1 Bug reports

When reporting a bug please include:

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

5.2 Documentation improvements

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

5.3 Feature requests and feedback

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

If you are proposing a feature:

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

5.4 Development

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

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

```
git clone git@github.com:your_name_here/oemof-tabular.git
```

3. Create a branch for local development:

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

   Now you can make your changes locally.

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

```
tox
```

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

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

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

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

For merging, you should:

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

5.4.2 Tips

To run a subset of tests:

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

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

```
detox
```

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

CHAPTER 6

Authors

- Stephan Günther - https://oemof.org
- Simon Hilpert
- Martin Söthe
- Cord Kaldemeyer

CHAPTER 7

Changelog

7.1 0.0.1 (2018-12-12)

- Moved the datapackage reader from core *oemof* into this package. That means the basic functionality of deserializing energy systems from datapackages has finally arrived.
- Moved *Facade* classes from *renpass* into this package. The *Facade* classes are designed to complement the datapackage reader, by enabling easy construction of energy system components from simple tabular data sources.
- Also moved the example datapackages from *renpass* into this package. These datapackages provide a good way of at least testing, that the datapackage reader doesn't throw errors.

7.2 0.0.0 (2018-11-23)

- First release on PyPI. Pretty much non functional because it only consists of the package boilerplate and nothing else. But this is what a version zero is for, IMHO.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

P

R

S

T

U

V

W